



# oneAPI GPU Optimization Guide

Oct 29, 2021



# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Productive Performance not Performance Portability . . . . .	3
1.2	Phases in the Optimization Workflow . . . . .	3
1.3	Profiling and Tuning your Code . . . . .	4
<b>2</b>	<b>Getting Started</b>	<b>5</b>
2.1	1. Remember Amdahl's Law . . . . .	5
2.2	2. Locality Matters . . . . .	5
2.3	3. Rightsize Your Work . . . . .	6
<b>3</b>	<b>Parallelization</b>	<b>7</b>
3.1	Use a Parallel Programming Language or API . . . . .	7
3.2	Parallelizing Compilers . . . . .	7
3.3	Parallel Libraries . . . . .	7
<b>4</b>	<b>Intel® Processors with Integrated Graphics</b>	<b>9</b>
4.1	Execution Unit . . . . .	9
4.2	SubSlice . . . . .	10
4.3	Slice . . . . .	11
4.4	Architecture Parameters across Generations . . . . .	13
<b>5</b>	<b>DPC++ Thread Hierarchy and Mapping</b>	<b>15</b>
5.1	nd_range . . . . .	15
5.2	Thread Synchronization . . . . .	16
5.3	Mapping Work-groups to SubSlices . . . . .	16
<b>6</b>	<b>Kernels</b>	<b>27</b>
6.1	Reduction . . . . .	27
6.2	Sub-groups . . . . .	34
6.3	Avoiding Register Spills . . . . .	43
6.4	Shared Local Memory . . . . .	49
6.5	Removing Conditional Checks . . . . .	57
6.6	Kernel Launch . . . . .	60
6.7	Using Libraries for Accelerator Offload . . . . .	62
6.8	Using Standard Library Functions in DPC++ Kernels . . . . .	65
6.9	Efficiently Implementing Fourier Correlation Using oneAPI Math Kernel Library (oneMKL) . . . . .	68
6.10	Executing Multiple Kernels on the Device at the Same Time . . . . .	77
6.11	Synchronization among Threads in a Kernel . . . . .	80
6.12	Restrict Directive . . . . .	89

6.13	Submitting Kernels to Multiple Queues . . . . .	94
6.14	Avoid Redundant Queue Construction . . . . .	98
6.15	Considerations for selecting work-group size . . . . .	102
<b>7</b>	<b>Memory</b>	<b>107</b>
7.1	Performance Impact of USM and Buffers . . . . .	107
7.2	Optimizing Memory Movement between Host and Accelerator . . . . .	111
7.3	Avoid moving data back and forth between host and device . . . . .	115
7.4	Avoid Declaring Buffers in a Loop . . . . .	119
7.5	Buffer Accessor Modes . . . . .	122
<b>8</b>	<b>Host/Device Coordination</b>	<b>131</b>
8.1	Asynchronous and Overlapping Data Transfers between Host and Device . . . . .	131
<b>9</b>	<b>Using multiple heterogeneous devices</b>	<b>137</b>
9.1	Overlapping Compute on Various Accelerators in the Platform . . . . .	137
<b>10</b>	<b>Compilation</b>	<b>141</b>
10.1	Just-In-Time Compilation in DPC++ . . . . .	141
10.2	Specialization Constants . . . . .	144
<b>11</b>	<b>Debugging and Profiling</b>	<b>149</b>
11.1	GPU Analysis with Intel® VTune™ Profiler . . . . .	149
11.2	Intel® Advisor GPU Analysis . . . . .	158
11.3	Doing IO in the kernel . . . . .	166
11.4	Using the timers . . . . .	170
11.5	How to Use the Intercept Layer for OpenCL™ Applications . . . . .	172
<b>12</b>	<b>NDA Prerelease Systems Tuning Guide</b>	<b>177</b>
12.1	Multi-GPU, Multi-Tile and Multi-C-Slice Architecture and Programming . . . . .	177
12.2	Implicit Scaling on Intel® Multi-Tile GPU . . . . .	194
12.3	Explicit Scaling on Multi-GPU, Multi-Tile and Multi-C-Slice . . . . .	208
12.4	OpenMP Offloading Tuning Guide . . . . .	222
12.5	NDA capabilities of VTune™ Profiler . . . . .	284
12.6	AOT Compiler Flags for ATS B0/A0 . . . . .	293
<b>13</b>	<b>Reference</b>	<b>295</b>
<b>14</b>	<b>Terms and Conditions</b>	<b>297</b>

## List of Figures

1	Execution Unit . . . . .	9
2	SubSlice . . . . .	10
3	Intel® Iris® Xe Graphics on Intel Ice Lake processors, one Slice . . . . .	12
4	Thread hierarchy . . . . .	15
5	Occupancy for VectorAdd2 as shown by VTune . . . . .	22
6	VectorAdd2 timeline view . . . . .	23
7	GPU occupancy VectorAdd3, VectorAdd4 kernels . . . . .	25
8	Thread occupancy metric on Vtune . . . . .	26
9	Each Work Item Has 256 Private Histogram Bins . . . . .	47
10	Sub-group Has 256 Private Histogram Bins . . . . .	47
11	Timeline of kernel execution . . . . .	62
12	Functions called on host to submit the kernel . . . . .	62
13	Timeline for kernels executed with in-order and out-of-order queues . . . . .	80
14	VTune dynamic instruction . . . . .	83
15	VTune atomic int . . . . .	84
16	VTune atomic double . . . . .	85
17	cllIntercept showing copy-in overlap with execution of compute kernel . . . . .	136
18	GPU Offload viewpoint . . . . .	151
19	GPU Utilization . . . . .	152
20	GPU Offload Platform window . . . . .	153
21	Computing Task grouping . . . . .	153
22	GPU Compute/Media Hotspots analysis . . . . .	154
23	Characterization profile . . . . .	155
24	VTune Profiler memory bandwidth metrics . . . . .	155
25	GPU Compute/Media Hotspots analysis, Source Analysis . . . . .	156
26	Offload Modelling for a C++ Matrix application on Linux*OS . . . . .	159
27	Intel Advisor GUI, Offload Advisor . . . . .	160
28	Top Hotspots pane . . . . .	162
29	Performance Characteristics pane . . . . .	163
30	GPU Roofline chart and performance metrics . . . . .	163
31	GPU Roofline new chart . . . . .	166
32	General Purpose Compute Model . . . . .	177
33	Intel® Iris® Xe GPU Multi-tile Architecture . . . . .	180
34	GPU Memory System . . . . .	181
35	EU thread and Memory Access . . . . .	182



## List of Tables

1	SubSlice computing capacity . . . . .	11
2	Key architecture parameters, Intel UHD Graphics . . . . .	13
3	Intel® Iris® Xe Graphics (TGL) GPU . . . . .	16
4	Configurations to ensure full occupancy . . . . .	17
5	Utilization for various configurations . . . . .	18
6	Occupancy . . . . .	22
7	Number of thread contexts available by device . . . . .	28
8	Matrix A Dimensions (Time in Seconds) . . . . .	64
9	PCIe bandwidth by generation . . . . .	131
10	Host Performance Timing Results . . . . .	174
11	Device Performance Timing Results for Intel(R) Gen9 HD Graphics NEO (24CUs, 1200MHz) . .	174
12	Memory latency table . . . . .	182
13	Atomics performance of acquire-release vs relaxed memory order . . . . .	186
14	Memory allocation types and characteristics . . . . .	186
15	Terminologies . . . . .	188
16	Work group partition to tiles . . . . .	197
17	Measured bandwidth with 1D kernel launch . . . . .	200
18	Measured bandwidth with 3D kernel launch . . . . .	202
19	Measured bandwidth with cross-tile accesses . . . . .	206



Welcome to the NDA oneAPI GPU Optimization Guide! This document provides tips for getting the best performance for oneAPI programs on GPUs.



## 1.0 Introduction

Designing high-performance software requires you to “think differently” than you might normally do when writing software. You need to be aware of the hardware on which your code is intended to run, and the characteristics which control the performance of that hardware. Your goal is to structure the code such that it produces correct answers, but does so in a way that maximizes the hardware’s ability to execute the code.

oneAPI is a cross-industry, open, standards-based unified programming model that delivers a common developer experience across accelerator architectures. A unique feature of accelerators is that they are additive to the main CPU on the platform. The primary benefit of using an accelerator is to improve the behavior of your software by partitioning it across the host and accelerator to specialize portions of the computation that run best on the accelerator. Accelerator architectures can offer a benefit through specialization of compute hardware for certain classes of computations. This enables them to deliver best results for software specialized to the accelerator architecture.

The primary focus of this document is GPUs. Each section focuses on different topics to guide you in your path to creating optimized solutions. The Intel® oneAPI toolkits provide the languages and development tools you will use to optimize your code. This includes compilers, debuggers, profilers, analyzers, and libraries.

### 1.1 Productive Performance not Performance Portability

While this document focuses on GPUs, you may also need your application to run on CPUs and other types of accelerators. Since accelerator architectures are specialized, you need to specialize your code to achieve best performance. Specialization includes restructuring and tuning the code to create the best mapping of the application to the hardware. In extreme cases, this may require redesigning your algorithms for each accelerator to best expose the right type of computation. The value of oneAPI is that it allows each of these variations to be expressed in a common language with device-specific variants launched on the appropriate accelerator.

### 1.2 Phases in the Optimization Workflow

The first phase in using a GPU is to identify which parts of the application can benefit. This is usually compute-intensive code that has the right ratio of memory accesses to computation, and has the right data dependence patterns to map onto the GPU. GPUs include local memory and typically provide massive parallelism. This determines which characteristics of the code are most important when deciding what to offload.

The Intel® Advisor tool included in the Intel® oneAPI Base Toolkit is designed to analyze your code and help you identify the best opportunities for parallel execution. The profilers in Advisor measure the data movement in your functions, the memory access patterns, and the amount of computation in order to project how code will perform when mapped onto different accelerators. The regions with highest potential benefit should be your first targets for acceleration.

GPUs often exploit parallelism at multiple levels. This includes overlap between host and GPU, parallelism across the compute cores, overlap between compute and memory accesses, concurrent pipelines, and vector computation. Using all these levels of parallelism requires a good understanding of the GPU architecture and capabilities in the libraries and languages at one’s disposal.

**Keep all the compute resources busy.** There must be enough independent tasks to saturate the device and fully utilize all execution resources. For example, if the device has 100 compute cores but you only have one task, 99% of the device will be idle. Often you create many more independent tasks than available compute resources so that the hardware can schedule more work as prior tasks complete.

**Minimize the synchronization between the host and the device.** The host launches a kernel on the device and waits for its completion. Launching a kernel incurs overhead, so structure the computation to minimize the number of times a kernel is launched.

**Minimize the data transfer between host and device.** Data typically starts on the host and is copied to the device as input to the computation. When a computation is finished, the results must be transferred back to the host. For best performance, minimize data transfer by keeping intermediate results on the device between computations. Reduce the impact of data transfer by overlapping computation and data movement so the compute cores never have to wait for data.

**Keep the data in faster memory and use an appropriate access pattern.** GPU architectures have different types of memory which have different access costs. Registers, caches, and scratchpads are cheaper to access than local memory, but have smaller capacity. When data is loaded into a register, cache line, or memory page, use an access pattern that will use all the data before moving to the next chunk. When memory is banked, use a stride that avoids all the compute cores trying to access the same memory bank simultaneously.

## 1.3 Profiling and Tuning your Code

After you have designed your code for high performance, the next step is to measure how it runs on the target accelerator. Add timers to the code, collect traces, and use tools like Intel® VTune™ Profiler to observe the program as it runs. The information collected can identify where hardware is bottlenecked and idle, illustrate how behavior compares with peak hardware roofline, and identify the most important hotspots to focus optimization effort.

## 2.0 Getting Started

If you only have time to read this far, then you at least need to know the three big concepts to optimize software for an accelerator.

### 2.1 1. Remember Amdahl's Law

This may appear obvious, but it is the first step in making use of an accelerator. Amdahl's law states that the fraction of time an application uses an accelerator ( $F_p$ ) limits the benefit of acceleration. The maximum speedup is bounded by  $1/(1 - F_p)$ . If you use the accelerator 50% of the time, you will get at most a  $2\times$  speedup, even with an infinitely powerful accelerator.

Note here that this is in terms of your program execution, not your program's source code. The parallel kernels may represent a very small fraction of your overall source code, but if this is where your execution time is concentrated you can still do well.

### 2.2 2. Locality Matters

An accelerator often has specialized memory with a disjoint address space. An application must allocate or move data into the right memory at the right time.

Accelerator memory is arranged in a hierarchy. Registers are more efficient to access than caches, which are more efficient to access than main memory. Bringing data closer to the point of execution improves efficiency.

There are many ways you can refactor your code to get your data closer to the execution. They will be outlined in the following sections. Here, we focus on three:

1. Allocate your data on the accelerator, and when copied there, keep it resident for as long as possible. Your application may have many offloaded regions. If you have data that is common between these regions, it makes sense to amortize the cost of the first copy, and just reuse it in place for the remaining kernel invocations.
2. Access contiguous blocks of memory as your kernel executes. The hardware will fetch contiguous blocks into the memory hierarchy, so you have already paid the cost for the entire block. After you use the first element of the block, the remaining elements are almost free to access so take advantage of it.
3. Restructure your code into blocks with higher data reuse. In a two-dimensional matrix, you can arrange your work to process one block of elements before moving onto the next block of elements. For example, in a stencil operation you may access the prior row, the current row, and the next row. As you walk over the elements in a block you reuse the data and avoid the cost of requesting it again.

## 2.3 3. Rightsize Your Work

Data-parallel accelerators are designed as throughput engines and are often specialized by replicating execution units many times. This is an easy way of getting higher performance on data-parallel algorithms since more of the elements can be processed at the same time.

However, fully utilizing a parallel processor can be challenging. For example, imaging you have 512 execution units, where each execution unit had eight threads, and each thread has 16-element vectors. You need to have a minimum of  $512 \times 8 \times 16 = 65536$  parallel activities scheduled at all times just to match this capacity. In addition, if each parallel activity is small, you need another large factor to amortize the cost of submitting this work to the accelerator. Fully utilizing a single large accelerator may require decomposing a computation into millions of parallel activities.

## 3.0 Parallelization

Parallelism is essential to effective use of accelerators because they contain many independent processing elements that are capable of executing code in parallel. There are three ways to develop parallel code.

### 3.1 Use a Parallel Programming Language or API

There are many parallel programming languages and APIs that can be used to express parallelism. oneAPI supports parallel program development through the Data Parallel C++ (DPC++) language. oneAPI also has a number of code generation tools to convert these programs into binaries that can be executed on different accelerators. The usual workflow is that a user starts with a serial program, identifies the parts of the code that take a long time to execute (referred to as hotspots), and converts them into parallel kernels that can be offloaded to an accelerator for execution.

### 3.2 Parallelizing Compilers

Directive-based approaches like OpenMP are another way to develop parallel programs. In a directive-based approach, the programmer provides hints to the compiler about parallelism without modifying the code explicitly. This approach is easier than developing a parallel program from first principles.

### 3.3 Parallel Libraries

oneAPI includes a number of libraries like oneTBB, oneMKL, oneDNN, oneVPL etc. that provide highly-optimized versions of common computational operations that run across a variety of accelerator architectures. Depending on the needs of the application a user can directly call the functions from these libraries and get efficient implementations of these for the underlying architecture. This is the easiest approach to developing parallel programs provided the library contains the required functions. For example, machine learning applications can take advantage of the optimized primitives in oneDNN. These libraries have been thoroughly tested for both correctness and performance, which makes programs more reliable when using them.

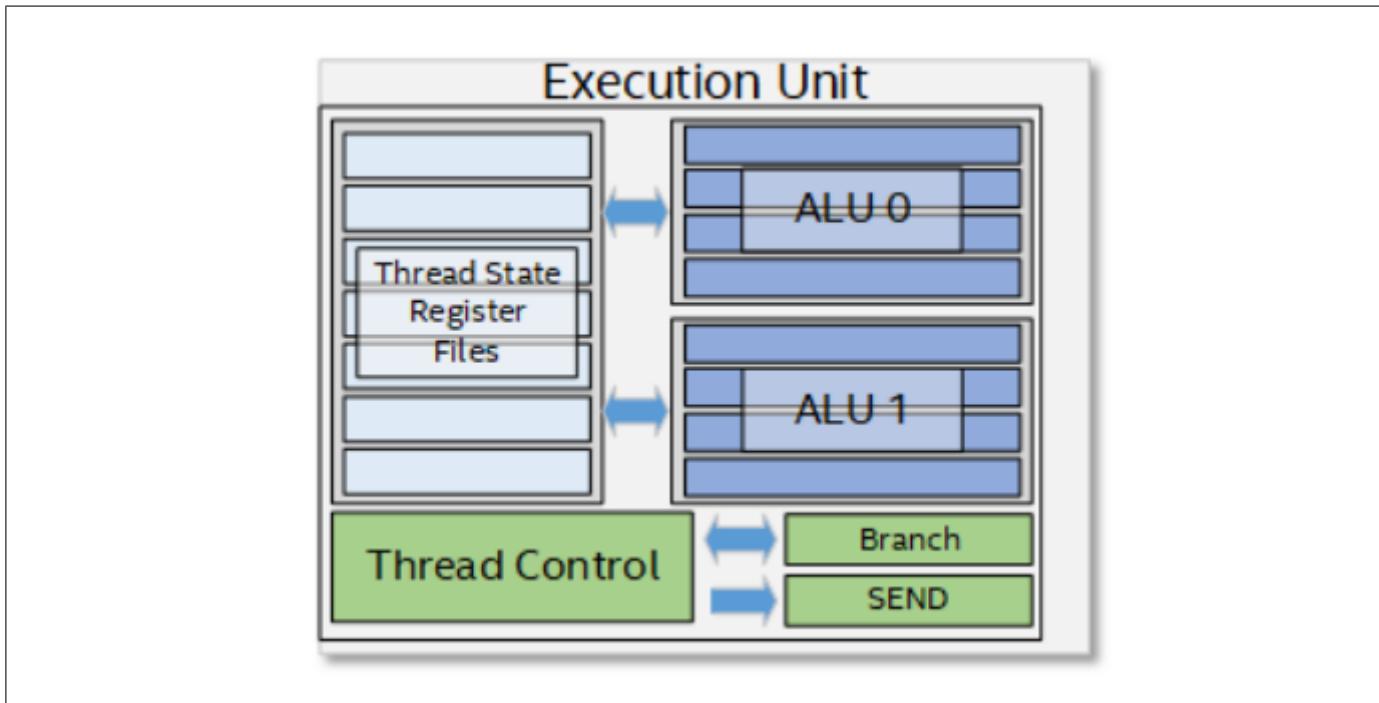


## 4.0 Intel® Processors with Integrated Graphics

Intel® UHD Graphics is a proprietary Intel technology that provides graphics, compute, media, and display capabilities for many Intel® processors. This section focuses on the compute components of UHD Graphics architecture.

### 4.1 Execution Unit

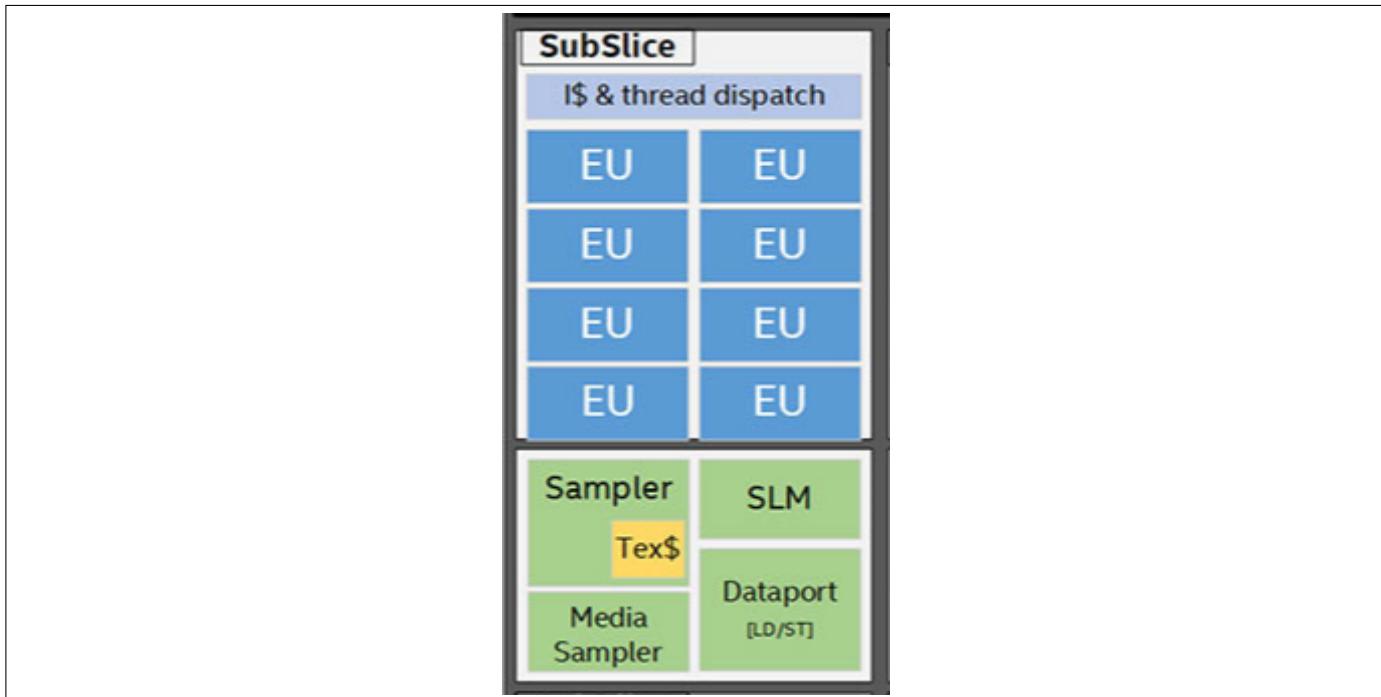
An Intel GPU consists of a set of execution units (EU). Each EU is simultaneously multithreaded (SMT) with seven threads. The primary computation units are a pair of Single Instruction Multiple Data (SIMD) Arithmetic Logic Units (ALU). Each ALU can execute up to four 32-bit floating-point or integer operations, or eight 16-bit floating-point operations. Effectively, each EU can execute eight SIMD F32 vector operations (16 with fused multiply and add) and 16 SIMD F16 (32 with fused multiply and add). Each hardware thread has 128 general-purpose registers (GRF) of 32B wide for the total of 4KB register/thread or 28KB/EU. Each GFR can hold a vector of one, two, four, or eight 32-bit floating point or integer values or 16 16-bit values. For convenience, we can model an EU as executing seven threads with eight SIMD lanes for a total of 56 concurrent 32-bit operations or 112 concurrent 16-bit operations.



**Fig.1:** Execution Unit

## 4.2 SubSlice

Each SubSlice contains an EU array of 8 EUs for Ice Lake with Iris Xe Graphics (ICX) to 16 EUs for Tiger Lake with Iris Xe Graphics (TGL). Hence each SubSlice can perform 448 (ICX) to 896 (TGL) concurrent 32-bit operations. In addition to EUs, each SubSlice also contains an instruction cache, a local thread dispatcher, a read-only texture/image sampler of 64B/cycle, a Dataport of 64B/cycle for both read and write, and 64KB of shared local memory (SLM). The Dataport's read bandwidth of 64B/cycle averages to 8B/cycle/EU or two FP32 inputs/cycle/EU. For maximum performance, compute kernels require a high computation to memory access ratio and must reuse data whenever possible. For read-only data, it is possible to use the sampler unit to get an additional 64B/cycle data inputs into a SubSlice. The total read bandwidth across Dataport, Sampler and SLM is 192B/cycle.



**Fig. 2:** SubSlice

The SLM is a 64KB highly banked memory accessible from the EUs in the SubSlice. One important usage of SLM is to share global atomic data among all the 448 (ICX) to 896 (TGL) concurrent work-items executing in a SubSlice. For this reason, if a kernel's work-group contains synchronization operations, all work-items of the work-group must be allocated to a single SubSlice so that they have shared access to the same 64KB SLM. The work-group size must be chosen carefully to maximize the occupancy and utilization of the SubSlice. In contrast, if a kernel does not access SLM, its work-items can be dispatched across multiple SubSlices for high occupancy and utilization.

The maximum number of work-groups that can be executed on a single SubSlice is 16. Small work-groups may make it impossible to achieve full occupancy on a SubSlice.

The following table summarizes the computing capacity of a SubSlice.

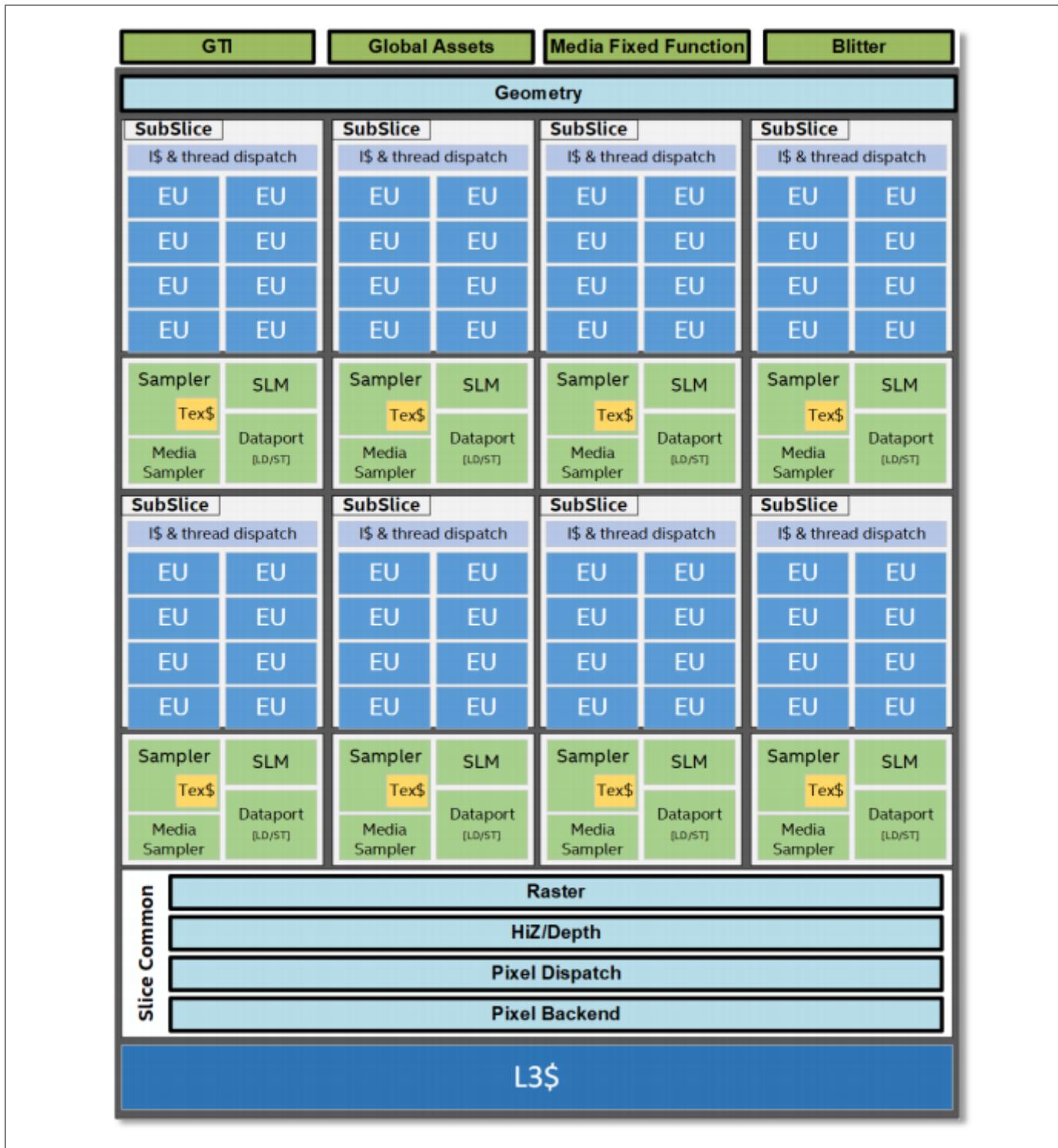
**Table 1:** SubSlice computing capacity

<b>GPU Generation</b>	<b>EUs</b>	<b>Threads</b>	<b>Operations</b>	<b>Maximum Work Groups</b>
Intel Iris Xe ICX	8	$7 \times 8 = 56$	$56 \times 8 = 448$	16
Intel Iris Xe TGL	16	$7 \times 16 = 112$	$112 \times 8 = 896$	16

## 4.3 Slice

On the Intel® Iris® Xe Graphics (ICX) GPU, eight SubSlices form a Slice for an aggregated total of 64 EU, or 3,584 simultaneous computations. Each Slice also contains a shared 3,072KB L3 cache and other slice-common units for graphics and media processing. The Intel® Iris® Xe Graphics on Intel Ice Lake processors contains one Slice with 64 EUs as illustrated below, which amounts to 3,584 simultaneous computations. For good performance, an application must keep the EU occupancy high with thousands of work-items.

Note that the number of SubSlices in each Intel® Iris® Xe Graphics generation is subject to changes. The GPUs on Intel Tiger Lake processors contain six SubSlices where each SubSlice contains sixteen EUs. The coming generations of Intel CPUs with Intel UHD Graphics contain multiple Slices to further scale computation capacities.



**Fig. 3:** Intel® Iris® Xe Graphics on Intel Ice Lake processors,  
one Slice

## 4.4 Architecture Parameters across Generations

The following table summarizes the key architecture parameters in the current released products with Intel UHD Graphics:

**Table 2:** Key architecture parameters, Intel UHD Graphics

Generations	Threads per EU	EUs per Sub-Slice	Sub-Slices	Total Threads	Total Operations
Gen9 (BDW)	7	8	3	168	1344
Intel Iris Xe (Gen11)	7	8	8	448	3584
Intel Iris Xe (Gen12)	7	16	6	672	5376



## 5.0 DPC++ Thread Hierarchy and Mapping

The DPC++ execution model exposes an abstract view of GPUs with Intel® Architecture. The DPC++ thread hierarchy consists of a 1-, 2-, or 3-dimensional grid of work-items. These work-items are grouped into equal sized thread groups called work-groups. Threads in a work-group are further divided into equal sized vector groups called sub-groups.

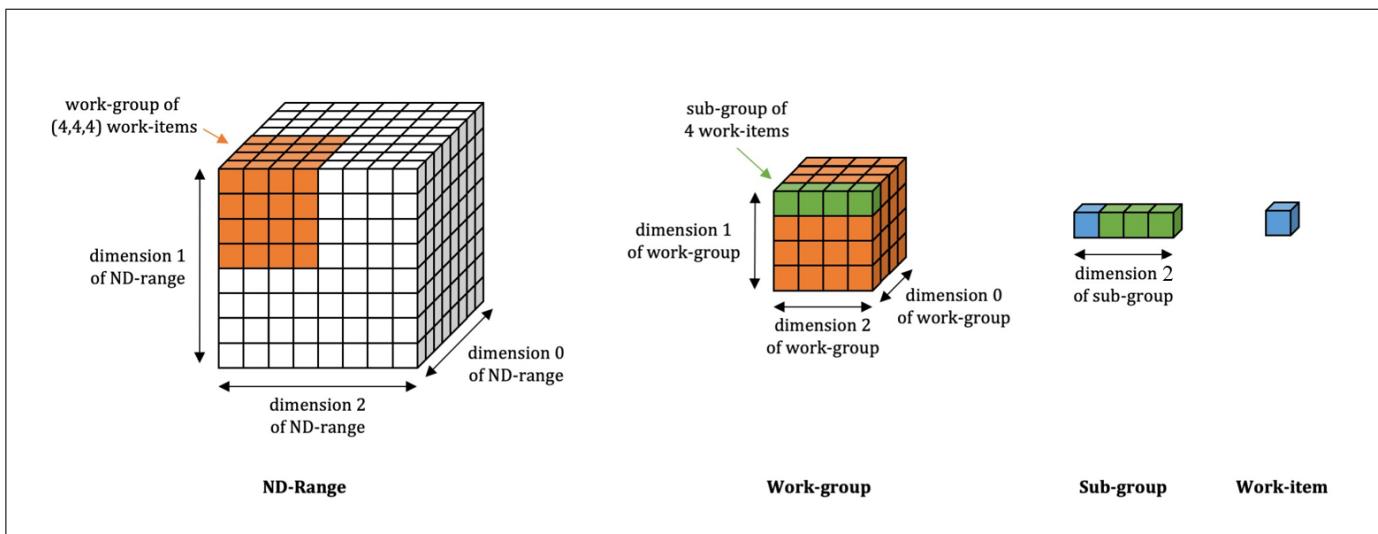
**Work-item** A work-item represents one of a collection of parallel executions of a kernel.

**Sub-group** A sub-group represents a short range of consecutive work-items that are processed together as a SIMD vector of length 8, 16, 32, or a multiple of the native vector length of a CPU with Intel® UHD Graphics.

**Work-group** A work-group is a 1-, 2-, or 3-dimensional set of threads within the thread hierarchy. In DPC++, synchronization across work-items is only possible with barriers for the work-items within the same work-group.

### 5.1 nd\_range

An `nd_range` divides the thread hierarchy into 1-, 2-, or 3-dimensional grids of work-groups. It is represented by the global range, the local range of each work-group.



**Fig. 4:** Thread hierarchy

The diagram above illustrates the relationship among ND-Range, work-group, sub-group, and work-item.

## 5.2 Thread Synchronization

DPC++ provides two synchronization mechanisms that can be called within a kernel function. Both are only defined for work-items within the same work-group. DPC++ does not provide any synchronization mechanism inside a kernel across all work-items across the entire `nd_range`.

`mem_fence` inserts a memory fence on global and local memory access across all work-items in a work-group.

`barrier` inserts a memory fence and blocks the execution of all work-items within the work-group until all work-items have reached its location.

## 5.3 Mapping Work-groups to SubSlices

In the rest of this chapter, we will explain how to pick a proper work-group size to maximize the occupancy of the GPU resources. We will use the Intel® Tiger Lake Processors with Intel® Iris® Xe Graphics (TGL) GPU as the execution target. From the [Key architecture parameters, Intel UHD Graphics](#) table, we summarize the architecture parameters for Intel® Iris® Xe Graphics (TGL) GPU below:

**Table 3:** Intel® Iris® Xe Graphics (TGL) GPU

	EU	Threads	Operations	Maximum Group Size	Work	Maximum Work Groups
Each Sub-Slice	16	$7 \times 16 = 112$	$112 \times 8 = 896$	512		16
Total	$16 \times 6 = 96$	$112 \times 6 = 672$	$896 \times 6 = 5376$	512		$16 \times 6 = 96$

The maximum work-group size is a constraint imposed by the hardware and GPU driver. One can query the maximum work-group using `device::get_info<cl::sycl::info::device::max_work_group_size>()` on the supported size.

Let's start with a simple kernel:

**Listing 1:** /examples/exec-model/simple.cpp

```

1 auto command_group =
2     [&](auto &cgh) {
3         cgh.parallel_for(sycl::range<3>(64, 64, 64), // global range
4                         [=](item<3> it) {
5                             // (kernel code)
6                         })
7     }

```

This kernel contains 262,144 work-items structured as a 3D range of  $64 \times 64 \times 64$ . It leaves the work-group and sub-group size selection to the compiler. To fully utilize the 5376 parallel operations available in the GPU slice, the compiler must choose a proper work group size.

The two most important GPU resources are:

**Thread Contexts** The kernel should have a sufficient number of threads to utilize the GPU's thread contexts.

**SIMD Units and SIMD Registers** The kernel should be organized to vectorize the work-items and utilize the SIMD registers.

In a DPC++ kernel, the programmer can affect the work distribution by structuring the kernel with proper work-group size, sub-group size, and organizing the work-items for efficient vector execution. Writing efficient vector kernels is covered in a separate section. In this chapter, we will only focus on work-group and sub-group size selection.

Thread contexts are easier to utilize than SIMD vector. Therefore, we start with selecting the number of threads in a work-group. Each SubSlice has 112 thread contexts, but usually we cannot use all the threads if the kernel is also vectorized by 8 ( $112 \times 8 = 896 > 512$ ). From this, we can derive that the maximum number of threads in a work-group is 64 ( $512 / 8$ ). Because each SubSlice can execute a maximum of 16 work-groups, we need at least 8 threads in a work-group ( $8 \times 16 = 128 > 112$ ) to fully utilize a SubSlice. Hence, we can derive the minimum number of threads in work-group should be 8.

DPC++ does not provide a mechanism to directly set the number of threads in a work-group. However, we can use work-group size and SIMD sub-group size to set the number of threads:

$$\text{Work group size} = \text{Threads} \times \text{SIMD sub-group size}$$

We can increase the sub-group size as long as there are a sufficient number of registers for the kernel after widening. Note that each EU has 128 SIMD8 registers. There is much room for widening on simple kernels. The effect of increasing sub-group size is similar to loop unrolling: while each EU still executes eight 32-bit operations per cycle, the amount of work per work-group interaction is doubled/quadrupled. In DPC++, a programmer can explicitly specify sub-group size using `intel::reqd_sub_group_size({8|16|32})` to override the compiler's selection.

The table below summarizes selection criteria of threads, sub-group sizes to keep all GPU resources occupied for TGL:

**Table 4:** Configurations to ensure full occupancy

Minimum Threads	Maximum Threads	Minimum Sub-group Size	Maximum Sub-group Size	Maximum Work-group Size	Constraint
8	64	8	32	512	$\text{Threads} \times \text{Sub - groupSize} \leq 512$

Back to our example program, if a programmer chooses a work-group size less than 64 for sub-group size 8, less than 128 for sub-group size 16, or less than 256 for sub-group size 32, then it would not be able to fully utilize TGL GPU's thread contexts. Choosing a larger work-group size has the additional advantage of reducing the number of rounds of work-group dispatching.

### 5.3.1 Impact of Work-item Synchronization within Work-group

Let's look at a kernel requiring work-item synchronization:

**Listing 2:** /examples/exec-model/barrier.cpp

```

1 auto command_group =
2     [&](auto &cgh) {
3         cgh.parallel_for(nd_range(sycl::range(64, 64, 128), // global range
4                                     sycl::range(1, R, 128)    // local range
5                                     ),
6                                     [=](sycl::nd_item<3> item) {
7                                         // (kernel code)
8                                         // Internal synchronization
9                                         item.barrier(access::fence_space::global_space);
10                                        // (kernel code)
11
12 })
13 }
```

This kernel is similar to the previous example, except it requires work-group barrier synchronization. Work-item synchronization is only available to work-items within the same work-group. A programmer must pick a work-group local range using `nd_range` and `nd_item`. Because synchronization is implemented using a SubSlice's SLM for shared variables, all the work-items of a work-group must be allocated to the same SubSlice, which affects SubSlice occupancy and kernel performance.

In this kernel, the local range of work-group is given as `range(1, R, 128)`. Assuming the sub-group size is eight, let's look at how the values of variable `R` affect EU occupancy. In the case of `R=1`, the local group range is `(1, 1, 128)` and work-group size is 128. The SubSlice allocated for a work-group contains only 16 threads out of 112 available thread contexts (i.e., very low occupancy). However, the system can dispatch 7 work-groups to the same SubSlices to reach full occupancy at the expense of a higher number of dispatches.

In the case of `R>4`, the work-group size will exceed the system supported maximum work-group size of 512, such that the kernel will fail to launch. In the case of `R=4`, a SubSlice is only 57% occupied (4/7) and the three unused thread contexts are not sufficient to accommodate another work-group, wasting 43% of the available EU capacities. Note that the driver may still be able to dispatch partial work-group to unused SubSlice. However, because of the barrier in the kernel, the partially dispatch work items would not be able to pass the barriers until the rest of the work group is dispatched. In most cases, the kernel's performance would not benefit much from the partial dispatch. Hence, it is important to avoid this problem by properly choosing the work-group size.

The table below summarizes the tradeoffs among group size, number of threads, SubSlice utilization, and occupancy.

**Table 5:** Utilization for various configurations

Work-items	Group Size	Threads	SubSlice Utilization	SubSlice Occupancy
$64 \times 64 \times 128 = 524288$	(R=1) 128	16	16/112 = 14%	100% with 7 work-groups
$64 \times 64 \times 128 = 524288$	(R=2) 128 × 2	$2 \times 16 = 32$	32/112 = 28.6%	86% with 3 work-groups
$64 \times 64 \times 128 = 524288$	(R=3) 128 × 4	$3 \times 16 = 48$	48/112 = 42.9%	86% with 2 work-groups
$64 \times 64 \times 128 = 524288$	(R=4) 128 × 4	$4 \times 16 = 64$	64/112 = 57%	57% maximum
$64 \times 64 \times 128 = 524288$	(R>4) 640+			Fail to launch

### 5.3.2 Impact of Local Memory within Work-group

Let's look at an example where a kernel allocates local memory for a work-group:

**Listing 3:** /examples/exec-model/local.cpp

```

1 auto command_group =
2     [&](auto &cgh) {
3         // local memory variables shared among work items
4         sycl::accessor<int, 1, sycl::access::mode::read_write,
5             sycl::access::target::local>
6         myLocal(sycl::range(R), cgh);
7         cgh.parallel_for(nd_range(sycl::range<3>(64, 64, 128), // global range
8                         sycl::range<3>(1, R, 128) // local range
9                         ),
10                    [=](ngroup<3> myGroup) {
11                        // (work group code)
12                        myLocal[myGroup.get_local_id()[1]] = ...
13                    })
14    }

```

Because work-group local variables are shared among its work-items, they are allocated in a SubSlice's SLM. Therefore, this work-group must be allocated to a single SubSlice, same as the intra-group synchronization. In addition, one must also weigh the sizes of local variables under different group size options such that the local variables fit within a SubSlice's 64KB SLM capacity limit.

### 5.3.3 A Detailed Example

Before we conclude this section, let's look at the hardware occupancies from the variants of a simple vector add example. Using Intel® Iris® Xe graphics from TGL platform as the underlying hardware with the resource parameters specified in [Intel® Iris® Xe Graphics \(TGL\) GPU](#).

**Listing 4:** /examples/exec-model/vec-add.cpp

```

1 int VectorAdd1(sycl::queue &q, const IntArray &a, const IntArray &b,
2                 IntArray &sum, int iter) {
3     sycl::range num_items{a.size()};
4
5     sycl::buffer a_buf(a);
6     sycl::buffer b_buf(b);
7     sycl::buffer sum_buf(sum.data(), num_items);
8
9     auto start = std::chrono::steady_clock::now();
10    auto e = q.submit([&](auto &h) {
11        // Input accessors
12        sycl::accessor a_acc(a_buf, h, sycl::read_only);
13        sycl::accessor b_acc(b_buf, h, sycl::read_only);
14        // Output accessor
15        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
16
17        h.parallel_for(num_items, [=](auto i) {
18            for (int j = 0; j < iter; j++)

```

(continues on next page)

(continued from previous page)

```

19     sum_acc[i] = a_acc[i] + b_acc[i];
20   });
21 });
22 q.wait();
23 auto end = std::chrono::steady_clock::now();
24 std::cout << "VectorAdd1 completed on device - took " << (end - start).count()
25       << " u-secs\n";
26 return ((end - start).count());
27 } // end VectorAdd1

```

The VectorAdd1 above lets the compiler select the work-group size and SIMD width. In this case, the compiler selects a work-group size of 512 and a SIMD width of 32 because the kernel's register pressure is low.

**Listing 5:** /examples/exec-model/vec-add.cpp

```

1 int VectorAdd2(sycl::queue &q, const IntArray &a, const IntArray &b,
2                 IntArray &sum, int iter) {
3     sycl::range num_items{a.size()};
4
5     sycl::buffer a_buf(a);
6     sycl::buffer b_buf(b);
7     sycl::buffer sum_buf(sum.data(), num_items);
8     size_t num_groups = groups;
9     size_t wg_size = 512;
10    // get the max wg_size instead of 512 size_t wg_size = 512;
11    auto start = std::chrono::steady_clock::now();
12    q.submit([&](auto &h) {
13        // Input accessors
14        sycl::accessor a_acc(a_buf, h, sycl::read_only);
15        sycl::accessor b_acc(b_buf, h, sycl::read_only);
16        // Output accessor
17        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
18
19        h.parallel_for(
20            sycl::nd_range<1>(num_groups * wg_size, wg_size), [=
21            ](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(32)]] {
22                size_t grp_id = index.get_group()[0];
23                size_t loc_id = index.get_local_id();
24                size_t start = grp_id * mysize;
25                size_t end = start + mysize;
26                for (int i = 0; i < iter; i++)
27                    for (size_t i = start + loc_id; i < end; i += wg_size) {
28                        sum_acc[i] = a_acc[i] + b_acc[i];
29                    }
30            });
31    });
32    q.wait();
33    auto end = std::chrono::steady_clock::now();
34    std::cout << "VectorAdd2<" << groups << "> completed on device - took "
35          << (end - start).count() << " u-secs\n";
36    return ((end - start).count());
37 } // end VectorAdd2

```

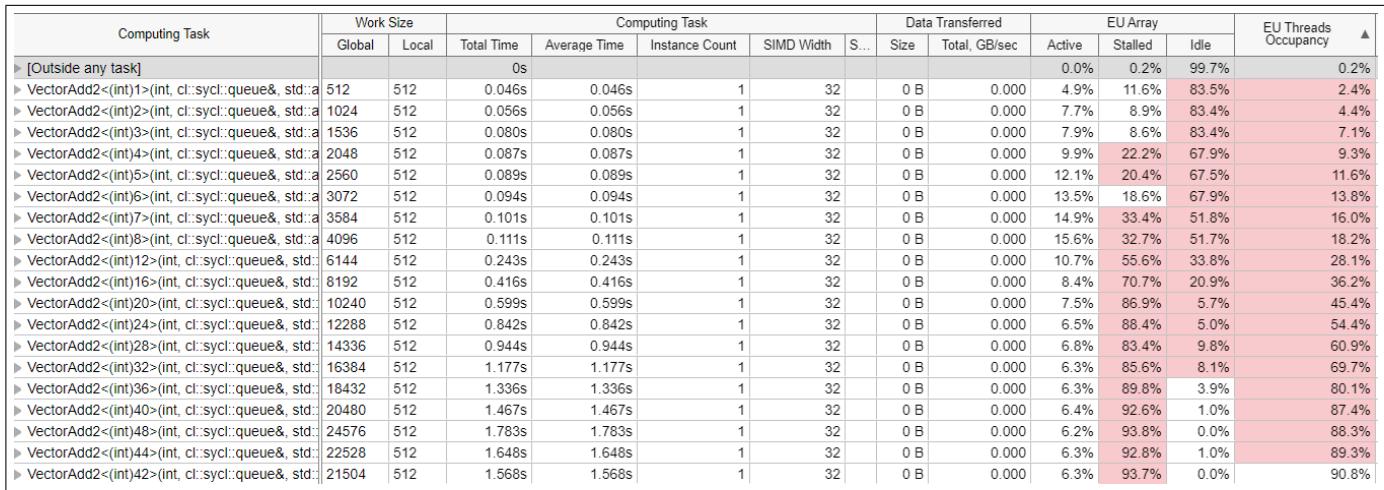
In the `VectorAdd2` above, we explicitly specify the work-group size of 512, SIMD width of 32, and a variable number of work-groups as a function parameter groups.

In the absence of intra-work group synchronization, we know that threads from any work-group can be dispatched to any SubSlice. Dividing the number of threads by the number of available thread contexts in the GPU gives us an estimate of the GPU hardware occupancy. The following table calculates the GPU hardware occupancy using the TGL Intel® Iris® Xe architecture parameters for each of the above two kernels with various arguments.

**Table 6:** Occupancy

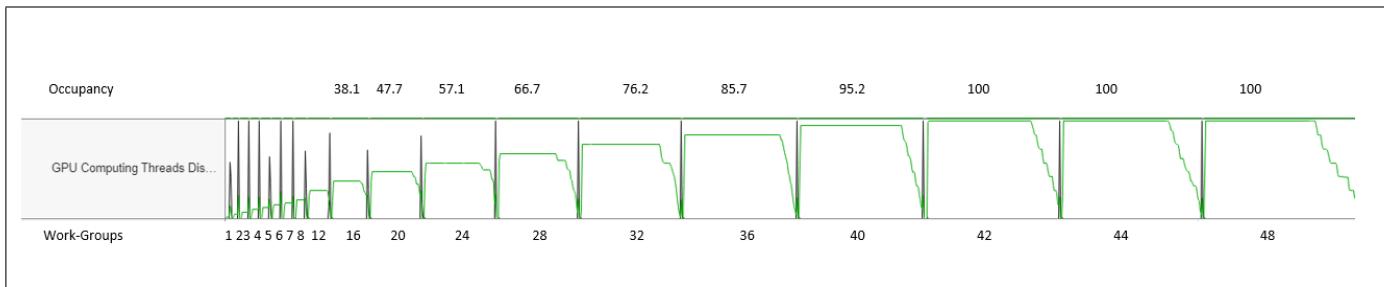
Program Occupancy	Work-groups	Work-items	Work-group Size	SIMD	Threads Work-group	Threads	Occupancy
VectorAdd1	53760	13.7M	512	32	16	430K	100%
VectorAdd2<1>	1	512	512	32	16	16	16/672 = 2.4%
VectorAdd2<2>	2	1024	512	32	16	32	32/672 = 4.8%
VectorAdd2<3>	3	1536	512	32	16	48	48/672 = 7.1%
VectorAdd2<4>	4	2048	512	32	16	64	64/672 = 9.5%
VectorAdd2<5>	5	2560	512	32	16	80	80/672 = 11.9%
VectorAdd2<6>	6	3072	512	32	16	96	96/672 = 14.3%
VectorAdd2<7>	7	3584	512	32	16	112	112/672 = 16.7%
VectorAdd2<8>	8	4096	512	32	16	128	128/672 = 19%
VectorAdd2<12>	12	6144	512	32	16	192	192/672 = 28.6%
VectorAdd2<16>	16	8192	512	32	16	256	256/672 = 38.1%
VectorAdd2<20>	20	10240	512	32	16	320	320/672 = 47.7%
VectorAdd2<24>	24	12288	512	32	16	384	384/672 = 57.1%
VectorAdd2<28>	28	14336	512	32	16	448	448/672 = 66.7%
VectorAdd2<32>	32	16384	512	32	16	512	512/672 = 76.2%
VectorAdd2<36>	36	18432	512	32	16	576	576/672 = 85.7%
VectorAdd2<40>	40	20480	512	32	16	640	640/672 = 95.2%
VectorAdd2<42>	42	21504	512	32	16	672	672/672 = 100%
VectorAdd2<44>	44	22528	512	32	16	704	100% then 4.7%
VectorAdd2<48>	48	24576	512	32	16	768	100% then 14.3%

The following VTune™ analyzer chart for VectorAdd2 with various work-group sizes confirms the accuracy of our estimate. The numbers in the grid view vary slightly from the estimate because the grid view gives an average across the entire execution.

**Fig. 5:** Occupancy for VectorAdd2 as shown by VTune

The following timeline view gives the occupancy over a period of time and it can be seen that the occupancy

metric is accurate for large part of the kernel execution and tapers off towards the end due to the varying times at which each of the threads finish their execution.



**Fig. 6:** VectorAdd2 timeline view

The kernel VectorAdd3 shown below is similar to the kernels above with two important differences.

1. It can be instantiated with the number of work-groups, work-group size and sub-group size as template parameters. This allows us to do experiments to investigate the impact of number of sub-groups and work-groups on thread occupancy.
2. The amount of work done inside the kernel is dramatically increased to ensure that these kernels are resident in the execution units doing work for a substantial amount of time.

**Listing 6:** /examples/exec-model/vaddsync.cpp

```

1 template <int groups, int wg_size, int sg_size>
2 int VectorAdd3(sycl::queue &q, const IntArray &a, const IntArray &b,
3                 IntArray &sum, int iter) {
4     sycl::range num_items{a.size()};
5
6     sycl::buffer a_buf(a);
7     sycl::buffer b_buf(b);
8     sycl::buffer sum_buf(sum.data(), num_items);
9     size_t num_groups = groups;
10    auto start = std::chrono::steady_clock::now();
11    q.submit([&](auto &h) {
12        // Input accessors
13        sycl::accessor a_acc(a_buf, h, sycl::read_only);
14        sycl::accessor b_acc(b_buf, h, sycl::read_only);
15        // Output accessor
16        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
17
18        h.parallel_for(
19            sycl::nd_range<1>(num_groups * wg_size, wg_size), [=
20            ](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(sg_size)]] {
21                size_t grp_id = index.get_group()[0];
22                size_t loc_id = index.get_local_id();
23                size_t start = grp_id * mysize;
24                size_t end = start + mysize;
25                for (int i = 0; i < iter; i++) {
26                    for (size_t i = start + loc_id; i < end; i += wg_size) {
27                        sum_acc[i] = a_acc[i] + b_acc[i];
28                    }
29                }
30            }
31        );
32    });
33}
```

(continues on next page)

(continued from previous page)

```

28         }
29     });
30 };
31 q.wait();
32 auto end = std::chrono::steady_clock::now();
33 std::cout << "VectorAdd3<" << groups << "> completed on device - took "
34     << (end - start).count() << " u-secs\n";
35     return ((end - start).count());
36 } // end VectorAdd3

```

The kernel VectorAdd4 is similar to the kernel VectorAdd3 above except that it has a barrier synchronization at the beginning and end of the kernel execution. This barrier although functionally is not needed, will significantly impact the way in which threads are scheduled on the hardware.

**Listing 7:** /examples/exec-model/vaddsync.cpp

```

1 template <int groups, int wg_size, int sg_size>
2 int VectorAdd4(sycl::queue &q, const IntArray &a, const IntArray &b,
3                 IntArray &sum, int iter) {
4     sycl::range num_items{a.size()};
5
6     sycl::buffer a_buf(a);
7     sycl::buffer b_buf(b);
8     sycl::buffer sum_buf(sum.data(), num_items);
9     size_t num_groups = groups;
10    auto start = std::chrono::steady_clock::now();
11    q.submit([&](auto &h) {
12        // Input accessors
13        sycl::accessor a_acc(a_buf, h, sycl::read_only);
14        sycl::accessor b_acc(b_buf, h, sycl::read_only);
15        // Output accessor
16        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
17
18        h.parallel_for(
19            sycl::nd_range<1>(num_groups * wg_size, wg_size), [=
20                (sycl::nd_item<1> index) [[intel::reqd_sub_group_size(sg_size)]] {
21                    index.barrier(sycl::access::fence_space::local_space);
22                    size_t grp_id = index.get_group()[0];
23                    size_t loc_id = index.get_local_id();
24                    size_t start = grp_id * mysize;
25                    size_t end = start + mysize;
26                    for (int i = 0; i < iter; i++) {
27                        for (size_t i = start + loc_id; i < end; i += wg_size) {
28                            sum_acc[i] = a_acc[i] + b_acc[i];
29                        }
30                    }
31                });
32    });
33    q.wait();
34    auto end = std::chrono::steady_clock::now();
35    std::cout << "VectorAdd4<" << groups << "> completed on device - took "

```

(continues on next page)

(continued from previous page)

```

36     << (end - start).count() << " u-secs\n";
37     return ((end - start).count());
38 } // end VectorAdd4

```

To illustrate the manner in which threads are scheduled, the above two kernels are called with 8 work-groups, sub-group size of 8 and work-group size of 320 as shown below. Based on the choice of work-group size and sub-group size, there will be 40 threads per work-group which need to be scheduled by the hardware.

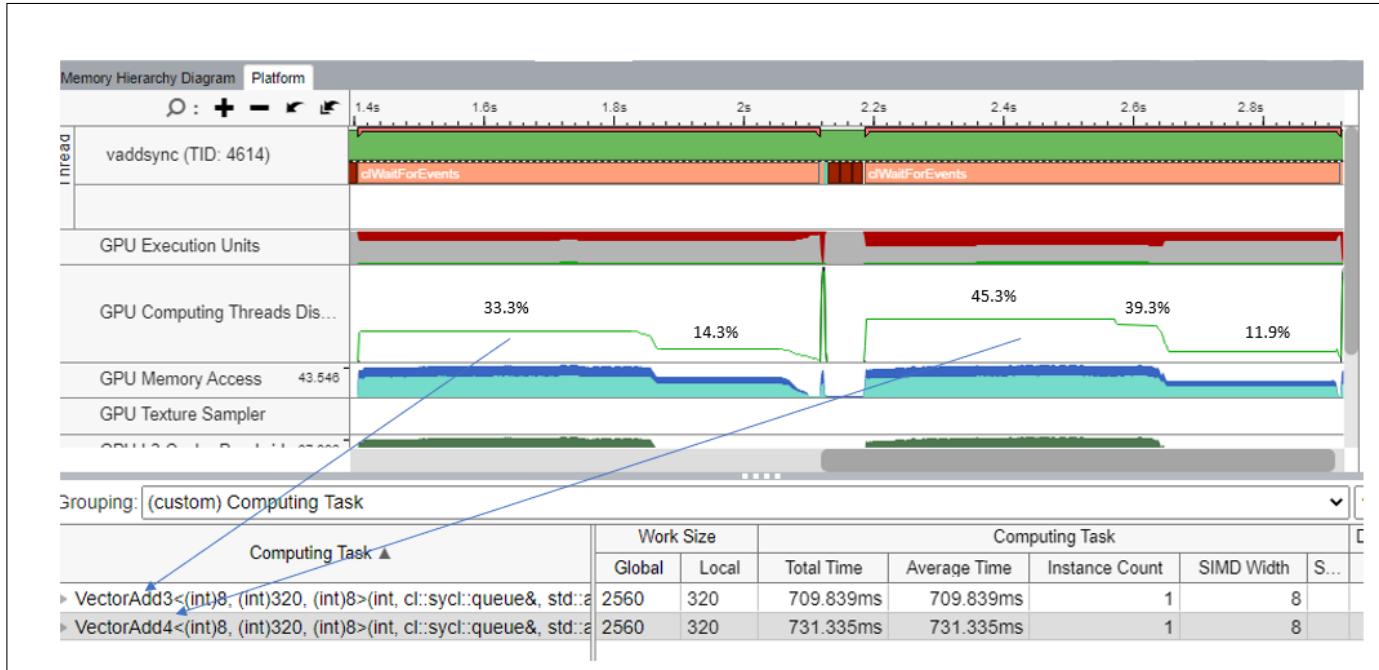
**Listing 8:** /examples/exec-model/vaddsync.cpp

```

1 Initialize(sum);
2 VectorAdd3<8, 320, 8>(q, a, b, sum, 10000);
3 Initialize(sum);
4 VectorAdd4<8, 320, 8>(q, a, b, sum, 10000);

```

The chart from Intel® VTune™ below shows that the measured GPU occupancy for VectorAdd3 and VectorAdd4 kernels.



**Fig. 7:** GPU occupancy VectorAdd3, VectorAdd4 kernels

For VectorAdd3 kernel it can be seen that there are two phases for occupancies. One is 33.3% (224 threads occupancy) and the other is 14.3% (96 threads occupancy) on a TGL machine which has a total of 672 threads. Since we know that there are a total of eight work-groups with each work-group having 40 threads, we can conclude that there are two sub-slices (each of which have 112 threads) into which the threads of six work-groups are scheduled. This means that 40 threads each of four work-groups are scheduled and 32 threads each from two other work-groups are scheduled in the first phase. Then later in the second phase we have 40 threads from remaining two work-groups are scheduled for execution.

For VectorAdd4 kernel it can be seen that there are three phases of occupancies which are 45.3% (304 threads), 39.3% (264 threads) and 11.9% (80 threads). In the first phase, all eight work-groups are scheduled

together on 3 sub-slices with two sub-slices getting 112 threads each (80 from two work-groups and 32 from one work-group) and one sub-slice getting 80 threads (from two work-groups). In the second phase, one work-group completed execution which gives us occupancy of (304-40=264). In the last phase, the remaining eight threads of two work-groups are scheduled and they complete the execution.

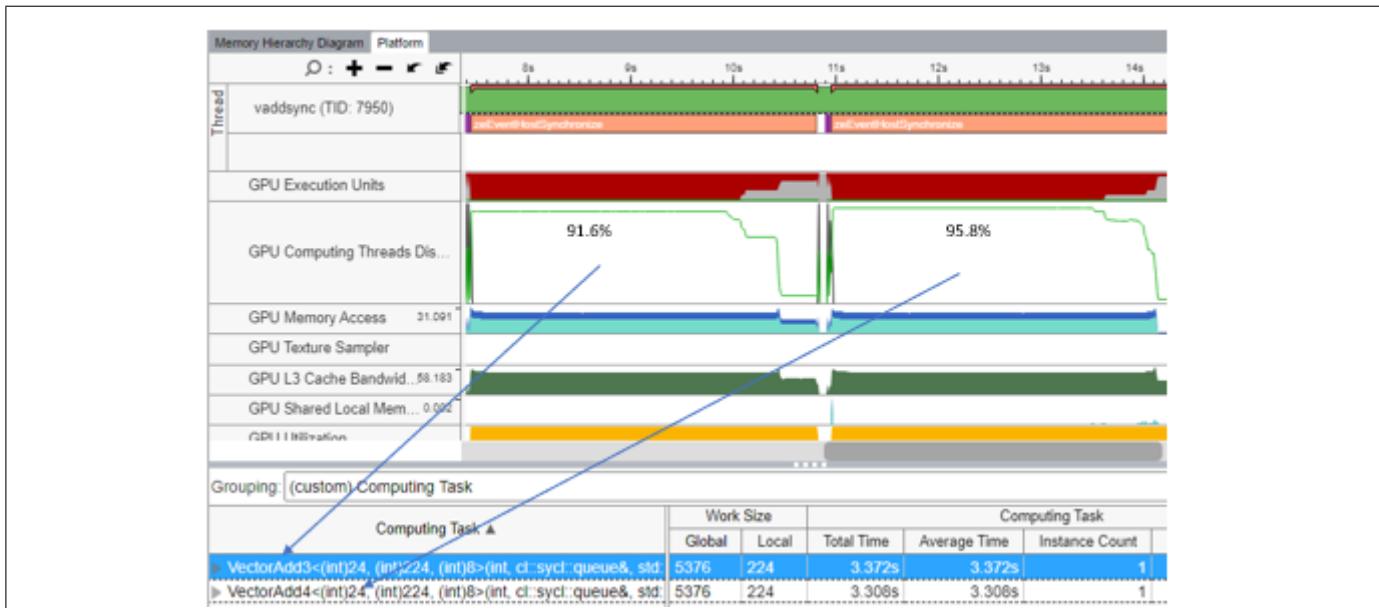
The same kernels as above when run with different work-group size which is chosen to be a multiple of the number of threads in a sub-slice and lot more work-groups gets good utilization of the hardware achieving close to 100% occupancy as shown below.

**Listing 9:** /examples/exec-model/vaddsync.cpp

```

1 Initialize(sum);
2 VectorAdd3<24, 224, 8>(q, a, b, sum, 10000);
3 Initialize(sum);
4 VectorAdd4<24, 224, 8>(q, a, b, sum, 10000);
```

This kernel execution has a different thread occupancy since we have lot more threads and also the work-group size is a multiple of the number of threads in a sub-slice - this is shown below in the thread occupancy metric on Vtune time-line.



**Fig. 8:** Thread occupancy metric on Vtune

Note that the above schedule is a guess based on the different occupancy numbers since we do not yet have a way to examine the per slice based occupancy numbers.

One can run different experiments with the above kernels to gain better understanding of the way in which the GPU hardware schedules the software threads on the Execution Units for execution. One needs to be careful about the work-group size and sub-group size in addition to a large number of work-groups to ensure effective utilization of the GPU hardware.

## 6.0 Kernels

A kernel is the unit of computation in the oneAPI offload model. By submitting a kernel on an iteration space, you are requesting that the computation be applied to the specified data objects.

In this section we cover topics related to the coding, submission, and execution of kernels.

### 6.1 Reduction

Reduction is a common operation in parallel programming where an operator is applied to all elements of an array and a single result is produced. The reduction operator is associative and in some cases commutative. Some examples of reductions are summation, maximum, minimum, etc. A serial summation reduction is shown below:

**Listing 10:** /examples/reduction/reduction.cpp

```

1  for (int it = 0; it < iter; it++) {
2      sum = 0;
3      for (size_t i = 0; i < data_size; ++i) {
4          sum += data[i];
5      }
6  }
```

The time complexity of reduction is linear with the number of elements. There are several ways this can be parallelized, and care must be taken to ensure that the amount of communication/synchronization is minimized between different processing elements. A naive way to parallelize this reduction is to use a global variable and let the threads to update this variable using an atomic operation:

**Listing 11:** /examples/reduction/reduction.cpp

```

1  q.submit([&](auto &h) {
2      sycl::accessor buf_acc(buf, h, sycl::read_only);
3      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
4
5      h.parallel_for(data_size, [=](auto index) {
6          size_t glob_id = index[0];
7          auto v = sycl::ext::oneapi::atomic_ref<
8              int, sycl::ext::oneapi::memory_order::relaxed,
9              sycl::ext::oneapi::memory_scope::device,
10             sycl::access::address_space::global_space>(sum_acc[0]);
11          v.fetch_add(buf_acc[glob_id]);
12      });
13  });
```

This kernel will perform poorly because the threads are atomically updating a single memory location and getting significant contention. A better approach is to split the array into small chunks, let each thread compute a local sum for each chunk, and then do a sequential/tree reduction of the local sums. The number of chunks will depend on the number of processing elements present in the platform. This can be queried using the `get_info<info::device::max_compute_units>()` function on the device object:

**Listing 12:** /examples/reduction/reduction.cpp

```

1  q.submit([&](auto &h) {
2      sycl::accessor buf_acc(buf, h, sycl::read_only);
3      sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
4      h.parallel_for(num_processing_elements, [=](auto index) {
5          size_t glob_id = index[0];
6          size_t start = glob_id * BATCH;
7          size_t end = (glob_id + 1) * BATCH;
8          if (end > N)
9              end = N;
10         int sum = 0;
11         for (size_t i = start; i < end; i++)
12             sum += buf_acc[i];
13         accum_acc[glob_id] = sum;
14     });
15 });

```

This kernel will perform better than the kernel that atomically updates a shared memory location. However, it is still inefficient because the compiler is not able to vectorize the loop. One way to get the compiler to produce vector code is to modify the loop as shown below:

**Listing 13:** /examples/reduction/reduction.cpp

```

1  q.submit([&](auto &h) {
2      sycl::accessor buf_acc(buf, h, sycl::read_only);
3      sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
4      h.parallel_for(num_work_items, [=](auto index) {
5          size_t glob_id = index[0];
6          int sum = 0;
7          for (size_t i = glob_id; i < data_size; i += num_work_items)
8              sum += buf_acc[i];
9          accum_acc[glob_id] = sum;
10     });
11 });

```

The compiler can vectorize this code so the performance is better.

In the case of GPUs, a number of thread contexts are available per physical processor (referred to as EU) on the machine. So the above code where the number of threads is equal to the number of EUs does not utilize all the thread contexts. Even in the case of CPUs which have two hyperthreads per core, the code will not use all the thread contexts. In general, it is better to divide the work into enough work-groups to get full occupancy of all thread contexts. This allows the code to better tolerate long latency instructions. The following table shows the number of thread contexts available per processing element in different devices:

**Table 7:** Number of thread contexts available by device

	<b>EUs</b>	<b>Threads/EU</b>	<b>Total Threads</b>
KBL	24	7	$24 \times 7 = 168$
TGL	96	7	$96 \times 7 = 672$

The code below shows the kernel with enough threads to fully utilize available resources. Notice that there is no good way to query the number of available thread contexts from the device. So, depending on the device, we scale the number of work-items we create for splitting the work among them.

**Listing 14:** /examples/reduction/reduction.cpp

```

1 q.submit([&](auto &h) {
2     sycl::accessor buf_acc(buf, h, sycl::read_only);
3     sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
4     h.parallel_for(num_work_items, [=](auto index) {
5         size_t glob_id = index[0];
6         int sum = 0;
7         for (size_t i = glob_id; i < data_size; i += num_work_items)
8             sum += buf_acc[i];
9         accum_acc[glob_id] = sum;
10    });
11 });

```

One popular way of doing reduction operation on GPUs is to create a number of work-groups and do a tree reduction in each work-group. In the kernel shown below, each work-item in the work-group participates in a reduction network to eventually sum up all the elements in that work-group. All the intermediate results from the work-groups are then summed up by doing a serial reduction (if this intermediate set of results is large enough then we can do few more round(s) of tree reductions). This tree reduction algorithm takes advantage of the very fast synchronization operations among the work-items in a work-group. The performance of this kernel is highly dependent on the efficiency of the kernel launches since a large number of kernels are launched. Also, the kernel as written below is not very efficient since the number of threads doing actual work reduces exponentially every time through the loop.

**Listing 15:** /examples/reduction/reduction.cpp

```

1 q.submit([&](auto &h) {
2     sycl::accessor buf_acc(buf, h, sycl::read_only);
3     sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
4     sycl::accessor<int, 1, sycl::access::mode::read_write,
5                  sycl::access::target::local>
6     scratch(work_group_size, h);
7
8     h.parallel_for(sycl::nd_range<1>(num_work_items, work_group_size),
9                   [=](sycl::nd_item<1> item) {
10        size_t global_id = item.get_global_id(0);
11        int local_id = item.get_local_id(0);
12        int group_id = item.get_group(0);
13        int sum = 0;
14
15        if (global_id < data_size)
16            scratch[local_id] = buf_acc[global_id];
17        else
18            scratch[local_id] = 0;
19
20        // Do a tree reduction on items in work-group
21        for (int i = work_group_size / 2; i > 0; i >>= 1) {
22            item.barrier(sycl::access::fence_space::local_space);

```

(continues on next page)

(continued from previous page)

```

23         if (local_id < i)
24             scratch[local_id] += scratch[local_id + i];
25     }
26
27     if (local_id == 0)
28         accum_acc[group_id] = scratch[0];
29   });
30 };

```

The single stage reduction is not very efficient since it will leave a lot work for the host. Adding one more stage will reduce the work on the host and improves performance quite a bit. It can be seen that in the kernel below the intermediate result computed in stage1 is used as input into stage2. This can be generalized to form a multi-stage reduction until the result is small enough so that it can be performed on the host.

**Listing 16:** /examples/reduction/reduction.cpp

```

1 q.submit([&](auto &h) {
2     sycl::accessor buf_acc(buf, h, sycl::read_only);
3     sycl::accessor accum_acc(accum1_buf, h, sycl::write_only, sycl::no_init);
4     sycl::accessor<int, 1, sycl::access::mode::read_write,
5                  sycl::access::target::local>
6     scratch(work_group_size, h);
7
8     h.parallel_for(sycl::nd_range<1>(num_work_items1, work_group_size),
9                    [=](sycl::nd_item<1> item) {
10            size_t global_id = item.get_global_id(0);
11            int local_id = item.get_local_id(0);
12            int group_id = item.get_group(0);
13            int sum = 0;
14
15            if (global_id < data_size)
16                scratch[local_id] = buf_acc[global_id];
17            else
18                scratch[local_id] = 0;
19
20            // Do a tree reduction on items in work-group
21            for (int i = work_group_size / 2; i > 0; i >= 1) {
22                item.barrier(sycl::access::fence_space::local_space);
23                if (local_id < i)
24                    scratch[local_id] += scratch[local_id + i];
25            }
26
27            if (local_id == 0)
28                accum_acc[group_id] = scratch[0];
29        });
30    });
31    q.submit([&](auto &h) {
32        sycl::accessor buf_acc(accum1_buf, h, sycl::read_only);
33        sycl::accessor accum_acc(accum2_buf, h, sycl::write_only, sycl::no_init);
34        sycl::accessor<int, 1, sycl::access::mode::read_write,
35                      sycl::access::target::local>

```

(continues on next page)

(continued from previous page)

```

36     scratch(work_group_size, h);
37
38     h.parallel_for(sycl::nd_range<1>(num_work_items2, work_group_size),
39                     [=](sycl::nd_item<1> item) {
40                         size_t global_id = item.get_global_id(0);
41                         int local_id = item.get_local_id(0);
42                         int group_id = item.get_group(0);
43                         int sum = 0;
44
45                         if (global_id < num_work_items2)
46                             scratch[local_id] = buf_acc[global_id];
47                         else
48                             scratch[local_id] = 0;
49
50                         // Do a tree reduction on items in work-group
51                         for (int i = work_group_size / 2; i > 0; i >= 1) {
52                             item.barrier(sycl::access::fence_space::local_space);
53                             if (local_id < i)
54                                 scratch[local_id] += scratch[local_id + i];
55                         }
56
57                         if (local_id == 0)
58                             accum_acc[group_id] = scratch[0];
59                     });
60     });

```

DPC++ also supports built-in reduction operations and in cases where it is suitable one should use it since its implementation is fine tuned to the underlying architecture. The following kernel shows how to use the built-in reduction operator in the compiler.

**Listing 17:** /examples/reduction/reduction.cpp

```

1 q.submit([&](auto &h) {
2     sycl::accessor buf_acc(buf, h, sycl::read_only);
3     sycl::accessor sum_acc(sum_buf, h, sycl::read_write);
4     auto sumr =
5         sycl::ext::oneapi::reduction(sum_acc, sycl::ext::oneapi::plus<>());
6     h.parallel_for(sycl::nd_range<1>{data_size, 256}, sumr,
7                     [=](sycl::nd_item<1> item, auto &sumr_arg) {
8                         int glob_id = item.get_global_id(0);
9                         sumr_arg += buf_acc[glob_id];
10                    });
11    });

```

A further optimization is to block the accesses to the input vector and use the shared local memory to store the intermediate results. This kernel is shown below. In this kernel every work-item operates on a certain number of vector elements and then one thread in the work-group reduces all these elements to one result by linearly going through the shared memory containing the intermediate results.

**Listing 18:** /examples/reduction/reduction.cpp

```

1  q.submit([&](auto &h) {
2      sycl::accessor buf_acc(buf, h, sycl::read_only);
3      sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
4      sycl::accessor<int, 1, sycl::access::mode::read_write,
5                      sycl::access::target::local>
6          scratch(work_group_size, h);
7      h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
8                     [=](sycl::nd_item<1> item) {
9                         size_t glob_id = item.get_global_id(0);
10                        size_t group_id = item.get_group(0);
11                        size_t loc_id = item.get_local_id(0);
12                        int offset = ((glob_id >> log2workitems_per_block)
13                                      << log2elements_per_block) +
14                                      (glob_id & mask);
15                        int sum = 0;
16                        for (size_t i = 0; i < elements_per_work_item; i++)
17                            sum +=
18                                buf_acc[(i << log2workitems_per_block) + offset];
19                        scratch[loc_id] = sum;
20                        // Serial Reduction
21                        item.barrier(sycl::access::fence_space::local_space);
22                        if (loc_id == 0) {
23                            int sum = 0;
24                            for (int i = 0; i < work_group_size; i++)
25                                sum += scratch[i];
26                            accum_acc[group_id] = sum;
27                        }
28                    });
29    });

```

The kernel below is similar to the one above except that tree reduction is used to reduce the intermediate results from all the work-items in a work-group. In most cases this does not seem to make a big difference in performance.

**Listing 19:** /examples/reduction/reduction.cpp

```

1  q.submit([&](auto &h) {
2      sycl::accessor buf_acc(buf, h, sycl::read_only);
3      sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
4      sycl::accessor<int, 1, sycl::access::mode::read_write,
5                      sycl::access::target::local>
6          scratch(work_group_size, h);
7      h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
8                     [=](sycl::nd_item<1> item) {
9                         size_t glob_id = item.get_global_id(0);
10                        size_t group_id = item.get_group(0);
11                        size_t loc_id = item.get_local_id(0);
12                        int offset = ((glob_id >> log2workitems_per_block)
13                                      << log2elements_per_block) +
14                                      (glob_id & mask);

```

(continues on next page)

(continued from previous page)

```

15     int sum = 0;
16     for (size_t i = 0; i < elements_per_work_item; i++)
17         sum +=
18             buf_acc[(i << log2workitems_per_block) + offset];
19     scratch[loc_id] = sum;
20     // tree reduction
21     item.barrier(sycl::access::fence_space::local_space);
22     for (int i = work_group_size / 2; i > 0; i >= 1) {
23         item.barrier(sycl::access::fence_space::local_space);
24         if (loc_id < i)
25             scratch[loc_id] += scratch[loc_id + i];
26     }
27     if (loc_id == 0)
28         accum_acc[group_id] = scratch[0];
29     });
30 });

```

The kernel below uses the blocking technique and then the compiler reduction operator to do final reduction. This gives good performance on most of the platforms that it was tested.

**Listing 20:** /examples/reduction/reduction.cpp

```

1 q.submit([&](auto &h) {
2     sycl::accessor buf_acc(buf, h, sycl::read_only);
3     sycl::accessor sum_acc(sum_buf, h, sycl::read_write, sycl::no_init);
4     auto sumr =
5         sycl::ext::oneapi::reduction(sum_acc, sycl::ext::oneapi::plus<>());
6     h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size}, sumr,
7                     [=](sycl::nd_item<1> item, auto &sumr_arg) {
8                         size_t glob_id = item.get_global_id(0);
9                         size_t group_id = item.get_group(0);
10                        size_t loc_id = item.get_local_id(0);
11                        int offset = ((glob_id >> log2workitems_per_block)
12                                     << log2elements_per_block) +
13                                     (glob_id & mask);
14                        int sum = 0;
15                        for (size_t i = 0; i < elements_per_work_item; i++)
16                            sum +=
17                                buf_acc[(i << log2workitems_per_block) + offset];
18                        sumr_arg += sum;
19                    });
20 });

```

This next kernel uses a completely different technique for accessing the memory. It uses sub-group loads to generate the intermediate result in a vector form. This intermediate result is then brought back to the host and the final reduction is performed there. In some cases it may be better to create another kernel to reduce this result in a single work-group which allows one to do perform tree reduction through efficient barriers.

**Listing 21:** /examples/reduction/reduction.cpp

```

1   q.submit([&](auto &h) {
2     const sycl::accessor buf_acc(buf, h);
3     sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
4     sycl::accessor<sycl::vec<int, 8>, 1, sycl::access::mode::read_write,
5                 sycl::access::target::local>
6       scratch(work_group_size, h);
7     h.parallel_for(
8       sycl::nd_range<1>{num_work_items, work_group_size}, [=
9     ](sycl::nd_item<1> item) [[intel::reqd_sub_group_size(16)]] {
10       size_t glob_id = item.get_global_id(0);
11       size_t group_id = item.get_group(0);
12       size_t loc_id = item.get_local_id(0);
13       sycl::ext::oneapi::sub_group sg = item.get_sub_group();
14       sycl::vec<int, 8> sum{0, 0, 0, 0, 0, 0, 0, 0};
15       using global_ptr =
16         sycl::multi_ptr<int, sycl::access::address_space::global_space>;
17       int base = (group_id * work_group_size +
18                   sg.get_group_id()[0] * sg.get_local_range()[0]) *
19                   elements_per_work_item;
20       for (size_t i = 0; i < elements_per_work_item / 8; i++)
21         sum += sg.load<8>(global_ptr(&buf_acc[base + i * 128]));
22       scratch[loc_id] = sum;
23       for (int i = work_group_size / 2; i > 0; i >>= 1) {
24         item.barrier(sycl::access::fence_space::local_space);
25         if (loc_id < i)
26           scratch[loc_id] += scratch[loc_id + i];
27       }
28       if (loc_id == 0)
29         accum_acc[group_id] = scratch[0];
30     });
31   });

```

Different implementations of reduction operation are provided and discussed here which can have different performance characteristics which depends on the architecture of the accelerator. Another important thing to notice here is that time it takes to being the result of reduction to the host over the PCIE interface (for a discrete GPU) is almost same as actually doing the entire reduction on the device. This shows that one should avoid data transfers between host and device as much as possible or overlap the kernel execution with data transfers.

## 6.2 Sub-groups

The index space of an ND-Range kernel is divided into work-groups, sub-groups, and work-items. A work-item is the basic unit. A collection of work-items form a sub-group, and a collection of sub-groups form a work-group. The mapping of work-items and work-groups to hardware execution units (EU) is implementation-dependent. All the work-groups run concurrently but may be scheduled to run at different points in time depending on availability of resources. Work-group execution may or may not be preempted depending on the capabilities of underlying hardware. Work-items in the same work-group are guaranteed to run concurrently. Work-items in the same sub-group may have additional scheduling guarantees and have access to additional functionality.

A sub-group is a collection of contiguous work-items in the global index space that execute in the same EU thread. When the device compiler compiles the kernel, multiple work-items are packed into a sub-group by vectorization so the generated SIMD instruction stream can perform tasks of multiple work-items simultaneously. Properly partitioning work-items into sub-groups can make a big performance difference.

Let's start with a simple example illustrating sub-groups:

**Listing 22:** /examples/sub-group/sub-group-0.cpp

```

1  q.submit([&](auto &h) {
2      sycl::stream out(65536, 256, h);
3      h.parallel_for(sycl::nd_range(sycl::range{32}, sycl::range{32}),
4                      [=](sycl::nd_item<1> it) {
5              int groupId = it.get_group(0);
6              int globalId = it.get_global_linear_id();
7              sycl::ext::oneapi::sub_group sg = it.get_sub_group();
8              int sgSize = sg.get_local_range()[0];
9              int sgGroupId = sg.get_group_id()[0];
10             int sgId = sg.get_local_id()[0];
11
12             out << "globalId = " << sycl::setw(2) << globalId
13             << " groupId = " << groupId
14             << " sgGroupId = " << sgGroupId << " sgId = " << sgId
15             << " sgSize = " << sycl::setw(2) << sgSize
16             << sycl::endl;
17         });
18     });

```

The output of this example may look like this:

```

Device: Intel(R) Gen12HP
globalId = 0 groupId = 0 sgGroupId = 0 sgId = 0 sgSize = 16
globalId = 1 groupId = 0 sgGroupId = 0 sgId = 1 sgSize = 16
globalId = 2 groupId = 0 sgGroupId = 0 sgId = 2 sgSize = 16
globalId = 3 groupId = 0 sgGroupId = 0 sgId = 3 sgSize = 16
globalId = 4 groupId = 0 sgGroupId = 0 sgId = 4 sgSize = 16
globalId = 5 groupId = 0 sgGroupId = 0 sgId = 5 sgSize = 16
globalId = 6 groupId = 0 sgGroupId = 0 sgId = 6 sgSize = 16
globalId = 7 groupId = 0 sgGroupId = 0 sgId = 7 sgSize = 16
globalId = 16 groupId = 0 sgGroupId = 1 sgId = 0 sgSize = 16
globalId = 17 groupId = 0 sgGroupId = 1 sgId = 1 sgSize = 16
globalId = 18 groupId = 0 sgGroupId = 1 sgId = 2 sgSize = 16
globalId = 19 groupId = 0 sgGroupId = 1 sgId = 3 sgSize = 16
globalId = 20 groupId = 0 sgGroupId = 1 sgId = 4 sgSize = 16
globalId = 21 groupId = 0 sgGroupId = 1 sgId = 5 sgSize = 16
globalId = 22 groupId = 0 sgGroupId = 1 sgId = 6 sgSize = 16
globalId = 23 groupId = 0 sgGroupId = 1 sgId = 7 sgSize = 16
globalId = 8 groupId = 0 sgGroupId = 0 sgId = 8 sgSize = 16
globalId = 9 groupId = 0 sgGroupId = 0 sgId = 9 sgSize = 16
globalId = 10 groupId = 0 sgGroupId = 0 sgId = 10 sgSize = 16
globalId = 11 groupId = 0 sgGroupId = 0 sgId = 11 sgSize = 16
globalId = 12 groupId = 0 sgGroupId = 0 sgId = 12 sgSize = 16
globalId = 13 groupId = 0 sgGroupId = 0 sgId = 13 sgSize = 16

```

(continues on next page)

(continued from previous page)

```
globalId = 14 groupId = 0 sgGroupId = 0 sgId = 14 sgSize = 16
globalId = 15 groupId = 0 sgGroupId = 0 sgId = 15 sgSize = 16
globalId = 24 groupId = 0 sgGroupId = 1 sgId = 8 sgSize = 16
globalId = 25 groupId = 0 sgGroupId = 1 sgId = 9 sgSize = 16
globalId = 26 groupId = 0 sgGroupId = 1 sgId = 10 sgSize = 16
globalId = 27 groupId = 0 sgGroupId = 1 sgId = 11 sgSize = 16
globalId = 28 groupId = 0 sgGroupId = 1 sgId = 12 sgSize = 16
globalId = 29 groupId = 0 sgGroupId = 1 sgId = 13 sgSize = 16
globalId = 30 groupId = 0 sgGroupId = 1 sgId = 14 sgSize = 16
globalId = 31 groupId = 0 sgGroupId = 1 sgId = 15 sgSize = 16
```

Each sub-group in this example has 16 work-items or the sub-group size is 16. This means each thread simultaneously executes 16 work-items and 32 work-items are executed by two EU threads.

By default, the compiler selects a sub-group size using device-specific information and a few heuristics. The user can override the compiler's selection using a kernel attribute `intel::reqd_sub_group_size` to specify the maximum sub-group size. Sometimes, not always, explicitly requesting a sub-group size may help performance.

**Listing 23:** /examples/sub-group/sub-group-1.cpp

```
1 q.submit([&](auto &h) {
2     sycl::stream out(65536, 256, h);
3     h.parallel_for(
4         sycl::nd_range(sycl::range{32}, sycl::range{32}), [=
5         ](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(32)]] {
6             int groupId = it.get_group(0);
7             int globalId = it.get_global_linear_id();
8             sycl::ext::oneapi::sub_group sg = it.get_sub_group();
9             int sgSize = sg.get_local_range()[0];
10            int sgGroupId = sg.get_group_id()[0];
11            int sgId = sg.get_local_id()[0];
12
13            out << "globalId = " << sycl::setw(2) << globalId
14                << " groupId = " << groupId << " sgGroupId = " << sgGroupId
15                << " sgId = " << sgId << " sgSize = " << sycl::setw(2) << sgSize
16                << sycl::endl;
17        });
18    });
});
```

The output will be:

```
Device: Intel(R) Gen12HP
globalId = 0 groupId = 0 sgGroupId = 0 sgId = 0 sgSize = 32
globalId = 1 groupId = 0 sgGroupId = 0 sgId = 1 sgSize = 32
globalId = 2 groupId = 0 sgGroupId = 0 sgId = 2 sgSize = 32
globalId = 3 groupId = 0 sgGroupId = 0 sgId = 3 sgSize = 32
globalId = 4 groupId = 0 sgGroupId = 0 sgId = 4 sgSize = 32
globalId = 5 groupId = 0 sgGroupId = 0 sgId = 5 sgSize = 32
globalId = 6 groupId = 0 sgGroupId = 0 sgId = 6 sgSize = 32
globalId = 7 groupId = 0 sgGroupId = 0 sgId = 7 sgSize = 32
globalId = 8 groupId = 0 sgGroupId = 0 sgId = 8 sgSize = 32
```

(continues on next page)

(continued from previous page)

```
globalId = 9 groupId = 0 sgGroupId = 0 sgId = 9 sgSize = 32
globalId = 10 groupId = 0 sgGroupId = 0 sgId = 10 sgSize = 32
globalId = 11 groupId = 0 sgGroupId = 0 sgId = 11 sgSize = 32
globalId = 12 groupId = 0 sgGroupId = 0 sgId = 12 sgSize = 32
globalId = 13 groupId = 0 sgGroupId = 0 sgId = 13 sgSize = 32
globalId = 14 groupId = 0 sgGroupId = 0 sgId = 14 sgSize = 32
globalId = 15 groupId = 0 sgGroupId = 0 sgId = 15 sgSize = 32
globalId = 16 groupId = 0 sgGroupId = 0 sgId = 16 sgSize = 32
globalId = 17 groupId = 0 sgGroupId = 0 sgId = 17 sgSize = 32
globalId = 18 groupId = 0 sgGroupId = 0 sgId = 18 sgSize = 32
globalId = 19 groupId = 0 sgGroupId = 0 sgId = 19 sgSize = 32
globalId = 20 groupId = 0 sgGroupId = 0 sgId = 20 sgSize = 32
globalId = 21 groupId = 0 sgGroupId = 0 sgId = 21 sgSize = 32
globalId = 22 groupId = 0 sgGroupId = 0 sgId = 22 sgSize = 32
globalId = 23 groupId = 0 sgGroupId = 0 sgId = 23 sgSize = 32
globalId = 24 groupId = 0 sgGroupId = 0 sgId = 24 sgSize = 32
globalId = 25 groupId = 0 sgGroupId = 0 sgId = 25 sgSize = 32
globalId = 26 groupId = 0 sgGroupId = 0 sgId = 26 sgSize = 32
globalId = 27 groupId = 0 sgGroupId = 0 sgId = 27 sgSize = 32
globalId = 28 groupId = 0 sgGroupId = 0 sgId = 28 sgSize = 32
globalId = 29 groupId = 0 sgGroupId = 0 sgId = 29 sgSize = 32
globalId = 30 groupId = 0 sgGroupId = 0 sgId = 30 sgSize = 32
globalId = 31 groupId = 0 sgGroupId = 0 sgId = 31 sgSize = 32
```

The valid sub-group sizes are device dependent. You can query the device to get this information:

**Listing 24:** /examples/sub-group/sub-group-sizes.cpp

```
1 std::cout << "Sub-group Sizes: ";
2 for (const auto &s :
3     q.get_device().get_info<sycl::info::device::sub_group_sizes>()) {
4     std::cout << s << " ";
5 }
6 std::cout << std::endl;
```

The valid sub-group sizes supported may be:

```
Device: Intel(R) Gen12HP
Subgroup Sizes: 8 16 32
```

Next, we will show how to use sub-groups to improve performance.

### 6.2.1 Vectorization and Memory Access

The Intel® graphics device has multiple EUs. Each EU is a multithreaded SIMD processor. The compiler generates SIMD instructions to pack multiple work-items in a sub-group to be executed simultaneously in an EU thread. The SIMD width (thus the sub-group size), selected by the compiler based on device characteristics and heuristics or requested explicitly by the kernel, can be 8, 16, or 32.

Given a SIMD width, maximizing SIMD lane utilization gives optimal instruction performance. If one or more lanes (or kernel instances or work items) diverge, the thread executes both branch paths before the paths merge later,

increasing dynamic instruction count. SIMD divergence negatively impacts performance. The compiler works hard to optimize divergence, but still it helps to avoid divergence in the source code, if possible.

How memory is accessed in work-items affects how memory is accessed in the sub-group or how the SIMD lanes are utilized. Accessing contiguous memory in a work-item is often not optimal. For example:

**Listing 25:** /examples/sub-group/sub-group-2.cpp

```

1 constexpr int N = 1024 * 1024;
2 int *data = sycl::malloc_shared<int>(N, q);
3
4 auto e = q.submit([&](auto &h) {
5     h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}),
6                     [=](sycl::nd_item<1> it) {
7                         int i = it.get_global_linear_id();
8                         i = i * 16;
9                         for (int j = i; j < (i + 16); j++) {
10                             data[j] = -1;
11                         }
12                     });
13 });
14 q.wait();

```

This simple kernel initializes an array of 1024 x 1024 integers. Each work-item initializes 16 contiguous integers. Assuming the sub-group size chosen by the compiler is 16, 256 integers are initialized in each sub-group or thread. However, the stores in 16 SIMD lanes are scattered.

Instead of initializing 16 contiguous integers in a work-item, initializing 16 contiguous integers in one SIMD instruction is more efficient.

**Listing 26:** /examples/sub-group/sub-group-3.cpp

```

1 constexpr int N = 1024 * 1024;
2 int *data = sycl::malloc_shared<int>(N, q);
3
4 auto e = q.submit([&](auto &h) {
5     h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}),
6                     [=](sycl::nd_item<1> it) {
7                         int i = it.get_global_linear_id();
8                         sycl::ext::oneapi::sub_group sg = it.get_sub_group();
9                         int sgSize = sg.get_local_range()[0];
10                        i = (i / sgSize) * sgSize * 16 + (i % sgSize);
11                        for (int j = 0; j < sgSize * 16; j += sgSize) {
12                            data[i + j] = -1;
13                        }
14                     });
15 });

```

We use memory writes in our examples, but the same technique is applicable to memory reads as well.

**Listing 27:** /examples/sub-group/sub-group-4.cpp

```

1  constexpr int N = 1024 * 1024;
2  int *data = sycl::malloc_shared<int>(N, q);
3  int *data2 = sycl::malloc_shared<int>(N, q);
4  memset(data2, 0xFF, sizeof(int) * N);
5
6  auto e = q.submit([&](auto &h) {
7      h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}),
8                      [=](sycl::nd_item<1> it) {
9                          int i = it.get_global_linear_id();
10                         i = i * 16;
11                         for (int j = i; j < (i + 16); j++) {
12                             data[j] = data2[j];
13                         }
14                     });
15     });

```

This kernel copies an array of 1024 x 1024 integers to another integer array of the same size. Each work-item copies 16 contiguous integers. However, the reads from **data2** are gathered and stores to **data** are scattered. It will be more efficient if we change the code to read and store contiguous integers in each sub-group instead of each work-item.

**Listing 28:** /examples/sub-group/sub-group-5.cpp

```

1  constexpr int N = 1024 * 1024;
2  int *data = sycl::malloc_shared<int>(N, q);
3  int *data2 = sycl::malloc_shared<int>(N, q);
4  memset(data2, 0xFF, sizeof(int) * N);
5
6  auto e = q.submit([&](auto &h) {
7      h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}),
8                      [=](sycl::nd_item<1> it) {
9                          int i = it.get_global_linear_id();
10                         sycl::ext::oneapi::sub_group sg = it.get_sub_group();
11                         int sgSize = sg.get_local_range()[0];
12                         i = (i / sgSize) * sgSize * 16 + (i % sgSize);
13                         for (int j = 0; j < sgSize * 16; j += sgSize) {
14                             data[i + j] = data2[i + j];
15                         }
16                     });
17     });

```

Intel® graphics have instructions optimized for memory block loads/stores. So if work-items in a sub-group access a contiguous block of memory, we can use the sub-group block access functions to take advantage of these block load/store instructions.

**Listing 29:** /examples/sub-group/sub-group-6.cpp

```

1  constexpr int N = 1024 * 1024;
2  int *data = sycl::malloc_shared<int>(N, q);

```

(continues on next page)

(continued from previous page)

```

3 int *data2 = sycl::malloc_shared<int>(N, q);
4 memset(data2, 0xFF, sizeof(int) * N);
5
6 auto e = q.submit([&](auto &h) {
7     h.parallel_for(
8         sycl::nd_range(sycl::range{N / 16}, sycl::range{32}), [=
9         ](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
10         sycl::ext::oneapi::sub_group sg = it.get_sub_group();
11         sycl::vec<int, 8> x;
12
13         using global_ptr =
14             sycl::multi_ptr<int, sycl::access::address_space::global_space>;
15         int base = (it.get_group(0) * 32 +
16                     sg.get_group_id()[0] * sg.get_local_range()[0]) *
17                     16;
18         x = sg.load<8>(global_ptr(&(data2[base + 0])));
19         sg.store<8>(global_ptr(&(data2[base + 0])), x);
20         x = sg.load<8>(global_ptr(&(data2[base + 128])));
21         sg.store<8>(global_ptr(&(data2[base + 128])), x);
22     });
23 });

```

You probably noticed that we explicitly requested the sub-group size 16. When you use sub-group functions, it is always good to override the compiler choice to make sure the sub-group size always matches what you expect. Please also note that, at the time of writing, block load/store does not work with sub-group size 32 on current Intel® hardware. So the group size explicitly requested must be 16 or smaller.

## 6.2.2 Data Sharing

Because the work-items in a sub-group execute in the same thread, it is more efficient to share data between work-items, even if the data is private to each work-item. Sharing data in a sub-group is more efficient than sharing data in a work-group using shared local memory, or SLM. One way to share data among work-items in a sub-group is to use shuffle functions.

**Listing 30:** /examples/sub-group transpose.cpp

```

1 constexpr size_t blockSize = 16;
2 sycl::buffer<unsigned int, 2> m(matrix.data(), sycl::range<2>(N, N));
3
4 auto e = q.submit([&](auto &h) {
5     sycl::accessor marr(m, h);
6     sycl::accessor<unsigned int, 2, sycl::access::mode::read_write,
7                     sycl::access::target::local>
8         barr1(sycl::range<2>(blockSize, blockSize), h);
9     sycl::accessor<unsigned int, 2, sycl::access::mode::read_write,
10                 sycl::access::target::local>
11         barr2(sycl::range<2>(blockSize, blockSize), h);
12
13     h.parallel_for(
14         sycl::nd_range<2>(sycl::range<2>(N / blockSize, N),

```

(continues on next page)

(continued from previous page)

```

15         sycl::range<2>(1, blockSize),
16 [=](sycl::nd_item<2> it) [[intel::reqd_sub_group_size(16)]] {
17     int gi = it.get_group(0);
18     int gj = it.get_group(1);
19
20     sycl::ext::oneapi::sub_group sg = it.get_sub_group();
21     int sgId = sg.get_local_id()[0];
22
23     unsigned int bcol[blockSize];
24     int ai = blockSize * gi;
25     int aj = blockSize * gj;
26
27     for (int k = 0; k < blockSize; k++) {
28         bcol[k] = sg.load(marr.get_pointer() + (ai + k) * N + aj);
29     }
30
31     unsigned int tcol[blockSize];
32     for (int n = 0; n < blockSize; n++) {
33         if (sgId == n) {
34             for (int k = 0; k < blockSize; k++) {
35                 tcol[k] = sg.shuffle(bcol[n], k);
36             }
37         }
38     }
39
40     for (int k = 0; k < blockSize; k++) {
41         sg.store(marr.get_pointer() + (ai + k) * N + aj, tcol[k]);
42     }
43 };
44 });

```

This kernel transposes a 16 x 16 matrix. It looks more complicated than the previous examples, but the idea is simple: a sub-group loads a 16 x 16 sub-matrix, then the sub-matrix is transposed using the sub-group shuffle functions. There is only one sub-matrix and the sub-matrix is the matrix so only one sub-group is needed. A bigger matrix, say 4096 x 4096, can be transposed using the same technique: each sub-group loads a sub-matrix, then the sub-matrices are transposed using the sub-group shuffle functions. We leave this to the reader as an exercise.

There are multiple variants of sub-group shuffle functions available in DPC++. Each variant is optimized for its specific purpose on the specific device. It is always a good idea to use these optimized functions (if they fit your needs) instead of creating your own.

### 6.2.3 Sub-group Size vs. Maximum Sub-group Size

So far in our examples, the work-group size is divisible by the sub-group size and both the work-group size and the sub-group size (either required by the user or automatically picked by the compiler) are powers of two). The sub-group size and maximum sub-group size are the same if the work-group size is divisible by the maximum sub-group size and both sizes are powers of two. But what happens if the work-group size is not divisible by the sub-group size? Consider the following example:

**Listing 31:** /examples/sub-group/sg-max-size.cpp

```

1 auto e = q.submit([&](auto &h) {
2     sycl::stream out(65536, 128, h);
3     h.parallel_for(
4         sycl::nd_range<1>(7, 7), [=
5            ](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(8)]] {
6                 int i = it.get_global_linear_id();
7                 sycl::ext::oneapi::sub_group sg = it.get_sub_group();
8                 int sgSize = sg.get_local_range()[0];
9                 int sgMaxSize = sg.get_max_local_range()[0];
10                int sId = sg.get_local_id()[0];
11                int j = data[i];
12                int k = data[i + sgSize];
13                out << "globalId = " << i << " sgMaxSize = " << sgMaxSize
14                  << " sgSize = " << sgSize << " sId = " << sId << " j = " << j
15                  << " k = " << k << sycl::endl;
16            });
17        });
18    q.wait();

```

The output of this example looks like this:

```

globalId = 0 sgMaxSize = 8 sgSize = 7 sId = 0 j = 0 k = 7
globalId = 1 sgMaxSize = 8 sgSize = 7 sId = 1 j = 1 k = 8
globalId = 2 sgMaxSize = 8 sgSize = 7 sId = 2 j = 2 k = 9
globalId = 3 sgMaxSize = 8 sgSize = 7 sId = 3 j = 3 k = 10
globalId = 4 sgMaxSize = 8 sgSize = 7 sId = 4 j = 4 k = 11
globalId = 5 sgMaxSize = 8 sgSize = 7 sId = 5 j = 5 k = 12
globalId = 6 sgMaxSize = 8 sgSize = 7 sId = 6 j = 6 k = 13

```

The sub-group size is seven, though the maximum sub-group size is still eight! The maximum sub-group size is actually the SIMD width so it does not change, but there are less than eight work-items in the sub-group, so the sub-group size is seven. So be careful when your work-group size is not divisible by the maximum sub-group size. The last sub-group with fewer work-items may need to be specially handled.

## 6.3 Avoiding Register Spills

### 6.3.1 Registers and Performance

It is well known that the register is the fastest storage in the memory hierarchy. Keeping data in registers as long as possible is critical to performance. On the other hand, register space is limited and much smaller than memory space. Current generation of Intel® GPUs, for example, has 128 general-purpose registers of 32-byte wide each by default for each EU thread. Though the compiler aims to assign as many variables to registers as possible, the limited number of registers can be allocated only to a small set of variables at any point during execution. A given register can hold different variables at different times because not all variables are needed at the same time and different set of variables are needed at different times. In the case that there are not enough registers to hold all the variables, register can spill, or some variables currently in the registers have to be moved to the memory to make room for other variables.

In DPC++, the compiler allocates registers to private variables in work items. Multiple work items in a sub-group are packed into one EU thread. By default, the compiler uses register pressure as one of the heuristics to choose SIMD width or sub-group size. High register pressures can result in smaller sub-group size, for example, 8 instead of 16, if a sub-group size is not explicitly requested, register spilling or certain variables not to be promoted to registers.

The hardware may not be fully utilized if sub-group size or SIMD width is not the maximum the hardware supports. Register spilling can cause significant performance degradation, especially when spills occur inside hot loops. When variables are not promoted to registers, accesses to these variables incur significant increase of memory traffic.

Though the compiler uses intelligent algorithms to avoid or minimize register spills if spilling is unavoidable, optimizations by developers can help the compiler to do a better job and often make big performance difference.

### 6.3.2 Optimization Techniques

A few techniques can be applied to reduce register pressures:

- keep the distance between loading value to a variable and using the variable as short as possible

Though the compiler schedules instructions and optimizes the distances, in some cases, moving the loading and using of the same variable closer or remove certain dependencies in the source can help the compiler to do a better job.

- avoid excessive loop unrolling

Loop unrolling exposes opportunities for instruction scheduling optimization by the compiler and thus can improve performance. However, temporary variables introduced by unrolling may increase pressure on register allocation and cause register spilling. It is always a good idea to compare the performance with and without loop unroll and different times of unrolls to decide if a loop should be unrolled or how many times to be unrolled.

- recompute cheap-to-compute values on-demand that otherwise would be held in registers for long time
- avoid big arrays or large structure if possible
- choose smaller data types if possible
- sharing registers in sub-group

- use shared local memory

The list here is not exhaustive.

The rest of this chapter will show how to apply these techniques, especially the last three ones in real examples.

### 6.3.3 Choosing Smaller Data Types

**Listing 32:** /examples/register/histogram32-long.cpp

```
1  constexpr int blockSize = 256;
2  constexpr int NUM_BINS = 32;
3
4  std::vector<unsigned long> hist(NUM_BINS, 0);
5  sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
6  sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);
7
8  auto e = q.submit([&](auto &h) {
9      sycl::accessor macc(mbuf, h, sycl::read_only);
10     auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
11     h.parallel_for(
12         sycl::nd_range(sycl::range{N / blockSize}, sycl::range{64}), [=
13         ](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
14             int group = it.get_group()[0];
15             int gSize = it.get_local_range()[0];
16             sycl::ext::oneapi::sub_group sg = it.get_sub_group();
17             int sgSize = sg.get_local_range()[0];
18             int sgGroup = sg.get_group_id()[0];
19
20             unsigned long
21                 histogram[NUM_BINS]; // histogram bins take too much storage to be
22                                         // promoted to registers
23             for (int k = 0; k < NUM_BINS; k++) {
24                 histogram[k] = 0;
25             }
26             for (int k = 0; k < blockSize; k++) {
27                 unsigned long x =
28                     sg.load(macc.get_pointer() + group * gSize * blockSize +
29                             sgGroup * sgSize * blockSize + sgSize * k);
30 #pragma unroll
31                 for (int i = 0; i < 8; i++) {
32                     unsigned int c = x & 0x1FU;
33                     histogram[c] += 1;
34                     x = x >> 8;
35                 }
36             }
37
38             for (int k = 0; k < NUM_BINS; k++) {
39                 hacc[k].fetch_add(histogram[k]);
40             }
41         });
42     });
});
```

This example calculates histograms with bin size of 32. Each work item has 32 private bins of unsigned long data type. Because of the large storage required, the private bins cannot fit in registers, resulting poor performance.

With blockSize 256, the maximum value of each private histogram bin will not exceed the maximum value of an unsigned integer. Instead of unsigned long type for private histogram bins, we can use unsigned integers to reduce register pressure so the private bins can fit in registers. This simple change makes significant performance difference.

**Listing 33:** /examples/register/histogram32-int.cpp

```

1  constexpr int blockSize = 256;
2  constexpr int NUM_BINS = 32;
3
4  std::vector<unsigned long> hist(NUM_BINS, 0);
5
6  sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
7  sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);
8
9  auto e = q.submit([&](auto &h) {
10    sycl::accessor macc(mbuf, h, sycl::read_only);
11    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
12    h.parallel_for(
13      sycl::nd_range(sycl::range{N / blockSize}, sycl::range{64}), [=
14      ](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
15        int group = it.get_group()[0];
16        int gSize = it.get_local_range()[0];
17        sycl::ext::oneapi::sub_group sg = it.get_sub_group();
18        int sgSize = sg.get_local_range()[0];
19        int sgGroup = sg.get_group_id()[0];
20
21        unsigned int histogram[NUM_BINS]; // histogram bins take less storage
22                                    // with smaller data type
23        for (int k = 0; k < NUM_BINS; k++) {
24          histogram[k] = 0;
25        }
26        for (int k = 0; k < blockSize; k++) {
27          unsigned long x =
28            sg.load(macc.get_pointer() + group * gSize * blockSize +
29                    sgGroup * sgSize * blockSize + sgSize * k);
30 #pragma unroll
31          for (int i = 0; i < 8; i++) {
32            unsigned int c = x & 0x1FU;
33            histogram[c] += 1;
34            x = x >> 8;
35          }
36        }
37
38        for (int k = 0; k < NUM_BINS; k++) {
39          hacc[k].fetch_add(histogram[k]);
40        }
41      });
42    });

```

### 6.3.4 Sharing Registers in Sub-group

Now we increase the histogram bins to 256:

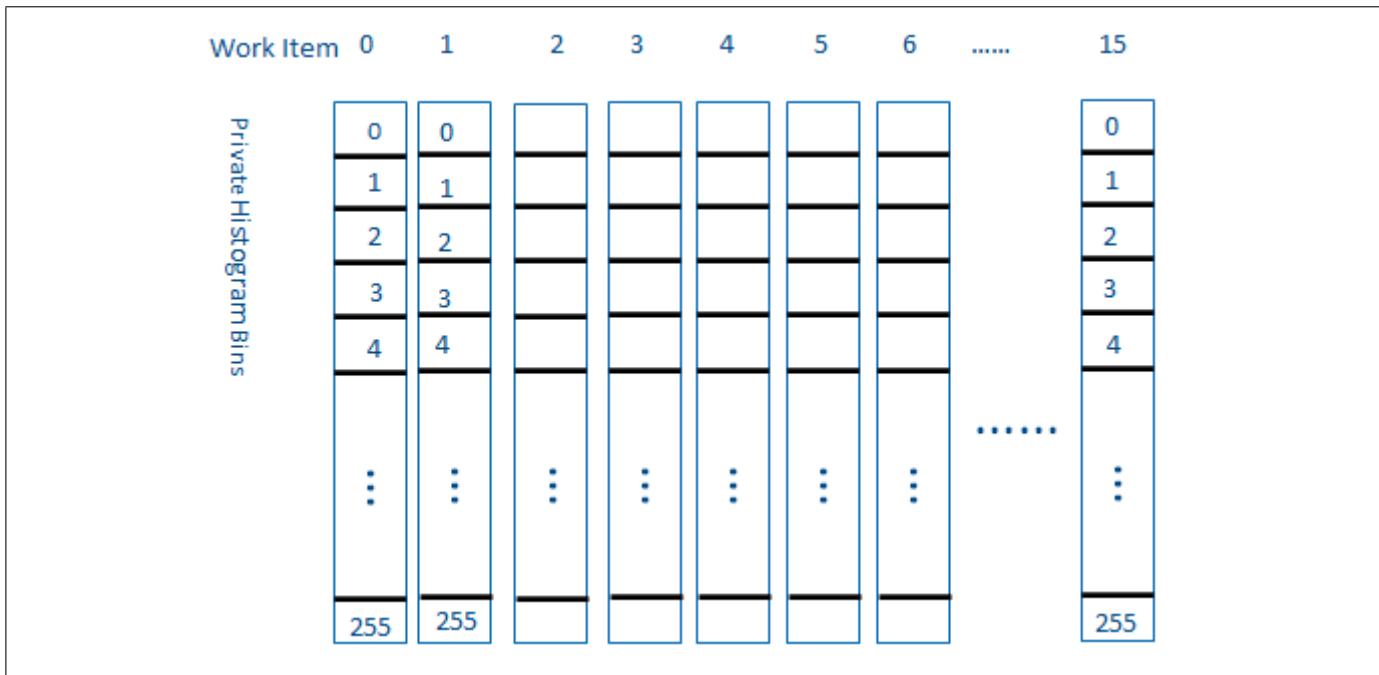
**Listing 34:** /examples/register/histogram256-int.cpp

```

1  constexpr int blockSize = 256;
2  constexpr int NUM_BINS = 256;
3
4  std::vector<unsigned long> hist(NUM_BINS, 0);
5
6  sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
7  sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);
8
9  auto e = q.submit([&](auto &h) {
10    sycl::accessor macc(mbuf, h, sycl::read_only);
11    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
12    h.parallel_for(
13      sycl::nd_range(sycl::range{N / blockSize}, sycl::range{64}), [=]
14      (sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
15        int group = it.get_group()[0];
16        int gSize = it.get_local_range()[0];
17        sycl::ext::oneapi::sub_group sg = it.get_sub_group();
18        int sgSize = sg.get_local_range()[0];
19        int sgGroup = sg.get_group_id()[0];
20
21        unsigned int
22          histogram[NUM_BINS]; // histogram bins take too much storage to be
23                           // promoted to registers
24        for (int k = 0; k < NUM_BINS; k++) {
25          histogram[k] = 0;
26        }
27        for (int k = 0; k < blockSize; k++) {
28          unsigned long x =
29            sg.load(macc.get_pointer() + group * gSize * blockSize +
30                    sgGroup * sgSize * blockSize + sgSize * k);
31 #pragma unroll
32          for (int i = 0; i < 8; i++) {
33            unsigned int c = x & 0x1FU;
34            histogram[c] += 1;
35            x = x >> 8;
36          }
37        }
38
39        for (int k = 0; k < NUM_BINS; k++) {
40          hacc[k].fetch_add(histogram[k]);
41        }
42      });
43    });

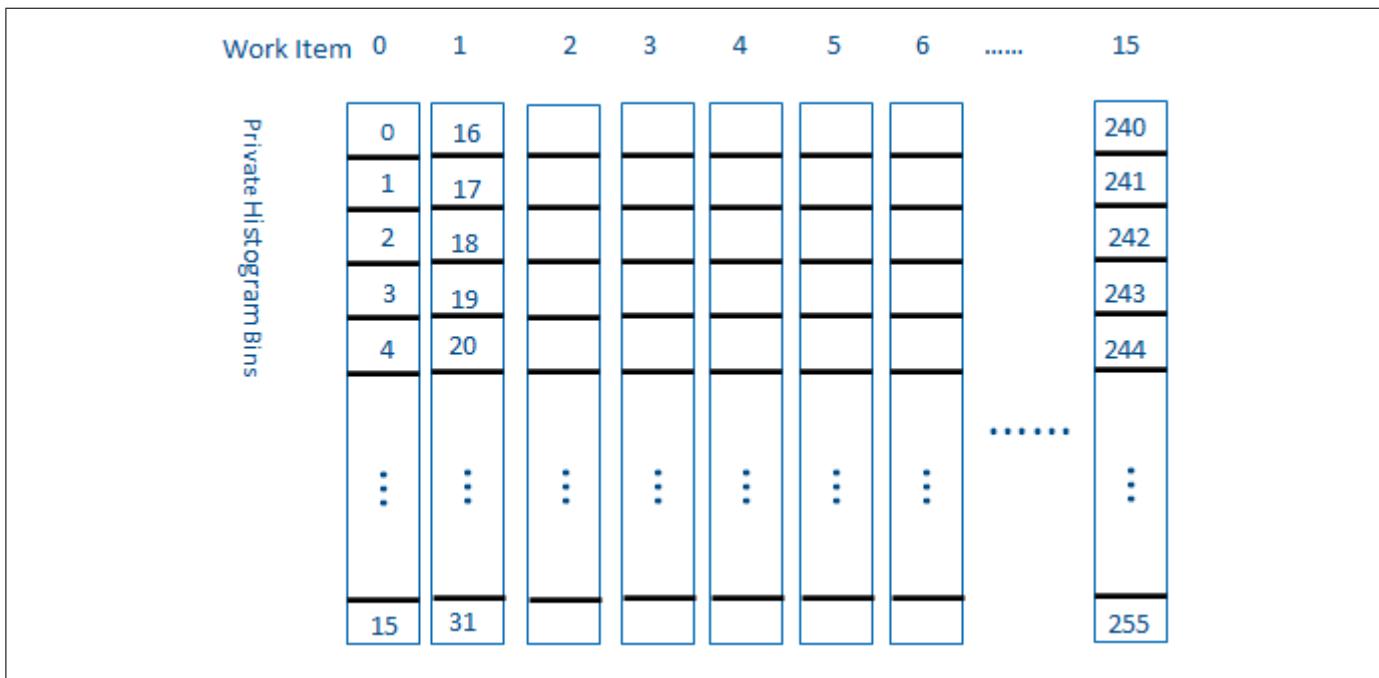
```

With 256 histogram bins, the performance degrades even with smaller data type unsigned integer. The storage of the private bins in each work item is too large for registers.



**Fig. 9:** Each Work Item Has 256 Private Histogram Bins

If the sub-group size is 16 as requested, we know that 16 work items are packed into one EU thread. We also know work items in the same sub-group can communicate and share data with each other very efficiently. If the work items in the same sub-group share the private histogram bins, only 256 private bins are needed for the whole sub-group, or 16 private bins for each work item instead.



**Fig. 10:** Sub-group Has 256 Private Histogram Bins

To share the histogram bins in the sub-group, each work item broadcasts its input data to every work item in the same sub-group. The work item that owns the corresponding histogram bin does the update.

## **Listing 35:**

/examples/registers/histogram256-int-shared-private.cpp

(continues on next page)

(continued from previous page)

```

48     for (int k = 0; k < NUM_BINS / 16; k++) {
49         hacc[16 * k + sg.get_local_id()[0]].fetch_add(histogram[k]);
50     }
51 };
52 }
```

### 6.3.5 Using Shared Local Memory

If the number of histogram bins gets larger to, for example, 1024, there will not be enough register space for private bins even the private bins are shared in the same sub-group. To reduce memory traffic, the local histogram bins can be allocated in the shared local memory and shared by work items in the same work-group. Please continue to the “Shared Local Memory” chapter and see how it is done in the histogram example there.

## 6.4 Shared Local Memory

Often work-items need to share data and communicate with each other. On one hand, all work-items in all work-groups can access global memory, so data sharing and communication can occur through global memory. However, due to its lower bandwidth and higher latency, sharing and communication through global memory is less efficient. On the other hand, work-items in a sub-group executing simultaneously in an execution unit (EU) thread can share data and communicate with each other very efficiently, but the number of work-items in a sub-group is usually small and the scope of data sharing and communication is very limited. Memory with higher bandwidth and lower latency accessible to a bigger scope of work-items is very desirable for data sharing communication among work-items. The shared local memory (SLM) in Intel® GPUs is designed for this purpose.

Each SubSlice of Intel® GPUs has its own SLM. Access to the SLM is limited to the EUs in the SubSlice or work-items in the same work-group scheduled to execute on the EUs of the same SubSlice. It is local to a SubSlice (or work-group) and shared by EUs in the same SubSlice (or work-items in the same work-group), so it is called SLM. Because it is on-chip in each SubSlice, the SLM has much higher bandwidth and much lower latency than global memory. Because it is accessible to all work-items in a work-group, the SLM can accommodate data sharing and communication among hundreds of work-items, depending on the work-group size.

It is often helpful to think of SLM as a work-group managed cache. When a work-group starts, work-items in the work-group can explicitly load data from global memory into SLM. The data stays in SLM during the lifetime of the work-group for faster access. Before the work-group finishes, the data in the SLM can be explicitly written back to the global memory by the work-items. After the work-group completes execution, the data in SLM is also gone and invalid. Data consistency between the SLM and the global memory is the program’s responsibility. Properly using SLM can make significant performance difference.

### 6.4.1 Shared Local Memory Size and Work-group Size

Because it is on-chip, the SLM has limited size. How much memory is available to a work-group is device-dependent and can be obtained by querying the device, e.g.:

**Listing 36:** /examples/slm/slm-size.cpp

```

1 std::cout << "Local Memory Size: "
2         << q.get_device().get_info<sycl::info::device::local_mem_size>()
3         << std::endl;

```

The output may look like:

```
Local Memory Size: 65536
```

The unit of the size is a byte. So this GPU device has 65,536 bytes or 64KB SLM for each work-group.

It is important to know the maximum SLM size a work-group can have. In a lot of cases, the total size of SLM available to a work-group is a non-constant function of the number of work-items in the work-group. The maximum SLM size can limit the total number of work-items in a group, i.e. work-group size. For example, if the maximum SLM size is 64KB and each work-item needs 512 bytes of SLM, the maximum work-group size cannot exceed 128.

#### 6.4.2 Bank Conflicts

The SLM is divided into equally sized memory banks that can be accessed simultaneously for high bandwidth. The total number of banks is device-dependent. At the time of writing, 64 consecutive bytes are stored in 16 consecutive banks at 4-byte (32-bit) granularity. Requests for access to different banks can be serviced in parallel, but requests to different addresses in the same bank cause a bank conflict and are serialized. Bank conflicts adversely affect performance. Consider this example:

**Listing 37:** /examples/slm/slm-bank-s16.cpp

```

1 constexpr int N = 32;
2 int *data = sycl::malloc_shared<int>(N, q);
3
4 auto e = q.submit([&](auto &h) {
5     sycl::accessor<int, 1, sycl::access::mode::read_write,
6             sycl::access::target::local>
7     slm(sycl::range(32 * 64), h);
8     h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{32}),
9                     [=](sycl::nd_item<1> it) {
10                     int i = it.get_global_linear_id();
11                     int j = it.get_local_linear_id();
12
13                     slm[j * 16] = 0;
14                     it.barrier(sycl::access::fence_space::local_space);
15
16                     for (int m = 0; m < 1024 * 1024; m++) {
17                         slm[j * 16] += i * m;
18                         it.barrier(sycl::access::fence_space::local_space);
19                     }
20
21                     data[i] = slm[j * 16];
22                 });
23 });

```

If the number of banks is 16, all work-items in the above example will read from and write to different addresses in the same bank. The memory bandwidth is 1/16 of full bandwidth.

The next example instead does not have SLM bank conflicts and achieves full memory bandwidth because every work-item reads from and writes to different addresses in different banks.

**Listing 38:** /examples/slm/slm-bank-s1.cpp

```

1  constexpr int N = 32;
2  int *data = sycl::malloc_shared<int>(N, q);
3
4  auto e = q.submit([&](auto &h) {
5      sycl::accessor<int, 1, sycl::access::mode::read_write,
6          sycl::access::target::local>
7      slm(sycl::range(32 * 64), h);
8      h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{32}), [=](sycl::nd_item<1> it) {
9          int i = it.get_global_linear_id();
10         int j = it.get_local_linear_id();
11
12         slm[j] = 0;
13         it.barrier(sycl::access::fence_space::local_space);
14
15         for (int m = 0; m < 1024 * 1024; m++) {
16             slm[j] += i * m;
17             it.barrier(sycl::access::fence_space::local_space);
18         }
19
20         data[i] = slm[j];
21     });
22 });
23

```

### 6.4.3 Data Sharing and Work-group Barriers

Let us consider the histogram with 256 bins example from the “Avoiding Register Spills” chapter once again.

**Listing 39:**  
/examples/registers/histogram256-int-shared-private.cpp

```

1  constexpr int blockSize = 256;
2  constexpr int NUM_BINS = 256;
3
4  std::vector<unsigned long> hist(NUM_BINS, 0);
5
6  sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
7  sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);
8
9  auto e = q.submit([&](auto &h) {
10     sycl::accessor macc(mbuf, h, sycl::read_only);
11     auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
12     h.parallel_for(
13         sycl::nd_range(sycl::range{N / blockSize}, sycl::range{64}), [=

```

(continues on next page)

(continued from previous page)

```

14 ](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
15     int group = it.get_group()[0];
16     int gSize = it.get_local_range()[0];
17     sycl::ext::oneapi::sub_group sg = it.get_sub_group();
18     int sgSize = sg.get_local_range()[0];
19     int sgGroup = sg.get_group_id()[0];
20
21     unsigned int
22         histogram[NUM_BINS / 16]; // histogram bins take too much storage
23                                     // to be promoted to registers
24     for (int k = 0; k < NUM_BINS / 16; k++) {
25         histogram[k] = 0;
26     }
27     for (int k = 0; k < blockSize; k++) {
28         unsigned long x =
29             sg.load(macc.get_pointer() + group * gSize * blockSize +
30                     sgGroup * sgSize * blockSize + sgSize * k);
31 // subgroup size is 16
32 #pragma unroll
33         for (int j = 0; j < 16; j++) {
34             unsigned long y = sycl::group_broadcast(sg, x, j);
35 #pragma unroll
36             for (int i = 0; i < 8; i++) {
37                 unsigned int c = y & 0xFF;
38                 // (c & 0xF) is the workitem in which the bin resides
39                 // (c >> 4) is the bin index
40                 if (sg.get_local_id()[0] == (c & 0xF)) {
41                     histogram[c >> 4] += 1;
42                 }
43                 y = y >> 8;
44             }
45         }
46     }
47
48     for (int k = 0; k < NUM_BINS / 16; k++) {
49         hacc[16 * k + sg.get_local_id()[0]].fetch_add(histogram[k]);
50     }
51 });
52 });

```

We optimized this example to use integer data type instead of long and to share registers in the sub-group so that the private histogram bins can fit in registers for optimal performance. If we need a larger bin size (e.g., 1024), it is inevitable that the private histogram bins will spill to global memory.

The histogram bins can be shared by work-items in a work-group as long as each bin is updated atomically.

#### **Listing 40:** /examples/slm/histogram-slm-1024.cpp

```

1 constexpr int NUM_BINS = 1024;
2 constexpr int blockSize = 256;
3

```

(continues on next page)

(continued from previous page)

```
4 std::vector<unsigned long> hist(NUM_BINS, 0);
5 sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
6 sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);
7
8 auto e = q.submit([&](auto &h) {
9     sycl::accessor macc(mbuf, h, sycl::read_only);
10    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
11    sycl::accessor<unsigned int, 1, sycl::access::mode::atomic,
12                  sycl::access::target::local>
13        local_histogram(sycl::range<NUM_BINS>, h);
14    h.parallel_for(
15        sycl::nd_range(sycl::range{N / blockSize}, sycl::range{64}),
16        [=](sycl::nd_item<1> it) {
17            int group = it.get_group()[0];
18            int gSize = it.get_local_range()[0];
19            sycl::ext::oneapi::sub_group sg = it.get_sub_group();
20            int sgSize = sg.get_local_range()[0];
21            int sgGroup = sg.get_group_id()[0];
22
23            int factor = NUM_BINS / gSize;
24            int local_id = it.get_local_id()[0];
25            if ((factor <= 1) && (local_id < NUM_BINS)) {
26                local_histogram[local_id].store(0);
27            } else {
28                for (int k = 0; k < factor; k++) {
29                    local_histogram[gSize * k + local_id].store(0);
30                }
31            }
32            it.barrier(sycl::access::fence_space::local_space);
33
34            for (int k = 0; k < blockSize; k++) {
35                unsigned long x =
36                    sg.load(macc.get_pointer() + group * gSize * blockSize +
37                            sgGroup * sgSize * blockSize + sgSize * k);
38                local_histogram[x & 0x3FFU].fetch_add(1);
39                local_histogram[(x >> 16) & 0x3FFU].fetch_add(1);
40                local_histogram[(x >> 32) & 0x3FFU].fetch_add(1);
41                local_histogram[(x >> 48) & 0x3FFU].fetch_add(1);
42            }
43            it.barrier(sycl::access::fence_space::local_space);
44
45            if ((factor <= 1) && (local_id < NUM_BINS)) {
46                hacc[local_id].fetch_add(local_histogram[local_id].load());
47            } else {
48                for (int k = 0; k < factor; k++) {
49                    hacc[gSize * k + local_id].fetch_add(
50                        local_histogram[gSize * k + local_id].load());
51                }
52            }
53        });
54});
```

When the work-group is started, each work-item in the work-group initializes a portion of the histogram bins in SLM to 0 (code in lines 21-27 in the above example). We could designate one work-item to initialize all the histogram bins, but it is usually more efficient to divide the job among all work-items in the work-group.

The work-group barrier after initialization at line 28 guarantees that all histogram bins are initialized to 0 before any work-item updates any bins.

Because the histogram bins in SLM are shared among all work-items, updates to any bin by any work-item has to be atomic.

The global histograms are updated once the local histograms in the work-group is completed. But before reading the local SLM bins to update the global bins, a work-group barrier is again called at line 43 to make sure all work-items have completed their work.

When SLM data is shared, work-group barriers are often required for work-item synchronization. The barrier has a cost and the cost may increase with a larger work-group size. It is always a good idea to try different work-group sizes to find the best one for your application.

We also have an SLM version of histogram with 256 bins in the examples folder. You can compare its performance with the performance of the version using registers. You may get some surprising results and want to think what further optimizations can be done for more performance. We leave this as an exercise.

#### 6.4.4 Using SLM as Cache

We sometimes find it is more desirable to have the application manage caching of some hot data than to have the hardware do it automatically for us. With the application managing data caching directly, whenever the data is needed, we know exactly where the data is and the cost to access it. The SLM can be used for such purpose.

Consider the following 1-D convolution example:

**Listing 41:** /examples/slm/convolution-global.cpp

```

1    sycl::buffer<int> ibuf(input.data(), N);
2    sycl::buffer<int> obuf(output.data(), N);
3    sycl::buffer<int> kbuf(kernel.data(), M);
4
5    auto e = q.submit([&](auto &h) {
6        sycl::accessor iacc(ibuf, h, sycl::read_only);
7        sycl::accessor oacc(obuf, h);
8        sycl::accessor kacc(kbuf, h, sycl::read_only);
9
10       h.parallel_for(sycl::nd_range<1>(sycl::range{N}, sycl::range{256}),
11                      [=](sycl::nd_item<1> it) {
12              int i = it.get_global_linear_id();
13              int group = it.get_group()[0];
14              int gSize = it.get_local_range()[0];
15
16              int t = 0;
17
18              if ((group == 0) || (group == N / gSize - 1)) {
19                  if (i < M / 2) {
20                      for (int j = M / 2 - i, k = 0; j < M; j++, k++) {

```

(continues on next page)

(continued from previous page)

```

21         t += iacc[k] * kacc[j];
22     }
23 } else {
24     if (i + M / 2 >= N) {
25         for (int j = 0, k = i - M / 2; j < M / 2 + N - i;
26             j++, k++) {
27             t += iacc[k] * kacc[j];
28         }
29     } else {
30         for (int j = 0, k = i - M / 2; j < M; j++, k++) {
31             t += iacc[k] * kacc[j];
32         }
33     }
34 }
35 } else {
36     for (int j = 0, k = i - M / 2; j < M; j++, k++) {
37         t += iacc[k] * kacc[j];
38     }
39 }
40
41     oacc[i] = t;
42 });
43 });

```

The example convolves an integer array of 8192 x 8192 elements using a kernel array of 257 elements and writes the result to an output array. Each work-item convolves one element. To convolve one element, however, up to 256 neighboring elements are needed.

Noticing each input element is used by multiple work-items, we can preload all input elements needed by a whole work-group into SLM. Later, when an element is needed, it can be loaded from SLM instead of global memory.

**Listing 42:** /examples/slm/convolution-slm-cache.cpp

```

1    sycl::buffer<int> ibuf(input.data(), N);
2    sycl::buffer<int> obuf(output.data(), N);
3    sycl::buffer<int> kbuf(kernel.data(), M);
4
5    auto e = q.submit([&](auto &h) {
6        sycl::accessor iacc(ibuf, h, sycl::read_only);
7        sycl::accessor oacc(obuf, h);
8        sycl::accessor kacc(kbuf, h, sycl::read_only);
9        sycl::accessor<int, 1, sycl::access::mode::read_write,
10                     sycl::access::target::local>
11                     ciacc(sycl::range(256 + (M / 2) * 2), h);
12
13        h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{256}),
14                      [=](sycl::nd_item<1> it) {
15                          int i = it.get_global_linear_id();
16                          int group = it.get_group()[0];
17                          int gSize = it.get_local_range()[0];
18                          int local_id = it.get_local_id()[0];

```

(continues on next page)

(continued from previous page)

```

19         ciacc[local_id + M / 2] = iacc[i];
20
21     if (local_id == 0) {
22         if (group == 0) {
23             for (int j = 0; j < M / 2; j++) {
24                 ciacc[j] = 0;
25             }
26         } else {
27             for (int j = 0, k = i - M / 2; j < M / 2; j++, k++) {
28                 ciacc[j] = iacc[k];
29             }
30         }
31     }
32 }
33 if (local_id == gSize - 1) {
34     if (group == it.get_group_range()[0] - 1) {
35         for (int j = gSize + M / 2;
36               j < gSize + M / 2 + M / 2; j++) {
37             ciacc[j] = 0;
38         }
39     } else {
40         for (int j = gSize + M / 2, k = i + 1;
41               j < gSize + M / 2 + M / 2; j++, k++) {
42             ciacc[j] = iacc[k];
43         }
44     }
45 }
46
47     it.barrier(sycl::access::fence_space::local_space);
48
49     int t = 0;
50     for (int j = 0, k = local_id; j < M; j++, k++) {
51         t += ciacc[k] * kacc[j];
52     }
53
54     oacc[i] = t;
55 });
56 });

```

When the work-group starts, all input elements needed by each work-item are loaded into SLM. Each work-item, except the first one and the last one, loads one element into SLM. The first work-item loads neighbors on the left of the first element and the last work item loads neighbors on the right of the last element in the SLM. If no neighbors exist, elements in SLM are filled with 0s.

Before convolution starts in each work-item, a local barrier is called to make sure all input elements are loaded into SLM.

The convolution in each work-item is straightforward. All neighboring elements are loaded from the faster SLM instead of global memory.

## 6.5 Removing Conditional Checks

In [Sub-groups](#), we learned that SIMD divergence can negatively affect performance. If all work items in a sub-group execute the same instruction, the SIMD lanes are maximally utilized. If one or more work items take a divergent path, then both paths have to be executed before they merge.

Divergence is caused by conditional checks though not all conditional checks cause divergence. Some conditional checks, even when they do not cause SIMD divergence, can still be performance hazards. In general, removing conditional checks can help performance.

### 6.5.1 Padding Buffers to Remove Conditional Checks

Look at the convolution example from [Shared Local Memory](#):

**Listing 43:** /examples/slm/convolution-global.cpp

```

1    sycl::buffer<int> ibuf(input.data(), N);
2    sycl::buffer<int> obuf(output.data(), N);
3    sycl::buffer<int> kbuf(kernel.data(), M);
4
5    auto e = q.submit([&](auto &h) {
6        sycl::accessor iacc(ibuf, h, sycl::read_only);
7        sycl::accessor oacc(obuf, h);
8        sycl::accessor kacc(kbuf, h, sycl::read_only);
9
10       h.parallel_for(sycl::nd_range<1>(sycl::range{N}, sycl::range{256}),
11                      [=](sycl::nd_item<1> it) {
12                          int i = it.get_global_linear_id();
13                          int group = it.get_group()[0];
14                          int gSize = it.get_local_range()[0];
15
16                          int t = 0;
17
18                          if ((group == 0) || (group == N / gSize - 1)) {
19                              if (i < M / 2) {
20                                  for (int j = M / 2 - i, k = 0; j < M; j++, k++) {
21                                      t += iacc[k] * kacc[j];
22                                  }
23                              } else {
24                                  if (i + M / 2 >= N) {
25                                      for (int j = 0, k = i - M / 2; j < M / 2 + N - i;
26                                           j++, k++) {
27                                          t += iacc[k] * kacc[j];
28                                      }
29                              } else {
30                                  for (int j = 0, k = i - M / 2; j < M; j++, k++) {
31                                      t += iacc[k] * kacc[j];
32                                  }
33                              }
34                          }
35                      } else {
36                          for (int j = 0, k = i - M / 2; j < M; j++, k++) {

```

(continues on next page)

(continued from previous page)

```
37             t += iacc[k] * kacc[j];
38         }
39     }
40
41     oacc[i] = t;
42 });
43});
```

The nested if-then-else conditional checks are necessary to take care of the first and last 128 elements in the input so indexing will not run out of bound. If we pad enough 0s before and after the input array, these conditional checks can be safely removed:

**Listing 44:** /examples/conditionals/convolution-global-conditionals.cpp

(continues on next page)

(continued from previous page)

```

37             oacc[i] = t;
38         });
39     });
40     q.wait();
41
42     size_t kernel_ns = (e.template get_profiling_info<
43                         sycl::info::event_profiling::command_end>() -
44                         e.template get_profiling_info<
45                         sycl::info::event_profiling::command_start>());
46     std::cout << "Kernel Execution Time Average: total = " << kernel_ns * 1e-6
47             << " msec" << std::endl;
48 }
49

```

## 6.5.2 Replacing Conditional Checks with Relational Functions

Another way to remove conditional checks is to replace them with relational functions, especially built-in relational functions. It is strongly recommended to use built-in functions if one is available. DPC++ provides a rich set of built-in relational functions like `select()`, `min()`, `max()`. In many cases cases you can use these functions can be used to replace conditional checks and achieve better performance.

Consider the convolution example again. The if-then-else conditional checks can be replaced with built-in functions `min()` and `max()`.

**Listing 45:** /examples/conditionals/convolution-global-conditionals-min-max.cpp

```

1  sycl::buffer<int> ibuf(input.data(), N);
2  sycl::buffer<int> obuf(output.data(), N);
3  sycl::buffer<int> kbuf(kernel.data(), M);
4
5  auto e = q.submit([&](auto &h) {
6      sycl::accessor iacc(ibuf, h, sycl::read_only);
7      sycl::accessor oacc(obuf, h);
8      sycl::accessor kacc(kbuf, h, sycl::read_only);
9
10     h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{256}),
11                     [=](sycl::nd_item<1> it) {
12             int i = it.get_global_linear_id();
13             int t = 0;
14             int startj = sycl::max<int>(M / 2 - i, 0);
15             int endj = sycl::min<int>(M / 2 + N - i, M);
16             int startk = sycl::max<int>(i - M / 2, 0);
17             for (int j = startj, k = startk; j < endj; j++, k++) {
18                 t += iacc[k] * kacc[j];
19             }
20             oacc[i] = t;
21         });
22     });
23

```

## 6.6 Kernel Launch

In DPC++, work is performed by enqueueing kernels into queues targeting specific devices. These kernels are submitted by the host to the device and then they get executed by the device and results are sent back. The kernel submission by the host and the actual start of execution do not happen immediately - they are asynchronous and as such we have to keep track of the following timings associated with a kernel.

**Kernel submission start time** This is the at which the host starts the process of submitting the kernel.

**Kernel submission end time** This is the time at which the host finished submitting the kernel. The host performs multiple tasks like queuing the arguments, allocating resources in the runtime for the kernel to start execution on the device.

**Kernel Launch time** This is the time at which the kernel that was submitted by the host starts executing on the device. Note that this is not exactly same as the kernel submission end time. There is a lag between the submission end time and the kernel launch time which depends on the availability of the device. It is possible for the host to queue up a number of kernels for execution before the kernels are actually launched for execution. More over, there are a few data transfers that need to happen before the actual kernel starts execution which is typically not accounted separately from kernel launch time.

**Kernel completion time** This is the time at which the kernel finishes execution on the device. The current generation of devices are non-preemptive which means that once a kernel starts it has to complete its execution.

Tools like VTune™ Profiler (**vtune**), **clIntercept** and **zelIntercept** provide a visual timeline for each of the above times for every kernel in the application.

The following simple example, shows time being measured for the kernel execution and this will involve the kernel submission time on the host and the kernel execution time on the device and any data transfer times (since there are no buffers or memory this is usually zero in this case).

**Listing 46:** /examples/kernels/launch.cpp

```

1 void emptyKernel1(sycl::queue &q) {
2     Timer timer;
3     for (int i = 0; i < iters; ++i)
4         q.parallel_for(1, [=](auto id) {
5             /* NOP */
6         }).wait();
7     std::cout << " emptyKernel1: Elapsed time: " << timer.Elapsed() / iters
8             << " sec\n";
9 } // end emptyKernel1

```

The same code without the wait at the end of the parallel\_for measures the time it takes for the host to submit the kernel to the runtime.

**Listing 47:** /examples/kernels/launch.cpp

```

1 void emptyKernel2(sycl::queue &q) {
2     Timer timer;
3     for (int i = 0; i < iters; ++i)
4         q.parallel_for(1, [=](auto id) {

```

(continues on next page)

(continued from previous page)

```

5     /* NOP */
6 };
7 std::cout << " emptyKernel2: Elapsed time: " << timer.Elapsed() / iters
8     << " sec\n";

```

These overheads are highly dependent on the backend runtime being used and the processing power of the host.

A way to measure the actual kernel execution time on the device is using DPC++ built-in profiling API. The following code demonstrates usage of the DPC++ profiling API to profile kernel execution times. It also shows the kernel submission time. There is no way to programmatically measure the kernel launch time since it is dependent on the runtime and the device driver - we have to depend on profiling tools to get this information.

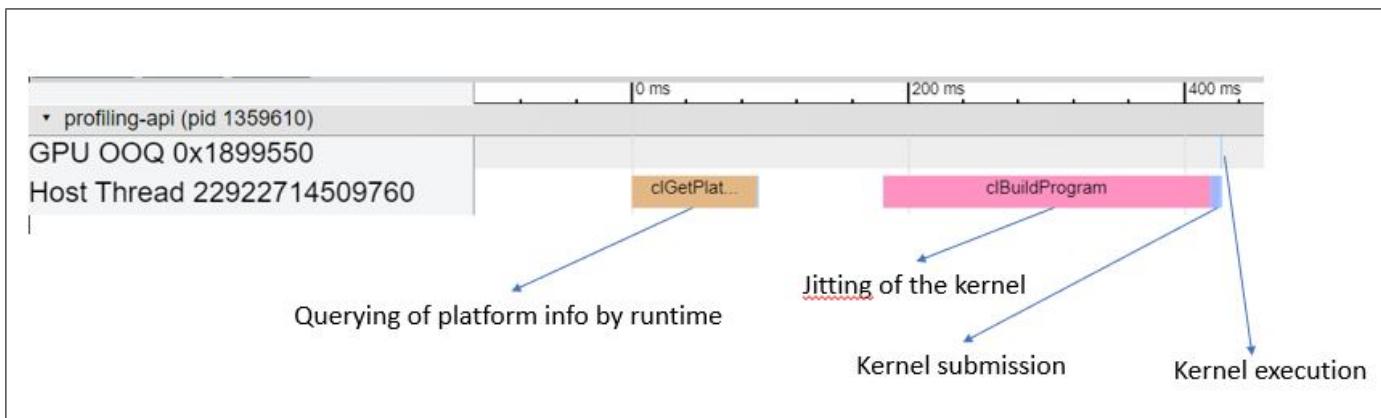
**Listing 48:** /examples/kernels/profiling-api.cpp

```

1 #include <CL/sycl.hpp>
2
3 class Timer {
4 public:
5     Timer() : start_(std::chrono::steady_clock::now()) {}
6
7     double Elapsed() {
8         auto now = std::chrono::steady_clock::now();
9         return std::chrono::duration_cast<Duration>(now - start_).count();
10    }
11
12 private:
13     using Duration = std::chrono::duration<double>;
14     std::chrono::steady_clock::time_point start_;
15 };
16
17 int main() {
18     Timer timer;
19     sycl::queue q{sycl::property::queue::enable_profiling()};
20     auto evt = q.parallel_for(1000, [=](auto id) {
21         /* kernel statements here */
22     });
23     double t1 = timer.Elapsed();
24     evt.wait();
25     double t2 = timer.Elapsed();
26     auto startK =
27         evt.get_profiling_info<sycl::info::event_profiling::command_start>();
28     auto endK =
29         evt.get_profiling_info<sycl::info::event_profiling::command_end>();
30     std::cout << "Kernel submission time: " << t1 << "secs\n";
31     std::cout << "Kernel submission + execution time: " << t2 << "secs\n";
32     std::cout << "Kernel execution time: "
33             << ((double)(endK - startK)) / 1000000.0 << "secs\n";
34
35     return 0;
36 }

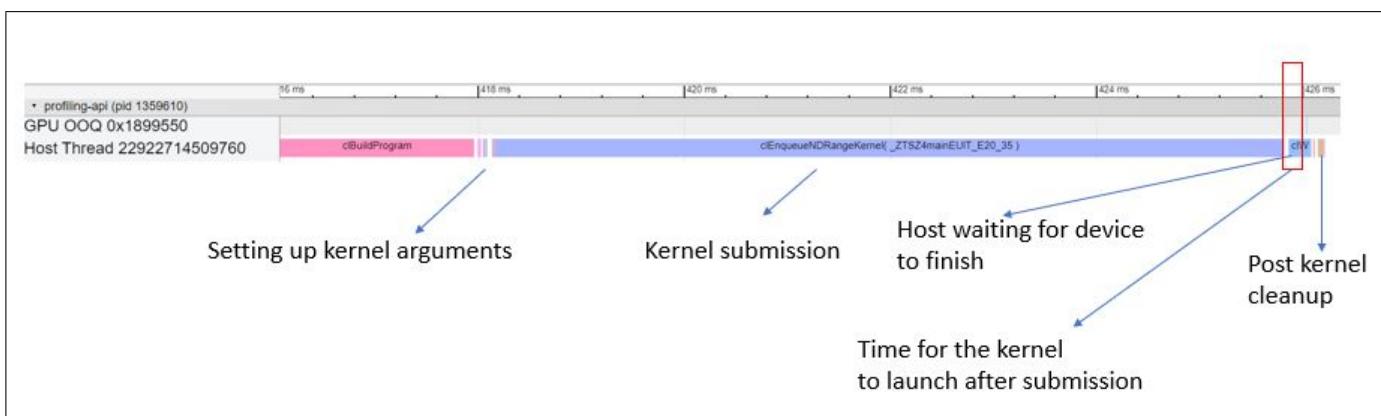
```

The following picture shows the timeline of the execution for the above example. This picture is generated from running `clIntercept` to generate a trace file and using Chrome\* tracing to visualize the timeline. In this timeline there are two swim lanes - one for the host side and another for the device side. Notice that the only activity on device side is the execution of the submitted kernel. A significant amount of work is done on the host side to get the kernel prepared for execution. In this case since the kernel is very small, total execution time is dominated by the JIT compilation of the kernel which is the block labeled `clBuildProgram` in the figure below.



**Fig. 11:** Timeline of kernel execution

The following picture is the zoomed in version to show the detail of the functions called on the host side to submit the kernel. Here the time is dominated by the `clEnqueueNDRangeKernel`. Also notice that there is a lag between the completion of kernel submission on the host and the actual launch of the kernel on the device.



**Fig. 12:** Functions called on host to submit the kernel

## 6.7 Using Libraries for Accelerator Offload

This section discusses using efficient functions from libraries like oneAPI Math Kernel Library (oneMKL) or oneAPI Deep Neural Network Library (oneDNN) instead of hand-coded alternatives. Unless you're an expert studying a particular mathematical operation, it's usually a bad idea to write your own version of that operation. For example, matrix multiplication is a common, straightforward mathematical operation:

$$C_{m,n} = A_{m,k} \times B_{k,n} = \sum_{k=1}^K A_{m,k} \times B_{k,n}$$

It's also easy to implement with just a few lines of code:

**Listing 49:** /examples/libraries-kernel/naive\_matmul.cpp

```

1 // Multiply matrices A and B
2 for (m = 0; m < M; m++) {
3     for (n = 0; n < N; n++) {
4         C[m][n] = 0.0;
5         for (k = 0; k < K; k++) {
6             C[m][n] += A[m][k] * B[k][n];
7         }
8     }
9 } // End matrix multiplication

```

However, this naive implementation won't give the best possible performance. Simple visual inspection of the inner loop shows non-contiguous memory access for matrix B. Cache reuse, and hence performance, will be poor.

It's not difficult to port the naive algorithm to Data Parallel C++ (DPC++) to offload the matrix multiplication kernel to an accelerator. The following code initializes the queue to submit work to the default device and allocates space for the matrices in unified shared memory (USM):

**Listing 50:** /examples/libraries-kernel/naive\_matmul\_sycl.cpp

```

1 // Initialize SYCL queue
2 sycl::queue Q(sycl::default_selector{});
3 auto sycl_device = Q.get_device();
4 auto sycl_context = Q.get_context();
5 std::cout << "Running on: "
6             << Q.get_device().get_info<sycl::info::device::name>() << std::endl;
7
8 // Allocate matrices A, B, and C in USM
9 auto A = sycl::malloc_shared<float *>(M, sycl_device, sycl_context);
10 for (m = 0; m < M; m++)
11     A[m] = sycl::malloc_shared<float>(K, sycl_device, sycl_context);
12
13 auto B = sycl::malloc_shared<float *>(K, sycl_device, sycl_context);
14 for (k = 0; k < K; k++)
15     B[k] = sycl::malloc_shared<float>(N, sycl_device, sycl_context);
16
17 auto C = sycl::malloc_shared<float *>(M, sycl_device, sycl_context);
18 for (m = 0; m < M; m++)
19     C[m] = sycl::malloc_shared<float>(N, sycl_device, sycl_context);
20
21 // Initialize matrices A, B, and C

```

Data in USM can be moved between host and device memories by the DPC++ runtime. Explicit buffering is not required. To offload the computation to the default accelerator, it is converted to a DPC++ kernel and submitted to the queue:

**Listing 51:** /examples/libraries-kernel/naive\_matmul\_sycl.cpp

```

1 // Offload matrix multiplication kernel
2 Q.parallel_for(sycl::range<2>{M, N}, [=](sycl::id<2> id) {
3     unsigned int m = id[0];
4     unsigned int n = id[1];
5
6     float sum = 0.0;
7     for (unsigned int k = 0; k < K; k++)
8         sum += A[m][k] * B[k][n];
9
10    C[m][n] = sum;
11 }) .wait(); // End matrix multiplication

```

However, simply offloading such code to an accelerator is unlikely to restore performance. In fact, performance gets worse. Badly written code is still badly written whether it runs on the host or a device. The table below shows how the naive matrix multiplication implementation performs on matrices of different shapes.

Common, computationally-demanding operations like matrix multiplication are well-studied. Experts have devised a number of algorithms that give better performance than naive implementations of the basic mathematical formulas. They also use tuning techniques like cache blocking and loop unrolling to achieve performance regardless of the shapes of matrices A and B.

oneMKL provides an optimized general matrix multiplication function (`oneapi::mkl::blas::gemm`) that gives high performance on the host processor or a variety of accelerator devices. The matrices are allocated in USM as before, and passed to the `gemm` function along with the device queue, matrix dimensions, and various other options:

**Listing 52:** /examples/libraries-kernel/matmul\_onemkl.cpp

```

1 // Offload matrix multiplication
2 float alpha = 1.0, beta = 0.0;
3 oneapi::mkl::transpose transA = oneapi::mkl::transpose::nontrans;
4 oneapi::mkl::transpose transB = oneapi::mkl::transpose::nontrans;
5 sycl::event gemm_done;
6 std::vector<sycl::event> gemm_dependencies;
7 gemm_done = oneapi::mkl::blas::gemm(Q, transA, transB, M, N, K, alpha, A, M,
8                                     B, K, beta, C, M, gemm_dependencies);
9 gemm_done.wait();

```

As expected, the library function gives better performance and is more versatile than the naive implementations. For example, the library function can transpose one or both matrices before multiplication, if necessary.

**Table 8:** Matrix A Dimensions (Time in Seconds)

Implementation	4000 x 4000	8000 x 2000	2000 x 8000	Processor
Naive DPC++	19.2	38.1	9.8	Gen9
oneMKL gemm	0.9	1.3	0.8	Gen9

This simple example illustrates the separation of concerns between application developers and tuning experts.

The former should rely on the latter to encapsulate common computations in highly-optimized libraries. The oneAPI specification defines many libraries to help create accelerated applications, e.g.:

- oneMKL for math operations
- oneDAL for data analytics and machine learning
- oneDNN for the development of deep learning frameworks
- oneVPL for video processing

Check whether your required operation is already available in a oneAPI library before implementing it from scratch.

## 6.8 Using Standard Library Functions in DPC++ Kernels

Some, but not all, standard C++ functions can be called inside DPC++ kernels. See Chapter 18 (Libraries) of [Data Parallel C++](#) for an overview of supported functions. A simple example is provided here to illustrate what happens when an unsupported function is called from a DPC++ kernel. The following program generates a sequence of random numbers using the `rand()` function:

**Listing 53:** /examples/libraries-stdlib/external\_rand.cpp

```

1 // Compile:
2 // dpcpp -D{HOST|CPU|GPU} -std=c++17 -fsycl external_rand.cpp -o external_rand
3
4 #include <CL/sycl.hpp>
5 #include <iostream>
6 #include <random>
7
8 #define N 5
9
10 extern SYCL_EXTERNAL int rand(void);
11
12 int main(int argc, char **argv) {
13 #if defined HOST
14     sycl::queue Q(sycl::host_selector{});
15 #elif defined CPU
16     sycl::queue Q(sycl::cpu_selector{});
17 #elif defined GPU
18     sycl::queue Q(sycl::gpu_selector{});
19 #endif
20
21     std::cout << "Running on: "
22             << Q.get_device().get_info<sycl::info::device::name>() << std::endl;
23
24 // Attempt to use rand() inside a DPC++ kernel
25 auto test1 = sycl::malloc_shared<float>(N, Q.get_device(), Q.get_context());
26
27 srand((unsigned)time(NULL));
28 Q.parallel_for(N, [=](auto idx) {
29     test1[idx] = (float)rand() / (float)RAND_MAX;

```

(continues on next page)

(continued from previous page)

```
30     }).wait();  
31  
32     // Show the random number sequence  
33     for (int i = 0; i < N; i++)  
34         std::cout << test1[i] << std::endl;  
35  
36     // Cleanup  
37     sycl::free(test1, Q.get_context());  
38 }
```

The program can be compiled to execute the DPC++ kernel on host (i.e., the SYCL host\_selector), the CPU (i.e., cpu\_selector), or GPU (i.e., gpu\_selector) devices. It compiles without errors on all three devices, and runs correctly on the CPU, but fails when run on the GPU:

```
$ dpcpp -DHOST -std=c++17 -fsycl external_rand.cpp -o external_rand  
$ ./external_rand  
Running on: Intel(R) Xeon(R) E-2176G CPU @ 3.70GHz  
0.572586  
0.691008  
0.451763  
0.793325  
0.000884  
  
$ dpcpp -DCPU -std=c++17 -fsycl external_rand.cpp -o external_rand  
$ ./external_rand  
Running on: Intel(R) Xeon(R) E-2176G CPU @ 3.70GHz  
0.141417  
0.821271  
0.898045  
0.218854  
0.304283  
  
$ dpcpp -DGPU -std=c++17 -fsycl external_rand.cpp -o external_rand  
$ ./external_rand  
Running on: Intel(R) Graphics Gen9 [0x3e96]  
terminate called after throwing an instance of 'cl::sycl::compile_program_error'  
what(): The program was built for 1 devices  
Build program log for 'Intel(R) Graphics Gen9 [0x3e96]':  
  
error: undefined reference to `rand()'  
  
error: backend compiler failed build.  
-11 (CL_BUILD_PROGRAM_FAILURE)  
Aborted
```

The failure occurs during just-in-time (JIT) compilation because of an undefined reference to rand(). Even though this function is declared SYCL\_EXTERNAL, there's no SYCL equivalent to the rand() function on the GPU device.

Fortunately, the DPC++ library contains alternatives to many standard C++ functions, including those to generate random numbers. The following example shows equivalent functionality using the Intel® oneAPI DPC++ Library ([oneDPL](#)) and the Intel® oneAPI Math Kernel Library ([oneMKL](#)):

**Listing 54:** /examples/libraries-stdlib/rng\_test.cpp

```

1 #include <CL/sycl.hpp>
2 #include <iostream>
3 #include <oneapi/dpl/random>
4 #include <oneapi/mkl/rng.hpp>
5
6 int main(int argc, char **argv) {
7     unsigned int N = (argc == 1) ? 20 : std::stoi(argv[1]);
8     if (N < 20)
9         N = 20;
10
11    // Generate sequences of random numbers between [0.0, 1.0] using oneDPL and
12    // oneMKL
13    sycl::queue Q(sycl::gpu_selector{});
14    std::cout << "Running on: "
15        << Q.get_device().get_info<sycl::info::device::name>() << std::endl;
16
17    auto test1 = sycl::malloc_shared<float>(N, Q.get_device(), Q.get_context());
18    auto test2 = sycl::malloc_shared<float>(N, Q.get_device(), Q.get_context());
19
20    std::uint32_t seed = (unsigned)time(NULL); // Get RNG seed value
21
22    // oneDPL random number generator on GPU device
23    clock_t start_time = clock(); // Start timer
24
25    Q.parallel_for(N, [=](auto idx) {
26        oneapi::dpl::minstd_rand rng_engine(seed, idx); // Initialize RNG engine
27        oneapi::dpl::uniform_real_distribution<float>
28            rng_distribution; // Set RNG distribution
29        test1[idx] = rng_distribution(rng_engine); // Generate RNG sequence
30    }).wait();
31
32    clock_t end_time = clock(); // Stop timer
33    std::cout << "oneDPL took " << float(end_time - start_time) / CLOCKS_PER_SEC
34        << " seconds to generate " << N
35        << " uniformly distributed random numbers." << std::endl;
36
37    // oneMKL random number generator on GPU device
38    start_time = clock(); // Start timer
39
40    oneapi::mkl::rng::mcg31ml engine(
41        Q, seed); // Initialize RNG engine, set RNG distribution
42    oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
43        rng_distribution(0.0, 1.0);
44    oneapi::mkl::rng::generate(rng_distribution, engine, N, test2)
45        .wait(); // Generate RNG sequence
46
47    end_time = clock(); // Stop timer
48    std::cout << "oneMKL took " << float(end_time - start_time) / CLOCKS_PER_SEC
49        << " seconds to generate " << N
50        << " uniformly distributed random numbers." << std::endl;

```

(continues on next page)

(continued from previous page)

```

51 // Show first ten random numbers from each method
52 std::cout << std::endl
53     << "oneDPL"
54     << "\t"
55     << "oneMKL" << std::endl;
56 for (int i = 0; i < 10; i++)
57     std::cout << test1[i] << " " << test2[i] << std::endl;
58
59 // Show last ten random numbers from each method
60 std::cout << "..." << std::endl;
61 for (int i = N - 10; i < N; i++)
62     std::cout << test1[i] << " " << test2[i] << std::endl;
63
64 // Cleanup
65 sycl::free(test1, Q.get_context());
66 sycl::free(test2, Q.get_context());
67 }

```

The necessary oneDPL and oneMKL functions are included in `<oneapi/dpl/random>` and `<oneapi/mkl/rng.hpp>`, respectively. The oneDPL and oneMKL examples perform the same sequence of operations: get a random number seed from the clock, initialize a random number engine, select the desired random number distribution, then generate the random numbers. The oneDPL code performs device offload explicitly using a DPC++ kernel. In the oneMKL code, the `mkl::rng` functions handle the device offload implicitly.

## 6.9 Efficiently Implementing Fourier Correlation Using oneAPI Math Kernel Library (oneMKL)

Now that straightforward use of oneMKL kernel functions has been covered, let's look at a more complex mathematical operation: cross-correlation. Cross-correlation has many applications, e.g.: measuring the similarity of two 1D signals, finding the best translation to overlay similar images, volumetric medical image segmentation, etc.

Consider the following simple signals, represented as vectors of ones and zeros:

Signal 1:	0	0	0	0	0	1	1	0
Signal 2:	0	0	1	1	0	0	0	0

The signals are treated as circularly shifted versions of each other, so shifting the second signal three elements relative to the first signal will give the maximum correlation score of two:

Signal 1:	0	0	0	0	0	1	1	0
Signal 2:				0	0	1	1	0
Correlation:	(1 * 1) + (1 * 1) = 2							

Shifts of two or four elements give a correlation score of one. Any other shift gives a correlation score of zero.

This is computed as follows:

$$corr_i = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} sig1_j \times sig2_{i+j}$$

where  $N$  is the number of elements in the signal vectors and  $i$  is the shift of  $sig2$  relative to  $sig1$ .

Real signals contain more data (and noise) but the principle is the same whether you are aligning 1D signals, overlaying 2D images, or performing 3D volumetric image registration. The goal is to find the translation that maximizes correlation. However, the brute force summation shown above requires  $N$  multiplications and additions for every  $N$  shifts. In 1D, 2D and 3D, the problem is  $O(N^2)$ ,  $O(N^3)$ , and  $O(N^4)$ , respectively.

The Fourier correlation algorithm is a much more efficient way to perform this computation because it takes advantage of the  $O(N \log N)$  of the Fourier transform:

```
corr = IDFT(DFT(sig1) * CONJG(DFT(sig2)))
```

where DFT is the discrete Fourier transform, IDFT is the inverse DFT, and CONJG is the complex conjugate. The Fourier correlation algorithm can be composed using oneMKL, which contains optimized forward and backward transforms and complex conjugate multiplication functions. Therefore, the entire computation can be performed on the accelerator device.

In many applications, only the final correlation result matters, so this is all that has to be transferred from the device back to the host. In this example, two artificial signals will be created on the device, transformed in-place, then correlated. The host will retrieve the final result and report the optimal translation and correlation score. Conventional wisdom suggests that buffering would give the best performance because it provides explicit control over data movement between the host and the device.

To test this hypothesis, let's generate two input signals:

**Listing 55:** /examples/libraries-fcorr/fcorr\_1d\_buffers.cpp

```
1 // Create buffers for signal data. This will only be used on the device.
2 sycl::buffer<float> sig1_buf{N + 2};
3 sycl::buffer<float> sig2_buf{N + 2};
4
5 // Declare container to hold the correlation result (computed on the device,
6 // used on the host)
7 std::vector<float> corr(N + 2);
```

Random noise is often added to signals to prevent overfitting during neural network training, to add visual effects to images, or to improve the detectability of signals obtained from suboptimal detectors, etc. The buffers are initialized with random noise using a simple random number generator in oneMKL:

**Listing 56:** /examples/libraries-fcorr/fcorr\_1d\_buffers.cpp

```
1 // Open new scope to trigger update of correlation result
2 {
3     sycl::buffer<float> corr_buf(corr);
4
5     // Initialize the input signals with artificial data
```

(continues on next page)

(continued from previous page)

```

6   std::uint32_t seed = (unsigned)time(NULL); // Get RNG seed value
7   oneapi::mkl::rng::mcg31m1 engine(0, seed); // Initialize RNG engine
8   // Set RNG distribution
9   oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
10    rng_distribution(-0.00005, 0.00005);
11
12   oneapi::mkl::rng::generate(rng_distribution, engine, N, sig1_buf); // Noise
13   oneapi::mkl::rng::generate(rng_distribution, engine, N, sig2_buf);

```

Notice that a new scope is opened and a buffer, `corr_buf`, is declared for the correlation result. When this buffer goes out of scope, `corr` will be updated on the host.

An artificial signal is placed at opposite ends of each buffer, similar to the trivial example above:

**Listing 57:** /examples/libraries-fcorr/fcorr\_1d\_buffers.cpp

```

1 Q.submit([&](sycl::handler &h) {
2     sycl::accessor sig1_acc{sig1_buf, h, sycl::write_only};
3     sycl::accessor sig2_acc{sig2_buf, h, sycl::write_only};
4     h.single_task<>([=](){
5         sig1_acc[N - N / 4 - 1] = 1.0;
6         sig1_acc[N - N / 4] = 1.0;
7         sig1_acc[N - N / 4 + 1] = 1.0; // Signal
8         sig2_acc[N / 4 - 1] = 1.0;
9         sig2_acc[N / 4] = 1.0;
10        sig2_acc[N / 4 + 1] = 1.0;
11    });
12}); // End signal initialization

```

Now that the signals are ready, let's transform them using the DFT functions in oneMKL:

**Listing 58:** /examples/libraries-fcorr/fcorr\_1d\_buffers.cpp

```

1 // Initialize FFT descriptor
2 oneapi::mkl::dft::descriptor<oneapi::mkl::dft::precision::SINGLE,
3                             oneapi::mkl::dft::domain::REAL>
4     transform_plan(N);
5 transform_plan.commit(Q);
6
7 // Perform forward transforms on real arrays
8 oneapi::mkl::dft::compute_forward(transform_plan, sig1_buf);
9 oneapi::mkl::dft::compute_forward(transform_plan, sig2_buf);

```

A single-precision, real-to-complex forward transform is committed to the SYCL queue, then an in-place DFT is performed on the data in both buffers. The result of  $DFT(sig1)$  must now be multiplied by  $CONJG(DFT(sig2))$ . This could be done with a hand-coded kernel:

```

Q.submit([&](sycl::handler &h)
{
    sycl::accessor sig1_acc{sig1_buf, h, sycl::read_only};
    sycl::accessor sig2_acc{sig2_buf, h, sycl::read_only};

```

(continues on next page)

(continued from previous page)

```

sycl::accessor corr_acc{corr_buf, h, sycl::write_only};

h.parallel_for<>(sycl::range<1>{N/2}, [=](auto idx)
{
    corr_acc[idx*2+0] = sig1_acc[idx*2+0] * sig2_acc[idx*2+0] +
        sig1_acc[idx*2+1] * sig2_acc[idx*2+1];
    corr_acc[idx*2+1] = sig1_acc[idx*2+1] * sig2_acc[idx*2+0] -
        sig1_acc[idx*2+0] * sig2_acc[idx*2+1];
});
});

```

However, this basic implementation is unlikely to give optimal cross-architecture performance. Fortunately, the oneMKL function, `oneapi::mkl::vm::mulbyconj`, can be used for this step. The `mulbyconj` function expects `std::complex<float>` input, but the buffers were initialized as the `float` data type. Even though they contain complex data after the forward transform, the buffers will have to be recast:

**Listing 59:** /examples/libraries-fcorr/fcorr\_1d\_buffers.cpp

```

1 auto sig1_buf_cplx =
2     sig1_buf.template reinterpret<std::complex<float>, 1>((N + 2) / 2);
3 auto sig2_buf_cplx =
4     sig2_buf.template reinterpret<std::complex<float>, 1>((N + 2) / 2);
5 auto corr_buf_cplx =
6     corr_buf.template reinterpret<std::complex<float>, 1>((N + 2) / 2);
7 oneapi::mkl::vm::mulbyconj(Q, N / 2, sig1_buf_cplx, sig2_buf_cplx,
8                             corr_buf_cplx);

```

The IDFT step completes the computation:

**Listing 60:** /examples/libraries-fcorr/fcorr\_1d\_buffers.cpp

```

1 // Perform backward transform on complex correlation array
2 oneapi::mkl::dft::compute_backward(transform_plan, corr_buf);

```

When the scope that was opened at the start of this example is closed, the buffer holding the correlation result goes out of scope, forcing an update of the host container. This is the only data transfer between the host and the device.

The complete Fourier correlation implementation using explicit buffering is included below:

**Listing 61:** /examples/libraries-fcorr/fcorr\_1d\_buffers.cpp

```

1 #include <CL/sycl.hpp>
2 #include <iostream>
3 #include <mkl.h>
4 #include <oneapi/mkl/dfti.hpp>
5 #include <oneapi/mkl/rng.hpp>
6 #include <oneapi/mkl/vm.hpp>
7
8 int main(int argc, char **argv) {
9     unsigned int N = (argc == 1) ? 32 : std::stoi(argv[1]);

```

(continues on next page)

(continued from previous page)

```
10 if ((N % 2) != 0)
11     N++;
12 if (N < 32)
13     N = 32;
14
15 // Initialize SYCL queue
16 sycl::queue Q(sycl::default_selector{});
17 std::cout << "Running on: "
18     << Q.get_device().get_info<sycl::info::device::name>() << std::endl;
19
20 // Create buffers for signal data. This will only be used on the device.
21 sycl::buffer<float> sig1_buf{N + 2};
22 sycl::buffer<float> sig2_buf{N + 2};
23
24 // Declare container to hold the correlation result (computed on the device,
25 // used on the host)
26 std::vector<float> corr(N + 2);
27
28 // Open new scope to trigger update of correlation result
29 {
30     sycl::buffer<float> corr_buf(corr);
31
32     // Initialize the input signals with artificial data
33     std::uint32_t seed = (unsigned)time(NULL); // Get RNG seed value
34     oneapi::mkl::rng::mcg31m1 engine(Q, seed); // Initialize RNG engine
35                                         // Set RNG distribution
36     oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
37     rng_distribution(-0.00005, 0.00005);
38
39     oneapi::mkl::rng::generate(rng_distribution, engine, N, sig1_buf); // Noise
40     oneapi::mkl::rng::generate(rng_distribution, engine, N, sig2_buf);
41
42     Q.submit([&](sycl::handler &h) {
43         sycl::accessor sig1_acc{sig1_buf, h, sycl::write_only};
44         sycl::accessor sig2_acc{sig2_buf, h, sycl::write_only};
45         h.single_task<>([=]() {
46             sig1_acc[N - N / 4 - 1] = 1.0;
47             sig1_acc[N - N / 4] = 1.0;
48             sig1_acc[N - N / 4 + 1] = 1.0; // Signal
49             sig2_acc[N / 4 - 1] = 1.0;
50             sig2_acc[N / 4] = 1.0;
51             sig2_acc[N / 4 + 1] = 1.0;
52         });
53     }); // End signal initialization
54
55     clock_t start_time = clock(); // Start timer
56
57     // Initialize FFT descriptor
58     oneapi::mkl::dft::descriptor<oneapi::mkl::dft::precision::SINGLE,
59                                         oneapi::mkl::dft::domain::REAL>
60     transform_plan(N);
```

(continues on next page)

(continued from previous page)

```

61 transform_plan.commit(Q);
62
63 // Perform forward transforms on real arrays
64 oneapi::mkl::dft::compute_forward(transform_plan, sig1_buf);
65 oneapi::mkl::dft::compute_forward(transform_plan, sig2_buf);
66
67 // Compute: DFT(sig1) * CONJG(DFT(sig2))
68 auto sig1_buf_cplx =
69     sig1_buf.template reinterpret<std::complex<float>, 1>((N + 2) / 2);
70 auto sig2_buf_cplx =
71     sig2_buf.template reinterpret<std::complex<float>, 1>((N + 2) / 2);
72 auto corr_buf_cplx =
73     corr_buf.template reinterpret<std::complex<float>, 1>((N + 2) / 2);
74 oneapi::mkl::vm::mulbyconj(Q, N / 2, sig1_buf_cplx, sig2_buf_cplx,
75                             corr_buf_cplx);
76
77 // Perform backward transform on complex correlation array
78 oneapi::mkl::dft::compute_backward(transform_plan, corr_buf);
79
80 clock_t end_time = clock(); // Stop timer
81 std::cout << "The 1D correlation (N = " << N << ") took "
82             << float(end_time - start_time) / CLOCKS_PER_SEC << " seconds."
83             << std::endl;
84
85 } // Buffer holding correlation result is now out of scope, forcing update of
86 // host container
87
88 // Find the shift that gives maximum correlation value
89 float max_corr = 0.0;
90 int shift = 0;
91 for (unsigned int idx = 0; idx < N; idx++) {
92     if (corr[idx] > max_corr) {
93         max_corr = corr[idx];
94         shift = idx;
95     }
96 }
97 shift =
98     (shift > N / 2) ? shift - N : shift; // Treat the signals as circularly
99                                         // shifted versions of each other.
100 std::cout << "Shift the second signal " << shift
101             << " elements relative to the first signal to get a maximum, "
102             << "normalized correlation score of "
103             << max_corr / N << "." << std::endl;
104 }
```

The Fourier correlation algorithm will now be reimplemented using Unified Shared Memory (USM) to compare to explicit buffering. Only the differences in the two implementations will be highlighted. First, the signal and correlation arrays are allocated in USM, then initialized with artificial data:

**Listing 62:** /examples/libraries-fcorr/fcorr\_ld\_usm.cpp

```

1 // Initialize signal and correlation arrays
2 auto sig1 = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
3 auto sig2 = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
4 auto corr = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
5
6 // Initialize input signals with artificial data
7 std::uint32_t seed = (unsigned)time(NULL); // Get RNG seed value
8 oneapi::mkl::rng::mcg31m1 engine(Q, seed); // Initialize RNG engine
9 // Set RNG distribution
10 oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
11     rng_distribution(-0.00005, 0.00005);
12
13 // Warning: These statements run on the device.
14 auto evt1 =
15     oneapi::mkl::rng::generate(rng_distribution, engine, N, sig1); // Noise
16 auto evt2 = oneapi::mkl::rng::generate(rng_distribution, engine, N, sig2);
17 evt1.wait();
18 evt2.wait();
19
20 // Warning: These statements run on the host, so sig1 and sig2 will have to be
21 // updated on the device.
22 sig1[N - N / 4 - 1] = 1.0;
23 sig1[N - N / 4] = 1.0;
24 sig1[N - N / 4 + 1] = 1.0; // Signal
25 sig2[N / 4 - 1] = 1.0;
26 sig2[N / 4] = 1.0;
27 sig2[N / 4 + 1] = 1.0;

```

The rest of the implementation is largely the same except pointers to USM are passed to the oneMKL functions instead of SYCL buffers:

**Listing 63:** /examples/libraries-fcorr/fcorr\_ld\_usm.cpp

```

1 // Perform forward transforms on real arrays
2 evt1 = oneapi::mkl::dft::compute_forward(transform_plan, sig1);
3 evt2 = oneapi::mkl::dft::compute_forward(transform_plan, sig2);
4
5 // Compute: DFT(sig1) * CONJG(DFT(sig2))
6 oneapi::mkl::vm::mulbyconj(
7     Q, N / 2, reinterpret_cast<std::complex<float> *>(sig1),
8     reinterpret_cast<std::complex<float> *>(sig2),
9     reinterpret_cast<std::complex<float> *>(corr), {evt1, evt2})
10    .wait();
11
12 // Perform backward transform on complex correlation array
13 oneapi::mkl::dft::compute_backward(transform_plan, corr).wait();

```

It is also necessary to free the allocated memory:

**Listing 64:** /examples/libraries-fcorr/fcorr\_1d\_usm.cpp

```

1  sycl::free(sig1, sycl_context);
2  sycl::free(sig2, sycl_context);
3  sycl::free(corr, sycl_context);

```

The USM implementation has a more familiar syntax. It is also conceptually simpler because it relies on implicit data transfer handled by the DPC++ runtime. However, a programmer error hurts performance.

Notice the warning messages in the previous code snippets. The oneMKL random number generation engine is initialized on the device, so `sig1` and `sig2` are initialized with random noise on the device. Unfortunately, the code adding the artificial signal runs on the host, so the DPC++ runtime has to make the host and device data consistent. The signals used in Fourier correlation are usually large, especially in 3D imaging applications, so unnecessary data transfer degrades performance.

Updating the signal data directly on the device keeps the data consistent, thereby avoiding the unnecessary data transfer:

**Listing 65:** /examples/libraries-fcorr/fcorr\_1d\_usm\_fixed.cpp

```

1 Q.single_task<>([=]() {
2     sig1[N - N / 4 - 1] = 1.0;
3     sig1[N - N / 4] = 1.0;
4     sig1[N - N / 4 + 1] = 1.0; // Signal
5     sig2[N / 4 - 1] = 1.0;
6     sig2[N / 4] = 1.0;
7     sig2[N / 4 + 1] = 1.0;
8 }).wait();

```

The explicit buffering and USM implementations now have equivalent performance, indicating that the DPC++ runtime is good at avoiding unnecessary data transfers (provided the programmer pays attention to data consistency).

The complete Fourier correlation implementation in USM is included below:

**Listing 66:** /examples/libraries-fcorr/fcorr\_1d\_usm\_fixed.cpp

```

1 #include <CL/sycl.hpp>
2 #include <iostream>
3 #include <mkl.h>
4 #include <oneapi/mkl/dfti.hpp>
5 #include <oneapi/mkl/rng.hpp>
6 #include <oneapi/mkl/vm.hpp>
7
8 int main(int argc, char **argv) {
9     unsigned int N = (argc == 1) ? 32 : std::stoi(argv[1]);
10    if ((N % 2) != 0)
11        N++;
12    if (N < 32)
13        N = 32;
14

```

(continues on next page)

(continued from previous page)

```
15 // Initialize SYCL queue
16 sycl::queue Q(sycl::default_selector{});
17 auto sycl_device = Q.get_device();
18 auto sycl_context = Q.get_context();
19 std::cout << "Running on: "
20         << Q.get_device().get_info<sycl::info::device::name>() << std::endl;
21
22 // Initialize signal and correlation arrays
23 auto sig1 = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
24 auto sig2 = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
25 auto corr = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
26
27 // Initialize input signals with artificial data
28 std::uint32_t seed = (unsigned)time(NULL); // Get RNG seed value
29 oneapi::mkl::rng::mcg31ml engine(Q, seed); // Initialize RNG engine
30                                         // Set RNG distribution
31 oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
32     rng_distribution(-0.00005, 0.00005);
33
34 auto evt1 =
35     oneapi::mkl::rng::generate(rng_distribution, engine, N, sig1); // Noise
36 auto evt2 = oneapi::mkl::rng::generate(rng_distribution, engine, N, sig2);
37 evt1.wait();
38 evt2.wait();
39
40 Q.single_task<>([=]() {
41     sig1[N - N / 4 - 1] = 1.0;
42     sig1[N - N / 4] = 1.0;
43     sig1[N - N / 4 + 1] = 1.0; // Signal
44     sig2[N / 4 - 1] = 1.0;
45     sig2[N / 4] = 1.0;
46     sig2[N / 4 + 1] = 1.0;
47 }).wait();
48
49 clock_t start_time = clock(); // Start timer
50
51 // Initialize FFT descriptor
52 oneapi::mkl::dft::descriptor<oneapi::mkl::dft::precision::SINGLE,
53                             oneapi::mkl::dft::domain::REAL>
54     transform_plan(N);
55 transform_plan.commit(Q);
56
57 // Perform forward transforms on real arrays
58 evt1 = oneapi::mkl::dft::compute_forward(transform_plan, sig1);
59 evt2 = oneapi::mkl::dft::compute_forward(transform_plan, sig2);
60
61 // Compute: DFT(sig1) * CONJG(DFT(sig2))
62 oneapi::mkl::vm::mulbyconj(
63     Q, N / 2, reinterpret_cast<std::complex<float> *>(sig1),
64     reinterpret_cast<std::complex<float> *>(sig2),
65     reinterpret_cast<std::complex<float> *>(corr), {evt1, evt2})
```

(continues on next page)

(continued from previous page)

```

66     .wait();
67
68 // Perform backward transform on complex correlation array
69 oneapi::mkl::dft::compute_backward(transform_plan, corr).wait();
70
71 clock_t end_time = clock(); // Stop timer
72 std::cout << "The 1D correlation (N = " << N << ") took "
73     << float(end_time - start_time) / CLOCKS_PER_SEC << " seconds."
74     << std::endl;
75
76 // Find the shift that gives maximum correlation value
77 float max_corr = 0.0;
78 int shift = 0;
79 for (unsigned int idx = 0; idx < N; idx++) {
80     if (corr[idx] > max_corr) {
81         max_corr = corr[idx];
82         shift = idx;
83     }
84 }
85 shift =
86     (shift > N / 2) ? shift - N : shift; // Treat the signals as circularly
87                                         // shifted versions of each other.
88 std::cout << "Shift the second signal " << shift
89     << " elements relative to the first signal to get a maximum, "
90     << "normalized correlation score of "
91     << max_corr / N << "." << std::endl;
92
93 // Cleanup
94 sycl::free(sig1, sycl_context);
95 sycl::free(sig2, sycl_context);
96 sycl::free(corr, sycl_context);
97 }
```

Note that the final step of finding the location of the maximum correlation value is performed on the host. It would be better to do this computation on the device, especially when the input data is large. Fortunately, the maxloc reduction is a common parallel pattern that can be implemented using DPC++. This is left as an exercise for the reader, but Figure 14-11 of [Data Parallel C++](#) provides a suitable example to help you get started.

## 6.10 Executing Multiple Kernels on the Device at the Same Time

DPC++ has two kinds of queues that a programmer can create and submit kernels for execution.

**in-order queues** where kernels are executed in the order they were submitted to the queue

**out-of-order queues** where kernels can be executed in an arbitrary order (subject to the dependency constraints among them).

The choice to create an in-order or out-of-order queue is at queue construction time through the property `sycl::property::queue::in_order()`. By default, when no property is specified the queue is out-of-order.

In the following example, three kernels are submitted per iteration. Each of these kernels uses only one work-group with 256 work-items. These kernels are created specifically with one group to ensure that they do not use the entire machine. This is done to illustrate the benefit of parallel kernel execution.

**Listing 67:** /examples/multiple-kernel-execution/kernels.cpp

```

1 int multi_queue(sycl::queue &q, const IntArray &a, const IntArray &b) {
2     size_t num_items = a.size();
3     IntArray s1, s2, s3;
4
5     sycl::buffer a_buf(a);
6     sycl::buffer b_buf(b);
7     sycl::buffer sum_buf1(s1);
8     sycl::buffer sum_buf2(s2);
9     sycl::buffer sum_buf3(s3);
10
11    size_t num_groups = 1;
12    size_t wg_size = 256;
13    auto start = std::chrono::steady_clock::now();
14    for (int i = 0; i < iter; i++) {
15        q.submit([&](sycl::handler &h) {
16            sycl::accessor a_acc(a_buf, h, sycl::read_only);
17            sycl::accessor b_acc(b_buf, h, sycl::read_only);
18            sycl::accessor sum_acc(sum_buf1, h, sycl::write_only, sycl::no_init);
19
20            h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
21                           [=](sycl::nd_item<1> index) {
22                               size_t loc_id = index.get_local_id();
23                               sum_acc[loc_id] = 0;
24                               for (int j = 0; j < 1000; j++)
25                                   for (size_t i = loc_id; i < array_size; i += wg_size) {
26                                       sum_acc[loc_id] += a_acc[i] + b_acc[i];
27                                   }
28                           });
29        });
30        q.submit([&](sycl::handler &h) {
31            sycl::accessor a_acc(a_buf, h, sycl::read_only);
32            sycl::accessor b_acc(b_buf, h, sycl::read_only);
33            sycl::accessor sum_acc(sum_buf2, h, sycl::write_only, sycl::no_init);
34
35            h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
36                           [=](sycl::nd_item<1> index) {
37                               size_t loc_id = index.get_local_id();
38                               sum_acc[loc_id] = 0;
39                               for (int j = 0; j < 1000; j++)
40                                   for (size_t i = loc_id; i < array_size; i += wg_size) {
41                                       sum_acc[loc_id] += a_acc[i] + b_acc[i];
42                                   }
43                           });
44        });
45        q.submit([&](sycl::handler &h) {
46            sycl::accessor a_acc(a_buf, h, sycl::read_only);
47            sycl::accessor b_acc(b_buf, h, sycl::read_only);

```

(continues on next page)

(continued from previous page)

```

48     sycl::accessor sum_acc(sum_buf3, h, sycl::write_only, sycl::no_init);
49
50     h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
51                     [=](sycl::nd_item<1> index) {
52                         size_t loc_id = index.get_local_id();
53                         sum_acc[loc_id] = 0;
54                         for (int j = 0; j < 1000; j++)
55                             for (size_t i = loc_id; i < array_size; i += wg_size) {
56                                 sum_acc[loc_id] += a_acc[i] + b_acc[i];
57                             }
58                     });
59     });
60     q.wait();
61     auto end = std::chrono::steady_clock::now();
62     std::cout << "multi_queue completed on device - took "
63                     << (end - start).count() << " u-secs\n";
64     // check results
65     return ((end - start).count());
66 } // end multi_queue

```

In the case where the underlying queue is in-order, these kernels cannot be executed in parallel and have to be executed sequentially even though there are adequate resources in the machine and there are no dependencies among the kernels. This can be seen from the larger total execution time for all the kernels. The creation of the queue and the kernel submission is shown below.

**Listing 68:** /examples/multiple-kernel-execution/kernels.cpp

```

1  sycl::property_list q_prop{sycl::property::queue::in_order()};
2  std::cout << "In order queue: Jitting+Execution time\n";
3  sycl::queue q1(d_selector, q_prop);
4  multi_queue(q1, a, b);
5  usleep(500 * 1000);
6  std::cout << "In order queue: Execution time\n";
7  multi_queue(q1, a, b);

```

When the queue is out-of-order, the overall execution time is much lower, indicating that the machine is able to execute different kernels from the queue at the same time. The creation of the queue and the invocation of the kernel is shown below.

**Listing 69:** /examples/multiple-kernel-execution/kernels.cpp

```

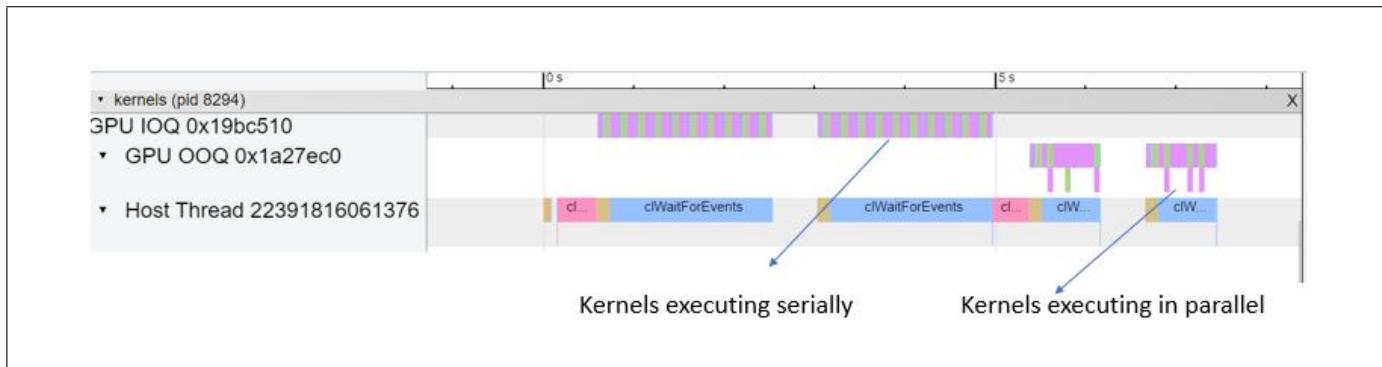
1  sycl::queue q2(d_selector);
2  std::cout << "Out of order queue: Jitting+Execution time\n";
3  multi_queue(q2, a, b);
4  usleep(500 * 1000);
5  std::cout << "Out of order queue: Execution time\n";
6  multi_queue(q2, a, b);

```

In situations where kernels do not scale strongly and therefore cannot effectively utilize full machine compute resources, it is better to allocate only the required compute units through appropriate selection of work-

group/work-item values and try to execute multiple kernels at the same time.

The following timeline view shows the kernels being executed by in-order and out-of-order queues (this was collected using the `clIntercept` tool when the binary is run on the OpenCL back-end). Here one can clearly see that kernels submitted to the out-of-order queue are being executed in parallel. Another thing to notice is that not all three kernels are executed in parallel. The reason for this is a heuristic the driver uses in deciding about the kernel submissions. When the driver sees the GPU is not active, it immediately submits the kernel for execution and then buffers the kernels in the queue.



**Fig. 13:** Timeline for kernels executed with in-order and out-of-order queues

It is also possible to statically partition a single device into sub-devices through the use of `create_sub_devices` function of `device` class. This provides more control to the programmer for submitting kernels to an appropriate sub-device. However, the partition of a device into sub-devices is static, so the runtime will not be able to adapt to the dynamic load of an application because it does not have flexibility to move kernels from one sub-device to another.

At the time of writing, only the OpenCL backend is able to execute the kernels out of order. Support in the Level Zero backend to execute kernels out of order is still in development.

## 6.11 Synchronization among Threads in a Kernel

There are a variety of ways in which the work-items in a kernel can synchronize to exchange data, update data, or cooperate with each other to accomplish a task in a specific order. These are:

**Accessor classes** Accessor classes specify acquisition and release of buffer and image data structures. Depending on where they are created and destroyed, the runtime generates appropriate data transfers and synchronization primitives.

**Atomic operations** DPC++ devices support a restricted subset of C++ atomics.

**Fences** Fence primitives are used to order loads and stores. Fences can have acquire semantics, release semantics, or both.

**Barriers** Barriers are used to synchronize sets of work-items within individual groups.

**Hierarchical parallel dispatch** In the hierarchical parallelism model of describing computations, synchronization within the work-group is made explicit through multiple instances of the `parallel_for_work_item` function call, rather than through the use of explicit work-group barrier operations.

**Device event** Events are used inside kernel functions to wait for asynchronous operations to complete.

In many cases, any of the above synchronization events can be used to achieve the same functionality but with significant differences in efficiency and performance.

### 6.11.1 Data types for Atomic Operations

Atomics allow multiple work-items for any cross work-item communication via memory. DPC++ atomics are similar to C++ atomics and make the access to resources protected by atomics guaranteed to be executed as a single unit. The following kernel shows the implementation of a reduction operation in DPC++ where every work-item is updating a global accumulator atomically. The input data type of this addition and the vector on which this reduction operation is being applied is an integer. The performance of this kernel is reasonable as compared to other techniques like blocking used for reduction.

**Listing 70:** /examples/atomics/atomics.cpp

```

1 q.submit([&](auto &h) {
2     sycl::accessor buf_acc(buf, h, sycl::read_only);
3     sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
4
5     h.parallel_for(data_size, [=](auto index) {
6         size_t glob_id = index[0];
7         auto v = sycl::ext::oneapi::atomic_ref<
8             int, sycl::ext::oneapi::memory_order::relaxed,
9             sycl::ext::oneapi::memory_scope::device,
10            sycl::access::address_space::global_space>(sum_acc[0]);
11         v.fetch_add(buf_acc[glob_id]);
12     });
13 });

```

If the data type of the vector is a float or a double as shown in the kernel below then the performance on certain accelerators is not good due to lack of hardware support for float or double atomics. The following two kernels demonstrates how that time to execute an atomic add can vary drastically based on whether native atomic is supported.

**Listing 71:** /examples/atomics/test\_atomic.cpp

```

1 // 
2 int VectorInt(sycl::queue &q, int iter) {
3     VectorAllocator<int> alloc;
4     AlignedVector<int> a(array_size, alloc);
5     AlignedVector<int> b(array_size, alloc);
6
7     InitializeArray<int>(a);
8     InitializeArray<int>(b);
9     sycl::range num_items{a.size()};
10    sycl::buffer a_buf(a);
11    sycl::buffer b_buf(b);
12    auto start = std::chrono::steady_clock::now();
13    for (int i = 0; i < iter; i++) {
14        q.submit([&](sycl::handler &h) {

```

(continues on next page)

(continued from previous page)

```

15 // InputGt accessors
16 sycl::accessor a_acc(a_buf, h, sycl::read_write);
17 sycl::accessor b_acc(a_buf, h, sycl::read_only);
18
19 h.parallel_for(num_items, [=](auto i) {
20     auto v = sycl::ext::oneapi::atomic_ref<
21         int, sycl::ext::oneapi::memory_order::relaxed,
22         sycl::ext::oneapi::memory_scope::device,
23         sycl::access::address_space::global_space>(a_acc[0]);
24     v += b_acc[i];
25 });
26 });
27 }
28 q.wait();
29 auto end = std::chrono::steady_clock::now();
30 std::cout << "Vector int completed on device - took " << (end - start).count()
31     << " u-secs\n";
32 return ((end - start).count());
33 }
```

When using atomics care must be taken to ensure that there is support in the hardware and that they can be executed efficiently. In Gen9 and Intel® Iris® Xe integrated graphics there is no support for atomics on float or double data types and the performance of VectorDouble will be very poor. In future GPUs where the float and double atomics are supported in hardware the performance of the above kernel will be much better.

**Listing 72:** /examples/atomics/test\_atomic.cpp

```

1 //
2 int VectorDouble(sycl::queue &q, int iter) {
3     VectorAllocator<double> alloc;
4     AlignedVector<double> a(array_size, alloc);
5     AlignedVector<double> b(array_size, alloc);
6
7     InitializeArray<double>(a);
8     InitializeArray<double>(b);
9     sycl::range num_items{a.size()};
10    sycl::buffer a_buf(a);
11    sycl::buffer b_buf(b);
12
13    auto start = std::chrono::steady_clock::now();
14    for (int i = 0; i < iter; i++) {
15        q.submit([&](sycl::handler &h) {
16            // InputGt accessors
17            sycl::accessor a_acc(a_buf, h, sycl::read_write);
18            sycl::accessor b_acc(a_buf, h, sycl::read_only);
19
20            h.parallel_for(num_items, [=](auto i) {
21                auto v = sycl::ext::oneapi::atomic_ref<
22                    double, sycl::ext::oneapi::memory_order::relaxed,
23                    sycl::ext::oneapi::memory_scope::device,
24                    sycl::access::address_space::global_space>(a_acc[0]);
25            });
26        });
27    }
28 }
```

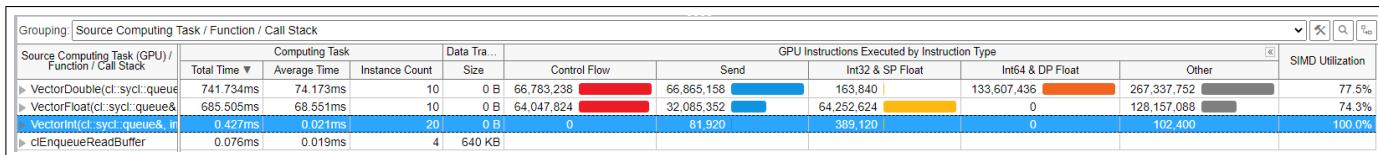
(continues on next page)

(continued from previous page)

```

25         v += b_acc[i];
26     });
27 });
28 }
29 q.wait();
30 auto end = std::chrono::steady_clock::now();
31 std::cout << "Vector Double completed on device - took "
32             << (end - start).count() << " u-secs\n";
33 return ((end - start).count());
34 }
```

By analyzing these kernels using VTune Profiler we can measure the impact of native atomic support. You can see that the VectorInt kernel is much faster than VectorDouble and VectorFloat.



**Fig. 14:** VTune dynamic instruction

VTune Profiler dynamic instruction analysis allows us to see the instruction counts vary dramatically when there is no support for native atomic.

Here is the assembly code for our VectorInt kernel.

Addr...	So...	Assembly	GPU Instructions Executed
0		Block 1:	
0	53	(W) mov (8 M0) r3.0<1>:ud r0.0<1;1,0>:ud	20,480
0x10	53	(W) or (1 M0) cr0.0<1>:ud cr0.0<0;1,0>:ud 0x4C0:uw {Swi}	20,480
0x20		(W) mul (1 M0) r10.0<1>:d r11.0<0;1,0>:d r3.1<0;1,0>:d	20,480
0x28		(W) mov (16 M0) r11.0<1>:d 0:w	20,480
0x38		(W) shl (1 M0) r10.2<1>:d r9.6<0;1,0>:d 2:w	20,480
0x48		(W) mov (16 M16) r4.0<1>:d 0:w	20,480
0x58		(W) shl (1 M0) r10.1<1>:d r9.0<0;1,0>:d 2:w	20,480
0x68		add (16 M0) r8.0<1>:d r10.0<0;1,0>:d r1.0<16;16,1>:uw	20,480
0x78		add (16 M16) r4.0<1>:d r10.0<0;1,0>:d r2.0<16;16,1>:uw	20,480
0x88		(W) add (1 M0) r10.0<1>:d r10.2<0;1,0>:d r10.5<0;1,0>:d	20,480
0x98		(W) mov (8 M0) r112.0<1>:ud r3.0<8;8,1>:ud {Compacted}	20,480
0xa0		(W) add (1 M0) r14.0<1>:d r10.1<0;1,0>:d r10.4<0;1,0>:d	20,480
0xb0		add (16 M0) r8.0<1>:d r8.0<8;8,1>:d r7.0<0;1,0>:d {Compacted}	20,480
0xb8		add (16 M16) r4.0<1>:d r4.0<8;8,1>:d r7.0<0;1,0>:d	20,480
0xc8		shl (16 M0) r8.0<1>:d r8.0<8;8,1>:d 2:w	20,480
0xd8		shl (16 M16) r4.0<1>:d r4.0<8;8,1>:d 2:w	20,480
0xe8		add (16 M0) r8.0<1>:d r10.0<0;1,0>:d r8.0<8;8,1>:d {Compacted}	20,480
0xf0		add (16 M16) r4.0<1>:d r10.0<0;1,0>:d r4.0<8;8,1>:d	20,480
0x100		send (16 M0) r6:w r8 0xC 0x04205E01 [Data Cache Data Po]	20,480
0x110		send (16 M16) r4:w r4 0xC 0x04205E01 [Data Cache Data Po]	20,480
0x120		mov (16 M0) r11.0<1>:d r6.0<8;8,1>:d {Compacted}	20,480
0x128		(W) add (16 M0) r4.0<1>:d r11.0<8;8,1>:d r4.0<8;8,1>:d	20,480
0x130		(W) add (8 M0) r2.0<1>:d r4.0<8;8,1>:d r5.0<8;8,1>:d {Compacted}	20,480
0x138		(W) add (4 M0) r2.0<1>:d r2.0<4;4,1>:d r2.4<4;4,1>:d {Compacted}	20,480
0x140		(W) add (2 M0) r2.0<1>:d r2.0<2;2,1>:d r2.2<2;2,1>:d	20,480
0x150		(W) add (1 M0) r2.0<1>:d r2.0<0;1,0>:d r2.1<0;1,0>:d {Compacted}	20,480
0x158		(W) sends (1 M0) null:ud r14 r2 0x4C 0x02009700 [Data Cache Data Po]	20,480
0x168		(W) send (8 M0) null r112 0x27 0x02000010 {EOT} [Thread]	20,480

**Fig. 15:** VTune atomic int

Compared to the assembly code for VectorDouble. There are 33 million more GPU instructions required when we execute our VectorDouble kernel.

Addr...	Sour...	Assembly	GPU Instructions Executed
0x50		(W) <code>shl (1 M0) r6.0&lt;1&gt;:q r6.0&lt;0;1,0&gt;:q 3:w</code>	20,480
0x60		<code>mov (8 M8) r16.0&lt;1&gt;:q 0:w</code>	20,480
0x70		<code>add (16 M0) r8.0&lt;1&gt;:d r8.0&lt;0;1,0&gt;:d r1.0&lt;16;16,1&gt;:uw</code>	20,480
0x80		(W) <code>add (1 M0) r6.2&lt;1&gt;:d r6.2&lt;0;1,0&gt;:d r7.4&lt;0;1,0&gt;:d</code>	20,480
0x90		(W) <code>add (1 M0) r26.0&lt;1&gt;:q r6.0&lt;0;1,0&gt;:q r5.0&lt;0;1,0&gt;:q</code>	20,480
0xa0		<code>add (16 M0) r8.0&lt;1&gt;:d r8.0&lt;8;8,1&gt;:d r4.0&lt;0;1,0&gt;:d {Compacted}</code>	20,480
0xa8		<code>mov (8 M0) r10.0&lt;1&gt;:uq r26.0&lt;0;1,0&gt;:uq</code>	20,480
0xb8		<code>mov (8 M8) r12.0&lt;1&gt;:uq r26.0&lt;0;1,0&gt;:uq</code>	20,480
0xc8		<code>shl (16 M0) r8.0&lt;1&gt;:d r8.0&lt;8;8,1&gt;:d 3:w</code>	20,480
0xd8		<code>add (16 M0) r8.0&lt;1&gt;:d r6.2&lt;0;1,0&gt;:d r8.0&lt;8;8,1&gt;:d {Compacted}</code>	20,480
0xe0		<code>send (16 M0) r4:w r8 0xC 0x04405C01 [Data Cache Data Pd]</code>	20,480
0xf0		<code>sends (8 M0) r8:uq r10 r14 0x8C 0x0424B2FF [Data Cache]</code>	20,480
0x100		<code>sends (8 M8) r10:uq r12 r16 0x8C 0x0424B2FF [Data Cache]</code>	20,480
0x110		<code>mov (8 M0) r23.0&lt;2&gt;:d r4.0&lt;8;8,1&gt;:d</code>	20,480
0x120		<code>mov (8 M8) r21.0&lt;2&gt;:d r5.0&lt;8;8,1&gt;:d</code>	20,480
0x130		<code>mov (8 M0) r23.1&lt;2&gt;:d r6.0&lt;8;8,1&gt;:d</code>	20,480
0x140		<code>mov (8 M8) r21.1&lt;2&gt;:d r7.0&lt;8;8,1&gt;:d</code>	20,480
0x150		<code>mov (8 M0) r19.0&lt;1&gt;:uq r8.0&lt;4;4,1&gt;:uq</code>	20,480
0x160		<code>mov (8 M8) r17.0&lt;1&gt;:uq r10.0&lt;4;4,1&gt;:uq</code>	20,480
<b>0x170</b>		<b>Block 2:</b>	
0x170		<code>add (8 M0) r11.0&lt;1&gt;:df r23.0&lt;4;4,1&gt;:df r19.0&lt;4;4,1&gt;:df</code>	33,391,619
0x180		<code>mov (8 M0) r5.0&lt;1&gt;:uq r26.0&lt;0;1,0&gt;:uq</code>	33,391,619
0x190		<code>mov (8 M0) r9.0&lt;1&gt;:q r19.0&lt;4;4,1&gt;:q</code>	33,391,619
0x1a0		<code>add (8 M8) r15.0&lt;1&gt;:df r21.0&lt;4;4,1&gt;:df r17.0&lt;4;4,1&gt;:df</code>	33,391,619
0x1b0		<code>mov (8 M8) r7.0&lt;1&gt;:uq r26.0&lt;0;1,0&gt;:uq</code>	33,391,619
0x1c0		<code>mov (8 M8) r13.0&lt;1&gt;:q r17.0&lt;4;4,1&gt;:q</code>	33,391,619
0x1d0		<code>sends (8 M0) r3:uq r5 r9 0x10C 0x0424BEFF [Data Cache]</code>	33,391,619
0x1e0		<code>sends (8 M8) r5:uq r7 r13 0x10C 0x0424BEFF [Data Cache]</code>	33,391,619
0x1f0		<code>mov (8 M0) r7.0&lt;1&gt;:uq r3.0&lt;4;4,1&gt;:uq</code>	33,391,619
0x200		<code>mov (8 M8) r3.0&lt;1&gt;:uq r5.0&lt;4;4,1&gt;:uq</code>	33,391,619
0x210		<code>cmp (8 M0) (eq)f0.0 null&lt;1&gt;:q r7.0&lt;4;4,1&gt;:q r19.0&lt;4;4,1&gt;:q</code>	33,391,619
0x220		<code>cmp (8 M8) (eq)f0.0 null&lt;1&gt;:q r3.0&lt;4;4,1&gt;:q r17.0&lt;4;4,1&gt;:q</code>	33,391,619
0x230		<u><code>(f0.0) break (16 M0) bb_4 bb_4</code></u>	33,391,619
<b>0x240</b>		<b>Block 3:</b>	
0x240		<code>mov (8 M0) r19.0&lt;1&gt;:q r7.0&lt;4;4,1&gt;:q</code>	33,371,139
0x250		<code>mov (8 M8) r17.0&lt;1&gt;:q r3.0&lt;4;4,1&gt;:q</code>	33,371,139
<b>0x260</b>		<b>Block 4:</b>	
0x260		<u><code>while (16 M0) bb_2</code></u>	33,391,619
<b>0x270</b>		<b>Block 5:</b>	
0x270		(W) <code>mov (8 M0) r112.0&lt;1&gt;:ud r2.0&lt;8;8,1&gt;:ud {Compacted}</code>	20,480
0x278		(W) <code>s VTune Profiler session timed out. All open results were closed and a new session was created.</code>	

**Fig. 16:** VTune atomic double

The standard C++ memory model assumes that applications execute on a single device with a single address space. Neither of these assumptions holds for DPC++ applications: different parts of the application execute on different devices (i.e., a host device and one or more accelerator devices); each device has multiple address

spaces (i.e., private, local, and global); and the global address space of each device may or may not be disjoint (depending on USM support).

When using atomics in the global address space, again, care must be taken because global updates are much slower than local.

**Listing 73:** /examples/atomics/global\_atomics\_ref.cpp

```

1 #include <CL/sycl.hpp>
2 #include <iostream>
3 int main() {
4     constexpr int N = 256 * 256;
5     constexpr int M = 512;
6     int total = 0;
7     int *a = static_cast<int *>(malloc(sizeof(int) * N));
8     for (int i = 0; i < N; i++)
9         a[i] = 1;
10    sycl::queue q({sycl::property::queue::enable_profiling()});
11    sycl::buffer<int> buf(&total, 1);
12    sycl::buffer<int> bufa(a, N);
13    auto e = q.submit([&](sycl::handler &h) {
14        sycl::accessor acc(buf, h);
15        sycl::accessor acc_a(bufa, h, sycl::read_only);
16        h.parallel_for(sycl::nd_range<1>(N, M), [=](auto it) {
17            auto i = it.get_global_id();
18            sycl::ext::oneapi::atomic_ref<int>(
19                sycl::ext::oneapi::memory_order_relaxed,
20                sycl::ext::oneapi::memory_scope_device,
21                sycl::access::address_space::global_space>
22                atomic_op(acc[0]));
23            atomic_op += acc_a[i];
24        });
25    });
26    sycl::host_accessor h_a(buf);
27    std::cout << "Reduction Sum : " << h_a[0] << "\n";
28    std::cout
29        << "Kernel Execution Time of Global Atomics Ref: "
30        << e.get_profiling_info<sycl::info::event_profiling::command_end>() -
31        e.get_profiling_info<sycl::info::event_profiling::command_start>()
32        << "\n";
33    return 0;
34 }
```

It is possible to refactor your code to use local memory space as the following example demonstrates.

**Listing 74:** /examples/atomics/local\_atomics\_ref.cpp

```

1 #include <CL/sycl.hpp>
2 #include <iostream>
3 int main() {
4     constexpr int N = 256 * 256;
5     constexpr int M = 512;
6     constexpr int NUM_WG = N / M;
```

(continues on next page)

(continued from previous page)

```

7 int total = 0;
8 int *a = static_cast<int *>(malloc(sizeof(int) * N));
9 for (int i = 0; i < N; i++)
10    a[i] = 1;
11 sycl::queue q({sycl::property::queue::enable_profiling()});
12 sycl::buffer<int> global(&total, 1);
13 sycl::buffer<int> bufa(a, N);
14 auto e1 = q.submit([&](sycl::handler &h) {
15    sycl::accessor b(global, h);
16    sycl::accessor acc_a(bufa, h, sycl::read_only);
17    auto acc = sycl::accessor<int, 1, sycl::access::mode::read_write,
18                      sycl::access::target::local>(NUM_WG, h);
19    h.parallel_for(sycl::nd_range<1>(N, M), [=](auto it) {
20        auto i = it.get_global_id(0);
21        auto group_id = it.get_group(0);
22        sycl::ext::oneapi::atomic_ref<int,
23                           sycl::ext::oneapi::memory_order_relaxed,
24                           sycl::ext::oneapi::memory_scope_device,
25                           sycl::access::address_space::local_space>
26        atomic_op(acc[group_id]);
27        sycl::ext::oneapi::atomic_ref<int,
28                           sycl::ext::oneapi::memory_order_relaxed,
29                           sycl::ext::oneapi::memory_scope_device,
30                           sycl::access::address_space::global_space>
31        atomic_op_global(b[0]);
32        atomic_op += acc_a[i];
33        it.barrier(sycl::access::fence_space::local_space);
34        if (it.get_local_id() == 0)
35            atomic_op_global += acc[group_id];
36    });
37 });
38 sycl::host_accessor h_global(global);
39 std::cout << "Reduction Sum : " << h_global[0] << "\n";
40 int total_time =
41     (e1.get_profiling_info<sycl::info::event_profiling::command_end>() -
42      e1.get_profiling_info<sycl::info::event_profiling::command_start>());
43 std::cout << "Kernel Execution Time of Local Atomics : " << total_time
44     << "\n";
45 return 0;
46 }

```

## 6.11.2 Local Barriers vs global atomics

Atomics allow multiple work-items in the kernel to work on shared resources. Barriers allow synchronization among the work-items in a work-group. It is possible to achieve the functionality of global atomics through judicious use of kernel launches and local barriers. Depending on the architecture and the amount of data involved one or the other can have better performance.

In the following example, we try to sum a relatively small number of elements in a vector. This task is can be achieved in different ways. The first kernel shown below does this using only one work-item which walks through all elements of the vector and sums them up.

**Listing 75:** /examples/local-global-sync/atomics.cpp

```

1  q.submit([&](auto &h) {
2      sycl::accessor buf_acc(buf, h, sycl::read_only);
3      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
4      h.parallel_for(data_size, [=](auto index) {
5          int glob_id = index[0];
6          if (glob_id == 0) {
7              int sum = 0;
8              for (int i = 0; i < N; i++)
9                  sum += buf_acc[i];
10             sum_acc[0] = sum;
11         }
12     });
13 });

```

In the kernel shown below, the same problem is solved using global atomics where every work-item updates a global variable with the value it needs to accumulate. Although there is a lot of parallelism here, the contention on the global variable is quite high and in most cases its performance will not be very good.

**Listing 76:** /examples/local-global-sync/atomics.cpp

```

1  q.submit([&](auto &h) {
2      sycl::accessor buf_acc(buf, h, sycl::read_only);
3      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
4
5      h.parallel_for(data_size, [=](auto index) {
6          size_t glob_id = index[0];
7          auto v = sycl::ext::oneapi::atomic_ref<
8              int, sycl::ext::oneapi::memory_order::relaxed,
9              sycl::ext::oneapi::memory_scope::device,
10             sycl::access::address_space::global_space>(sum_acc[0]);
11            v.fetch_add(buf_acc[glob_id]);
12        });
13 });

```

In the following kernel, every work-item is responsible for accumulating multiple elements of the vector. This accumulation is done in parallel and then updated into an array that is shared among all work-items of the work-group. At this point all work-items of the work-group do a tree reduction using barriers to synchronize among themselves to reduce intermediate results in shared memory to the final result. This kernel explicitly created exactly one work-group and distributes the responsibility of all elements in the vector to the work-items in the work-group. Although it is not using the full capability of the machine in terms of the number of threads, sometimes this amount of parallelism is enough for small problem sizes.

**Listing 77:** /examples/local-global-sync/atomics.cpp

```

1  Timer timer;
2  q.submit([&](auto &h) {
3      sycl::accessor buf_acc(buf, h, sycl::read_only);
4      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
5      sycl::accessor<int, 1, sycl::access::mode::read_write,

```

(continues on next page)

(continued from previous page)

```

6     sycl::access::target::local>
7     scratch(work_group_size, h);
8     h.parallel_for(sycl::nd_range<1>{work_group_size, work_group_size},
9                     [=](sycl::nd_item<1> item) {
10                     size_t loc_id = item.get_local_id(0);
11                     int sum = 0;
12                     for (int i = loc_id; i < data_size; i += num_work_items)
13                         sum += buf_acc[i];
14                     scratch[loc_id] = sum;
15                     for (int i = work_group_size / 2; i > 0; i >>= 1) {
16                         item.barrier(sycl::access::fence_space::local_space);
17                         if (loc_id < i)
18                             scratch[loc_id] += scratch[loc_id + i];
19                     }
20                     if (loc_id == 0)
21                         sum_acc[0] = scratch[0];
22                 });
23             });

```

The performance of these three kernels varies quite a bit among various platforms and the developer needs to pick the technique that suits their application and the hardware on which it is expected to run.

## 6.12 Restrict Directive

Kernels typically operate on arrays of elements that are provided as pointer arguments. When the compiler cannot determine whether these pointers alias each other, it will conservatively assume that they do, in which case it will not reorder operations on these pointers. Consider the following vector-add example where each iteration of the loop has two loads and one store.

**Listing 78:** /examples/restrict/vec-add-restrict.cpp

```

1 int VectorAdd(sycl::queue &q, const IntArray &a, const IntArray &b,
2                 IntArray &sum, int iter) {
3     sycl::range num_items{a.size()};
4
5     sycl::buffer a_buf(a);
6     sycl::buffer b_buf(b);
7     sycl::buffer sum_buf(sum.data(), num_items);
8
9     auto start = std::chrono::steady_clock::now();
10    for (int i = 0; i < iter; i++) {
11        auto e = q.submit([&](auto &h) {
12            // Input accessors
13            sycl::accessor a_acc(a_buf, h, sycl::read_only);
14            sycl::accessor b_acc(b_buf, h, sycl::read_only);
15            // Output accessor
16            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
17
18            h.parallel_for(num_items,
19                           [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });

```

(continues on next page)

(continued from previous page)

```

20     });
21 }
22 q.wait();
23 auto end = std::chrono::steady_clock::now();
24 std::cout << "Vector add completed on device - took " << (end - start).count()
25     << " u-secs\n";
26 return ((end - start).count());
27 } // end VectorAdd

```

In this case, the programmer leaves all the choices about vector length, the number of work-groups to the compiler. In most cases compiler does a pretty good job of selecting these parameters to get good performance. In some situations it may be better to explicitly choose the number of work-groups and work-group sizes to get good performance and provide hints to the compiler to get better performing code.

The kernel below is written to process multiple elements of the array per work-item and explicitly chooses the number of work-groups and the work-group size. The `intel::kernel_args_restrict` on line 25 tells the compiler that the buffer accessors in this kernel do not alias each other. This will allow the compiler to hoist the loads and stores, thereby providing more time for the instructions to complete and getting better instruction scheduling. The pragma on line 27 directs the compiler to unroll the loop by a factor of two.

**Listing 79:** /examples/restrict/vec-add-restrict.cpp

```

1 int VectorAdd2(sycl::queue &q, const IntArray &a, const IntArray &b,
2                 IntArray &sum, int iter) {
3     sycl::range num_items{a.size()};
4
5     sycl::buffer a_buf(a);
6     sycl::buffer b_buf(b);
7     sycl::buffer sum_buf(sum.data(), num_items);
8     // size_t num_groups =
9     // d_selector.get_info<sycl::info::device::max_compute_units>(); size_t
10    // wg_size = d_selector.get_info<sycl::info::device::max_work_group_size>();
11    size_t num_groups = 1;
12    size_t wg_size = 8;
13    auto start = std::chrono::steady_clock::now();
14    for (int i = 0; i < iter; i++) {
15        q.submit([&](auto &h) {
16            // Input accessors
17            sycl::accessor a_acc(a_buf, h, sycl::read_only);
18            sycl::accessor b_acc(b_buf, h, sycl::read_only);
19            // Output accessor
20            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
21
22            h.parallel_for(
23                sycl::nd_range<1>(num_items * wg_size, wg_size),
24                [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(8)]] [
25                    [intel::kernel_args_restrict]] {
26                        size_t loc_id = index.get_local_id();
27                        // unroll with a directive
28 #pragma unroll(2)
29                        for (size_t i = loc_id; i < mysize; i += wg_size) {

```

(continues on next page)

(continued from previous page)

```

30         sum_acc[i] = a_acc[i] + b_acc[i];
31     }
32   });
33 }
34 q.wait();
35 auto end = std::chrono::steady_clock::now();
36 std::cout << "Vector add2 completed on device - took "
37       << (end - start).count() << " u-secs\n";
38 return ((end - start).count());
39 } // end VectorAdd2

```

The kernel below illustrates manually unrolling of the loop instead of the compiler directive (the compiler may or may not honor the directive depending on its internal heuristic cost model). The advantage of unrolling is that fewer instructions are executed because the loop does not have to iterate as many times, thereby saving on the compare and branch instructions.

**Listing 80:** /examples/restrict/vec-add-restrict.cpp

```

1 int VectorAdd3(sycl::queue &q, const IntArray &a, const IntArray &b,
2                 IntArray &sum, int iter) {
3     sycl::range num_items{a.size()};
4
5     sycl::buffer a_buf(a);
6     sycl::buffer b_buf(b);
7     sycl::buffer sum_buf(sum.data(), num_items);
8     size_t num_groups = 1;
9     size_t wg_size = 8;
10    auto start = std::chrono::steady_clock::now();
11    for (int i = 0; i < iter; i++) {
12        q.submit([&](auto &h) {
13            // Input accessors
14            sycl::accessor a_acc(a_buf, h, sycl::read_only);
15            sycl::accessor b_acc(b_buf, h, sycl::read_only);
16            // Output accessor
17            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
18
19            h.parallel_for(
20                sycl::nd_range<1>(num_groups * wg_size, wg_size), [=
21                ](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(8)]] {
22                    // Manual unrolling
23                    size_t loc_id = index.get_local_id();
24                    for (size_t i = loc_id; i < mysize; i += 16) {
25                        sum_acc[i] = a_acc[i] + b_acc[i];
26                        sum_acc[i + 8] = a_acc[i + 8] + b_acc[i + 8];
27                    }
28                });
29            });
30        q.wait();
31    auto end = std::chrono::steady_clock::now();

```

(continues on next page)

(continued from previous page)

```

33     std::cout << "Vector add3 completed on device - took "
34             << (end - start).count() << " u-secs\n";
35     return ((end - start).count());
36 } // end VectorAdd3

```

The kernel below shows how to reorder the loads and stores so that all loads are issued before any operations on them are done. Typically, there can be many outstanding loads for every thread in the GPU. It is always better to issue the loads before any operations on them are done. This will allow the loads to complete before the data are actually needed for computation.

**Listing 81:** /examples/restrict/vec-add-restrict.cpp

```

1 int VectorAdd4(sycl::queue &q, const IntArray &a, const IntArray &b,
2                 IntArray &sum, int iter) {
3     sycl::range num_items{a.size()};
4
5     sycl::buffer a_buf(a);
6     sycl::buffer b_buf(b);
7     sycl::buffer sum_buf(sum.data(), num_items);
8     size_t num_groups = 1;
9     size_t wg_size = 8;
10    auto start = std::chrono::steady_clock::now();
11    for (int i = 0; i < iter; i++) {
12        q.submit([&](auto &h) {
13            // Input accessors
14            sycl::accessor a_acc(a_buf, h, sycl::read_only);
15            sycl::accessor b_acc(b_buf, h, sycl::read_only);
16            // Output accessor
17            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
18
19            h.parallel_for(
20                sycl::nd_range<1>(num_groups * wg_size, wg_size), [=
21                sycl::nd_item<1> index] [[intel::reqd_sub_group_size(8)]] {
22                    // Manual unrolling
23                    size_t loc_id = index.get_local_id();
24                    for (size_t i = loc_id; i < mysize; i += 16) {
25                        int t1 = a_acc[i];
26                        int t2 = b_acc[i];
27                        int t3 = a_acc[i + 8];
28                        int t4 = b_acc[i + 8];
29                        sum_acc[i] = t1 + t2;
30                        sum_acc[i + 8] = t3 + t4;
31                    }
32                });
33            });
34        }
35        q.wait();
36        auto end = std::chrono::steady_clock::now();
37        std::cout << "Vector add4 completed on device - took "
38                  << (end - start).count() << " u-secs\n";
39    return ((end - start).count());

```

(continues on next page)

(continued from previous page)

```
40 } // end VectorAdd4
```

The following kernel has a restrict directive, which provides hint to the compiler that there is no aliasing among the vectors accessed inside the loop and the compiler can hoist the load over the store just like it was done manually in the previous example.

**Listing 82:** /examples/restrict/vec-add-restrict.cpp

```

1 int VectorAdd5(sycl::queue &q, const IntArray &a, const IntArray &b,
2                 IntArray &sum, int iter) {
3     sycl::range num_items{a.size()};
4
5     sycl::buffer a_buf(a);
6     sycl::buffer b_buf(b);
7     sycl::buffer sum_buf(sum.data(), num_items);
8     size_t num_groups = 1;
9     size_t wg_size = 8;
10    auto start = std::chrono::steady_clock::now();
11    for (int i = 0; i < iter; i++) {
12        q.submit([&](auto &h) {
13            // Input accessors
14            sycl::accessor a_acc(a_buf, h, sycl::read_only);
15            sycl::accessor b_acc(b_buf, h, sycl::read_only);
16            // Output accessor
17            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
18
19            h.parallel_for(
20                sycl::nd_range<1>(num_groups * wg_size, wg_size),
21                [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(8)]] [
22                    [intel::kernel_args_restrict]] {
23                        // compiler needs to hoist the loads
24                        size_t loc_id = index.get_local_id();
25                        for (size_t i = loc_id; i < mysize; i += 16) {
26                            sum_acc[i] = a_acc[i] + b_acc[i];
27                            sum_acc[i + 8] = a_acc[i + 8] + b_acc[i + 8];
28                        }
29                    });
30    });
31 }
32 q.wait();
33 auto end = std::chrono::steady_clock::now();
34 std::cout << "Vector add5 completed on device - took "
35             << (end - start).count() << " u-secs\n";
36 return ((end - start).count());
37 } // end VectorAdd5

```

## 6.13 Submitting Kernels to Multiple Queues

Queues provide a channel to submit kernels for execution on an accelerator. Queues also hold a context that describes the state of the device. This state includes the contents of buffers and any memory needed to execute the kernels. The runtime keeps track of the current device context and avoids unnecessary memory transfers between host and device. Therefore, it is better to submit and launch kernels from one context together as opposed to interleaving the kernel submissions in different contexts.

The following example submits thirty independent kernels that use same buffers as input to compute the result into different output buffers. All these kernels are completely independent and can potentially execute concurrently and out of order. The kernels are submitted to three queues, and the execution of each kernel will incur different costs depending on the how the queues are created.

**Listing 83:** /examples/multiple-queue-submission/multi-queue-light-kernel.cpp

```

1 int VectorAdd(sycl::queue &q1, sycl::queue &q2, sycl::queue &q3,
2                 const IntArray &a, const IntArray &b) {
3     size_t num_items = a.size();
4
5     sycl::buffer a_buf(a);
6     sycl::buffer b_buf(b);
7     sycl::buffer<int> *sum_buf[3 * iter];
8     for (size_t i = 0; i < (3 * iter); i++)
9         sum_buf[i] = new sycl::buffer<int>(256);
10
11    size_t num_groups = 1;
12    size_t wg_size = 256;
13    auto start = std::chrono::steady_clock::now();
14    for (int i = 0; i < iter; i++) {
15        q1.submit([&](auto &h) {
16            sycl::accessor a_acc(a_buf, h, sycl::read_only);
17            sycl::accessor b_acc(b_buf, h, sycl::read_only);
18            auto sum_acc = sum_buf[3 * i]->get_access<sycl::access::mode::write>(h);
19
20            h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
21                           [=](sycl::nd_item<1> index) {
22                               size_t loc_id = index.get_local_id();
23                               sum_acc[loc_id] = 0;
24                               for (size_t i = loc_id; i < array_size; i += wg_size) {
25                                   sum_acc[loc_id] += a_acc[i] + b_acc[i];
26                               }
27                           });
28        });
29        q2.submit([&](auto &h) {
30            sycl::accessor a_acc(a_buf, h, sycl::read_only);
31            sycl::accessor b_acc(b_buf, h, sycl::read_only);
32            auto sum_acc =
33                sum_buf[3 * i + 1]->get_access<sycl::access::mode::write>(h);
34
35            h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
36                           [=](sycl::nd_item<1> index) {

```

(continues on next page)

(continued from previous page)

```

37         size_t loc_id = index.get_local_id();
38         sum_acc[loc_id] = 0;
39         for (size_t i = loc_id; i < array_size; i += wg_size) {
40             sum_acc[loc_id] += a_acc[i] + b_acc[i];
41         }
42     });
43 });
44 q3.submit([&](auto &h) {
45     sycl::accessor a_acc(a_buf, h, sycl::read_only);
46     sycl::accessor b_acc(b_buf, h, sycl::read_only);
47     auto sum_acc =
48         sum_buf[3 * i + 2]->get_access<sycl::access::mode::write>(h);
49
50     h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
51                     [=](sycl::nd_item<1> index) {
52             size_t loc_id = index.get_local_id();
53             sum_acc[loc_id] = 0;
54             for (size_t i = loc_id; i < array_size; i += wg_size) {
55                 sum_acc[loc_id] += a_acc[i] + b_acc[i];
56             }
57         });
58     });
59 }
60 q1.wait();
61 q2.wait();
62 q3.wait();
63 auto end = std::chrono::steady_clock::now();
64 std::cout << "Vector add completed on device - took " << (end - start).count()
65     << " u-secs\n";
66 // check results
67 for (size_t i = 0; i < (3 * iter); i++)
68     delete sum_buf[i];
69 return ((end - start).count());
70 } // end VectorAdd

```

If we submit the kernels to the same queue, we get best performance because all the kernels are able to just transfer the needed inputs once at the beginning and do all their computations.

**Listing 84:** /examples/multiple-queue-submission/multi-queue-light-kernel.cpp

```
1 VectorAdd(q, q, q, a, b);
```

If the kernels are submitted to different queues that share the same context, the performance is similar to submitting it to one queue. The issue to note here is that when a kernel is submitted to a new queue with a different context, the JIT process compiles the kernel to the new device associated with the context. If this JIT compilation time is discounted, the actual execution of the kernels is similar.

**Listing 85:** /examples/multiple-queue-submission/multi-queue-light-kernel.cpp

```

1 sycl::queue q1(d_selector);
2 sycl::queue q2(q1.get_context(), d_selector);
3 sycl::queue q3(q1.get_context(), d_selector);
4 VectorAdd(q1, q2, q3, a, b);

```

If the kernels are submitted to three different queues that have have three different contexts, performance degrades because at kernel invocation, the runtime needs to transfer all input buffers to the accelerator every time. In addition, the kernels will be JITed for each of the contexts.

**Listing 86:** /examples/multiple-queue-submission/multi-queue-light-kernel.cpp

```

1 sycl::queue q4(d_selector);
2 sycl::queue q5(d_selector);
3 sycl::queue q6(d_selector);
4 VectorAdd(q4, q5, q6, a, b);

```

If for some reason there is a need to use different queues, the problem can be alleviated by creating the queues with shared context. This will prevent the need to transfer the input buffers, but the memory footprint of the kernels will increase because all the output buffers have be resident at the same time in the context whereas earlier the same memory on the device could be used for the output buffers. Another thing to remember is the issue of memory-to-compute ratio in the kernels. In the example above, the compute requirement of the kernel is low so the overall execution is dominated by the memory transfers. When the compute is high, these transfers do not contribute much to the overall execution time. This is illustrated in the example below where the amount of computation in the kernel is increased a thousand-fold and the runtime will be different.

**Listing 87:** /examples/multiple-queue-submission/multi-queue-heavy-kernel.cpp

```

1 int VectorAdd(sycl::queue &q1, sycl::queue &q2, sycl::queue &q3,
2                 const IntArray &a, const IntArray &b) {
3     size_t num_items = a.size();
4
5     sycl::buffer a_buf(a);
6     sycl::buffer b_buf(b);
7     sycl::buffer<int> *sum_buf[3 * iter];
8     for (size_t i = 0; i < (3 * iter); i++)
9         sum_buf[i] = new sycl::buffer<int>(256);
10
11    size_t num_groups = 1;
12    size_t wg_size = 256;
13    auto start = std::chrono::steady_clock::now();
14    for (int i = 0; i < iter; i++) {
15        q1.submit([&](auto &h) {
16            sycl::accessor a_acc(a_buf, h, sycl::read_only);
17            sycl::accessor b_acc(b_buf, h, sycl::read_only);
18            auto sum_acc = sum_buf[3 * i]->get_access<sycl::access::mode::write>(h);

```

(continues on next page)

(continued from previous page)

```

19     h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
20                     [=](sycl::nd_item<1> index) {
21                         size_t loc_id = index.get_local_id();
22                         sum_acc[loc_id] = 0;
23                         for (int j = 0; j < 1000; j++)
24                             for (size_t i = loc_id; i < array_size; i += wg_size) {
25                                 sum_acc[loc_id] += a_acc[i] + b_acc[i];
26                             }
27                     });
28     });
29 );
30 q2.submit([&](auto &h) {
31     sycl::accessor a_acc(a_buf, h, sycl::read_only);
32     sycl::accessor b_acc(b_buf, h, sycl::read_only);
33     auto sum_acc =
34         sum_buf[3 * i + 1]->get_access<sycl::access::mode::write>(h);
35
36     h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
37                     [=](sycl::nd_item<1> index) {
38                         size_t loc_id = index.get_local_id();
39                         sum_acc[loc_id] = 0;
40                         for (int j = 0; j < 1000; j++)
41                             for (size_t i = loc_id; i < array_size; i += wg_size) {
42                                 sum_acc[loc_id] += a_acc[i] + b_acc[i];
43                             }
44                     });
45     });
46 q3.submit([&](auto &h) {
47     sycl::accessor a_acc(a_buf, h, sycl::read_only);
48     sycl::accessor b_acc(b_buf, h, sycl::read_only);
49     auto sum_acc =
50         sum_buf[3 * i + 2]->get_access<sycl::access::mode::write>(h);
51
52     h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
53                     [=](sycl::nd_item<1> index) {
54                         size_t loc_id = index.get_local_id();
55                         sum_acc[loc_id] = 0;
56                         for (int j = 0; j < 1000; j++)
57                             for (size_t i = loc_id; i < array_size; i += wg_size) {
58                                 sum_acc[loc_id] += a_acc[i] + b_acc[i];
59                             }
60                     });
61     });
62 }
63 q1.wait();
64 q2.wait();
65 q3.wait();
66 auto end = std::chrono::steady_clock::now();
67 std::cout << "Vector add completed on device - took " << (end - start).count()
68             << " u-secs\n";
69 // check results

```

(continues on next page)

(continued from previous page)

```

70   for (size_t i = 0; i < (3 * iter); i++)
71     delete sum_buf[i];
72   return ((end - start).count());
73 } // end VectorAdd

```

## 6.14 Avoid Redundant Queue Construction

To execute kernels on a device, the user must create a queue, which references an associated context, platform, and device. The context, platform, and device may be chosen automatically, or specified by the user.

A context is constructed, either directly by the user or implicitly when creating a queue, to hold all the runtime information required by the SYCL runtime and the SYCL backend to operate on a device. When a queue is created with no context specified, a new context is implicitly constructed using the default constructor. In general creating a new context is a heavy duty operation due to the need for JIT compiling the program every time a kernel is submitted to a queue with a new context. For good performance one should use as few contexts as possible in their application.

In the following example, a queue is created inside the loop and the kernel is submitted to this new queue. This will essentially invoke the JIT compiler for every iteration of the loop.

**Listing 88:** /examples/redundant-queues/queues.cpp

```

1 int reductionMultipleQMultipleC(std::vector<int> &data, int iter) {
2   const size_t data_size = data.size();
3   int sum = 0;
4
5   int work_group_size = 512;
6   int num_work_groups = 1;
7   int num_work_items = work_group_size;
8
9   const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
10
11  sycl::buffer<int> buf(data.data(), data_size, props);
12  sycl::buffer<int> sum_buf(&sum, 1, props);
13
14  sycl::queue q1{sycl::default_selector{}, exception_handler};
15  // initialize data on the device
16  q1.submit([&](auto &h) {
17    sycl::accessor buf_acc(buf, h, sycl::write_only, sycl::no_init);
18    h.parallel_for(data_size, [=](auto index) { buf_acc[index] = 1; });
19  });
20
21  double elapsed = 0;
22  for (int i = 0; i < iter; i++) {
23    sycl::queue q2{sycl::default_selector{}, exception_handler};
24    if (i == 0)
25      std::cout << q2.get_device().get_info<sycl::info::device::name>() << "\n";
26    // reductionMultipleQMultipleC main begin
27    Timer timer;
28    q2.submit([&](auto &h) {

```

(continues on next page)

(continued from previous page)

```

29     sycl::accessor buf_acc(buf, h, sycl::read_only);
30     sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
31     sycl::accessor<int, 1, sycl::access::mode::read_write,
32                     sycl::access::target::local>
33     scratch(work_group_size, h);
34     h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
35                    [=](sycl::nd_item<1> item) {
36                     size_t loc_id = item.get_local_id(0);
37                     int sum = 0;
38                     for (int i = loc_id; i < data_size; i += num_work_items)
39                         sum += buf_acc[i];
40                     scratch[loc_id] = sum;
41                     for (int i = work_group_size / 2; i > 0; i >= 1) {
42                         item.barrier(sycl::access::fence_space::local_space);
43                         if (loc_id < i)
44                             scratch[loc_id] += scratch[loc_id + i];
45                     }
46                     if (loc_id == 0)
47                         sum_acc[0] = scratch[0];
48                 });
49             });
50             // reductionMultipleQMultipleC main end
51             q2.wait();
52             sycl::host_accessor h_acc(sum_buf);
53             sum = h_acc[0];
54             elapsed += timer.Elapsed();
55         }
56         elapsed = elapsed / iter;
57         if (sum == sum_expected)
58             std::cout << "SUCCESS: Time reductionMultipleQMultipleC = " << elapsed
59             << "s"
60             << " sum = " << sum << "\n";
61         else
62             std::cout << "ERROR: reductionMultipleQMultipleC Expected " << sum_expected
63             << " but got " << sum << "\n";
64         return sum;
65     } // end reductionMultipleQMultipleC

```

The above program can be rewritten by moving the queue declaration outside the loop and the performance improves quite dramatically.

**Listing 89:** /examples/redundant-queues/queues.cpp

```

1 int reductionSingleQ(std::vector<int> &data, int iter) {
2     const size_t data_size = data.size();
3     int sum = 0;
4
5     int work_group_size = 512;
6     int num_work_groups = 1;
7     int num_work_items = work_group_size;
8

```

(continues on next page)

(continued from previous page)

```
9 const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
10
11 sycl::buffer<int> buf(data.data(), data_size, props);
12 sycl::buffer<int> sum_buf(&sum, 1, props);
13 sycl::queue q{sycl::default_selector{}, exception_handler};
14 std::cout << q.get_device().get_info<sycl::info::device::name>() << "\n";
15
16 // initialize data on the device
17 q.submit([&](auto &h) {
18     sycl::accessor buf_acc(buf, h, sycl::write_only, sycl::no_init);
19     h.parallel_for(data_size, [=](auto index) { buf_acc[index] = 1; });
20 });
21
22 double elapsed = 0;
23 for (int i = 0; i < iter; i++) {
24     // reductionIntBarrier main begin
25     Timer timer;
26     q.submit([&](auto &h) {
27         sycl::accessor buf_acc(buf, h, sycl::read_only);
28         sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
29         sycl::accessor<int, 1, sycl::access::mode::read_write,
30                         sycl::access::target::local>
31             scratch(work_group_size, h);
32         h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
33                         [=](sycl::nd_item<1> item) {
34             size_t loc_id = item.get_local_id(0);
35             int sum = 0;
36             for (int i = loc_id; i < data_size; i += num_work_items)
37                 sum += buf_acc[i];
38             scratch[loc_id] = sum;
39             for (int i = work_group_size / 2; i > 0; i >= 1) {
40                 item.barrier(sycl::access::fence_space::local_space);
41                 if (loc_id < i)
42                     scratch[loc_id] += scratch[loc_id + i];
43             }
44             if (loc_id == 0)
45                 sum_acc[0] = scratch[0];
46         });
47     });
48     // reductionSingleQ main end
49     q.wait();
50     sycl::host_accessor h_acc(sum_buf);
51     sum = h_acc[0];
52     elapsed += timer.Elapsed();
53 }
54 elapsed = elapsed / iter;
55 if (sum == sum_expected)
56     std::cout << "SUCCESS: Time reductionSingleQ    = " << elapsed << "s"
57     << " sum = " << sum << "\n";
58 else
59     std::cout << "ERROR: reductionSingleQ Expected " << sum_expected
```

(continues on next page)

(continued from previous page)

```

60     << " but got " << sum << "\n";
61     return sum;
62 } // end reductionSingleQ

```

In case there is a need to create multiple queues, one should try to share the contexts among the queues. This will improve the performance. The above kernel is rewritten as shown below where the new queues created inside the loop and the queue outside the loop share the context. In this case the performance is same as the one with one queue.

**Listing 90:** /examples/redundant-queues/queues.cpp

```

1 int reductionMultipleQSingleC(std::vector<int> &data, int iter) {
2     const size_t data_size = data.size();
3     int sum = 0;
4
5     int work_group_size = 512;
6     int num_work_groups = 1;
7     int num_work_items = work_group_size;
8
9     const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
10
11    sycl::buffer<int> buf(data.data(), data_size, props);
12    sycl::buffer<int> sum_buf(&sum, 1, props);
13
14    sycl::queue q1{sycl::default_selector{}, exception_handler};
15    // initialize data on the device
16    q1.submit([&](auto &h) {
17        sycl::accessor buf_acc(buf, h, sycl::write_only, sycl::no_init);
18        h.parallel_for(data_size, [=](auto index) { buf_acc[index] = 1; });
19    });
20
21    double elapsed = 0;
22    for (int i = 0; i < iter; i++) {
23        sycl::queue q2{q1.get_context(), sycl::default_selector{},
24                      exception_handler};
25        if (i == 0)
26            std::cout << q2.get_device().get_info<sycl::info::device::name>() << "\n";
27        // reductionMultipleQSingleC main begin
28        Timer timer;
29        q2.submit([&](auto &h) {
30            sycl::accessor buf_acc(buf, h, sycl::read_only);
31            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
32            sycl::accessor<int, 1, sycl::access::mode::read_write,
33                          sycl::access::target::local>
34            scratch(work_group_size, h);
35            h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
36                          [=](sycl::nd_item<1> item) {
37                size_t loc_id = item.get_local_id(0);
38                int sum = 0;
39                for (int i = loc_id; i < data_size; i += num_work_items)
40                    sum += buf_acc[i];
41            });
42        });
43    }
44
45    std::cout << "Time taken: " << timer.get_time() << " ms\n";
46
47    return sum;
48 }

```

(continues on next page)

(continued from previous page)

```

41     scratch[loc_id] = sum;
42     for (int i = work_group_size / 2; i > 0; i >= 1) {
43         item.barrier(sycl::access::fence_space::local_space);
44         if (loc_id < i)
45             scratch[loc_id] += scratch[loc_id + i];
46     }
47     if (loc_id == 0)
48         sum_acc[0] = scratch[0];
49     });
50 };
51 // reductionMultipleQSingleC main end
52 q2.wait();
53 sycl::host_accessor h_acc(sum_buf);
54 sum = h_acc[0];
55 elapsed += timer.Elapsed();
56 }
57 elapsed = elapsed / iter;
58 if (sum == sum_expected)
59     std::cout << "SUCCESS: Time reductionMultipleQSingleContext = " << elapsed
60     << "s"
61     << " sum = " << sum << "\n";
62 else
63     std::cout << "ERROR: reductionMultipleQSingleContext Expected "
64     << sum_expected << " but got " << sum << "\n";
65 return sum;
66 } // end reductionMultipleQSingleC

```

## 6.15 Considerations for selecting work-group size

In DPC++ one can select the work-group size for `nd_range` kernels. The size of work-group has important implications for utilization of the compute resources, vector lanes and communication among the work-items. The work-items in same work-group may have access to hardware resources like shared memory and hardware synchronization capabilities which will allow them to run and communicate more efficiently than work-items across work-groups. So in general one needs to pick the maximum work-group size supported by the accelerator. The maximum work-group size can be queried by the call `device::get_info<cl::sycl::info::device::max_work_group_size>()`.

To illustrate the impact of the choice of work-group size consider the following reduction kernel which goes through a large vector to add all the elements in it. The function that runs the kernels takes in the work-group-size and sub-group-size as arguments which allows one to run experiments with different values. The performance difference can be seen from the timings reported when the kernel is called with different values for work-group size.

**Listing 91:** /examples/work-group-size/reduction-wg-size.cpp

```

1 void reduction(sycl::queue &q, std::vector<int> &data, std::vector<int> &flush,
2                 int iter, int vec_size, int work_group_size) {
3     const size_t data_size = data.size();
4     const size_t flush_size = flush.size();

```

(continues on next page)

(continued from previous page)

```

5 int sum = 0;
6
7 const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
8 // int vec_size =
9 // q.get_device().get_info<sycl::info::device::native_vector_width_int>();
10 int num_work_items = data_size / work_group_size;
11 sycl::buffer<int> buf(data.data(), data_size, props);
12 sycl::buffer<int> flush_buf(flush.data(), flush_size, props);
13 sycl::buffer<int> sum_buf(&sum, 1, props);
14
15 init_data(q, buf, data_size);
16
17 double elapsed = 0;
18 for (int i = 0; i < iter; i++) {
19     q.submit([&](auto &h) {
20         sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
21
22         h.parallel_for(1, [=](auto index) { sum_acc[index] = 0; });
23     });
24     // flush the cache
25     q.submit([&](auto &h) {
26         sycl::accessor flush_acc(flush_buf, h, sycl::write_only, sycl::no_init);
27         h.parallel_for(flush_size, [=](auto index) { flush_acc[index] = 1; });
28     });
29
30     Timer timer;
31     // reductionMapToHWVector main begin
32     q.submit([&](auto &h) {
33         sycl::accessor buf_acc(buf, h, sycl::read_only);
34         sycl::accessor<int, 1, sycl::access::mode::read_write,
35                         sycl::access::target::local>
36             scratch(work_group_size, h);
37         sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
38
39         h.parallel_for(
40             sycl::nd_range<1>(num_work_items, work_group_size), [=
41             ](sycl::nd_item<1> item) [[intel::reqd_sub_group_size(16)]] {
42                 auto v = sycl::ext::oneapi::atomic_ref<
43                     int, sycl::ext::oneapi::memory_order::relaxed,
44                     sycl::ext::oneapi::memory_scope::device,
45                     sycl::access::address_space::global_space>(sum_acc[0]);
46                 int sum = 0;
47                 int glob_id = item.get_global_id();
48                 int loc_id = item.get_local_id();
49                 for (int i = glob_id; i < data_size; i += num_work_items)
50                     sum += buf_acc[i];
51                 scratch[loc_id] = sum;
52
53                 for (int i = work_group_size / 2; i > 0; i >= 1) {
54                     item.barrier(sycl::access::fence_space::local_space);
55                     if (loc_id < i)

```

(continues on next page)

(continued from previous page)

```

56         scratch[loc_id] += scratch[loc_id + i];
57     }
58
59     if (loc_id == 0)
60         v.fetch_add(scratch[0]);
61     });
62 };
63 q.wait();
64 elapsed += timer.Elapsed();
65 sycl::host_accessor h_acc(sum_buf);
66 sum = h_acc[0];
67 }
68 elapsed = elapsed / iter;
69 std::string msg = "with work-groups=" + std::to_string(work_group_size);
70 check_result(elapsed, msg, sum);
71 } // reduction end

```

In the code below, the above kernel is called with two different values - **2\*vec-size** and maximum possible work-group size supported by the accelerator. The performance of the kernel when work-group size is equal to **2\*vec-size** will be lower than when the work-group size is the maximum possible value.

### **Listing 92:**

/examples/work-group-size/reduction-wg-size.cpp

```

1 int vec_size = 16;
2 int work_group_size = vec_size;
3 reduction(q, data, extra, 16, vec_size, work_group_size);
4 work_group_size =
5     q.get_device().get_info<sycl::info::device::max_work_group_size>();
6 reduction(q, data, extra, 16, vec_size, work_group_size);

```

In situations where there are no barriers or atomics used, the work-group size will not impact the performance. To illustrate this consider the following **vec\_copy** kernel where there are no atomics or barriers.

### **Listing 93:** /examples/work-group-size/vec-copy.cpp

```

1 void vec_copy(sycl::queue &q, std::vector<int> &src, std::vector<int> &dst,
2               std::vector<int> &flush, int iter, int work_group_size) {
3     const size_t data_size = src.size();
4     const size_t flush_size = flush.size();
5     int sum = 0;
6
7     const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
8     int num_work_items = data_size;
9     double elapsed = 0;
10    {
11        sycl::buffer<int> src_buf(src.data(), data_size, props);
12        sycl::buffer<int> dst_buf(dst.data(), data_size, props);
13        sycl::buffer<int> flush_buf(flush.data(), flush_size, props);
14
15        for (int i = 0; i < iter; i++) {

```

(continues on next page)

(continued from previous page)

```

16    // flush the cache
17    q.submit([&](auto &h) {
18        sycl::accessor flush_acc(flush_buf, h, sycl::write_only, sycl::no_init);
19        h.parallel_for(flush_size, [=](auto index) { flush_acc[index] = 1; });
20    });
21
22    Timer timer;
23    q.submit([&](auto &h) {
24        sycl::accessor src_acc(src_buf, h, sycl::read_only);
25        sycl::accessor dst_acc(dst_buf, h, sycl::write_only, sycl::no_init);
26
27        h.parallel_for(
28            sycl::nd_range<1>(num_work_items, work_group_size), [=]
29            )(sycl::nd_item<1> item) [[intel::reqd_sub_group_size(16)]] {
30            int glob_id = item.get_global_id();
31            dst_acc[glob_id] = src_acc[glob_id];
32        });
33    });
34    q.wait();
35    elapsed += timer.Elapsed();
36 }
37 }
38 elapsed = elapsed / iter;
39 std::string msg = "with work-group-size=" + std::to_string(work_group_size);
40 check_result(elapsed, msg, dst);
41 } // vec_copy end

```

In the code below, the above kernel is called with different work-group sizes. All the above calls to the kernel will have similar run times which indicates that there is no impact of work-group size on performance. The reason for this is that the threads created within a work-group and threads from different work-groups behave in a similar manner from scheduling and resourcing point of view when there are no barriers or shared memory in the work-groups.

**Listing 94:** /examples/work-group-size/vec-copy.cpp

```

1 int vec_size = 16;
2 int work_group_size = vec_size;
3 vec_copy(q, src, dst, extra, 16, work_group_size);
4 work_group_size = 2 * vec_size;
5 vec_copy(q, src, dst, extra, 16, work_group_size);
6 work_group_size = 4 * vec_size;
7 vec_copy(q, src, dst, extra, 16, work_group_size);
8 work_group_size = 8 * vec_size;
9 vec_copy(q, src, dst, extra, 16, work_group_size);
10 work_group_size = 16 * vec_size;
11 vec_copy(q, src, dst, extra, 16, work_group_size);

```

In some accelerators, a minimum sub-group size is needed to obtain good performance due to the way in which threads are scheduled among the processing elements. In such a situation one can see big performance difference when the number of sub-groups are less than the minimum. The call to the kernel on line 3 above has only one sub-group while the call on line 5 has two sub-groups. There will be a significant performance difference in

the timings for these two kernel invocations on an accelerator that performs scheduling of 2 sub-groups at a time.

## 7.0 Memory

Accelerators have access to a rich memory hierarchy. Utilizing the right level in the hierarchy is critical to get best performance. In this section we cover topics related to the declaration, movement, and access to the memory hierarchy.

### 7.1 Performance Impact of USM and Buffers

SYCL offers several choices for managing memory on the device. In this section, we discuss the performance tradeoffs. We briefly introduce the concepts. See [Data Parallel C++](#) for an in-depth explanation.

As with other language features, the specification defines the behavior but not the implementation so performance characteristics can change between software versions and devices. We provide best practices.

**Buffers.** A buffer is a container for data that can be accessed from a device and the host. The SYCL runtime manages the memory by providing APIs for allocating, reading, and writing memory. The runtime is responsible for moving data between host and device and synchronizing access to the data.

**Unified Shared Memory (USM).** USM allows reading and writing of data with conventional pointers, in contrast to buffers where access to data is exclusively by API. USM has 2 commonly-used variants. **Device** allocations can only be accessed from the device and therefore require explicit movement of data between host and device. **Shared** allocations can be referenced from device or host, with the runtime automatically moving memory.

We illustrate the tradeoffs between the choices by showing the same example program written with the 3 models. To bring out the issues, we use a program where a GPU and the host cooperatively compute, and therefore need to ship data back and forth.

We start by showing the serial computation below. Assume that we want to perform the loop at line 9 on the GPU and the loop on line 14 on the CPU. Both loops read and write the data array so data must move between host and GPU for each iteration of the loop in line 8.

**Listing 95:** /examples/usm/usm-buffer.cpp

```

1 void serial(int stride) {
2     // Allocate and initialize data
3     float *data = new float[data_size];
4     init(data);
5
6     timer it;
7
8     for (int i = 0; i < time_steps; i++) {
9         for (int j = 0; j < data_size; j++) {
10            for (int k = 0; k < device_steps; k++)
11                data[j] += 1.0;
12        }
13
14        for (int j = 0; j < data_size; j += stride)

```

(continues on next page)

(continued from previous page)

```

15     data[j] += 1.0;
16 }
17 put_elapsed_time(it);
18
19 check(data);
20
21 delete[] data;
22 } // serial

```

## 7.1.1 Buffers

Below, we show the same computation using buffers to manage data. A buffer is created at line 3 and initialized by the `init` function. The `init` function is not shown. It accepts an accessor or a pointer. The `parallel_for` executes the kernel defined on line 13. The kernel uses the `device_data` accessor to read and write data in `buffer_data`. Note that the code does not specify the location of data. An accessor indicates when and where the data is needed, and the SYCL runtime moves the data to the device (if necessary) and then launches the kernel. The `host_accessor` on line 21 indicates that the data will be read/written on the host. Since the kernel is also read/writing `buffer_data`, the `host_accessor` constructor waits for the kernel to complete and moves data to the host to perform the read/write on line 23. In the next iteration of the loop the accessor constructor on line 11 waits until the until the data is moved back to the device, which effectively delays launching the kernel.

**Listing 96:** /examples/usm/usm-buffer.cpp

```

1 void buffer_data(int stride) {
2     // Allocate buffer, initialize on host
3     sycl::buffer<float> buffer_data{data_size};
4     init(sycl::host_accessor(buffer_data, sycl::write_only, sycl::no_init));
5
6     timer it;
7     for (int i = 0; i < time_steps; i++) {
8
9         // Compute on device
10        q.submit([&](auto &h) {
11            sycl::accessor device_data(buffer_data, h);
12
13            auto compute = [=](auto id) {
14                for (int k = 0; k < device_steps; k++)
15                    device_data[id] += 1.0;
16            };
17            h.parallel_for(data_size, compute);
18        });
19
20         // Compute on host
21        sycl::host_accessor host_data(buffer_data);
22        for (int i = 0; i < data_size; i += stride)
23            host_data[i] += 1.0;
24    }
25    put_elapsed_time(it);
26
27    const sycl::host_accessor h(buffer_data);

```

(continues on next page)

(continued from previous page)

```
28     check(h);
29 } // buffer_data
```

## Performance Considerations

The data access on lines 15 and 23 appear to be simple array references, but they are implemented by the SYCL runtime with C++ operator overloading. The efficiency of accessor array references depends on the implementation. In practice, device code pays no overhead for overloading compared to direct memory references. The runtime does not know in advance which part of the buffer is accessed, so it must ensure all the data is on the device before the kernel begins. This is true today, but may change over time. The same is not currently true for the host\_accessor. The runtime does not move all the data to the host. The array references are implemented with more complex code and are significantly slower than native C++ array references. While it is acceptable to reference a small amount of data, computationally intensive algorithms using host\_accessor pay a large performance penalty and should be avoided.

Another issue is concurrency. A host\_accessor can block kernels that reference the same buffer from launching, even if the accessor is not actively being used to read/write data. Limit the scope that contains the host\_accessor to the minimum possible. In this example, the host accessor on line 4 is destroyed after the init function returns and the host accessor on line 21 is destroyed at the end of each loop iteration.

### 7.1.2 Shared Allocations

Next we show the same algorithm implemented with shared allocations. Data is allocated on line 2. Accessors are not needed because USM-allocated data can be referenced with conventional allows pointers. Therefore, the array references on lines 10 and 15 can be implemented with simple indexing. The parallel\_for on line 12 ends with a wait to ensure the kernel finishes before the host accesses data on line 15. Similar to buffers, the SYCL runtime ensures that all the data is resident on the device before launching a kernel. And like buffers, shared allocations are not copied to the host unless it is referenced. The first time the host references data, there is an operating system page fault, a page of data is copied from device to host, and execution continues. Subsequent references to data on the same page execute at full speed. When a kernel is launched, all of the host-resident pages are flushed back to the device.

**Listing 97:** /examples/usm/usm-buffer.cpp

```
1 void shared_usm_data(int stride) {
2     float *data = sycl::malloc_shared<float>(data_size, q);
3     init(data);
4
5     timer it;
6
7     for (int i = 0; i < time_steps; i++) {
8         auto compute = [=](auto id) {
9             for (int k = 0; k < device_steps; k++)
10                 data[id] += 1.0;
11         };
12         q.parallel_for(data_size, compute).wait();
13
14         for (int k = 0; k < data_size; k += stride)
```

(continues on next page)

(continued from previous page)

```

15     data[k] += 1.0;
16 }
17 q.wait();
18 put_elapsed_time(it);
19
20 check(data);
21
22 sycl::free(data, q);
23 } // shared_usm_data

```

## Performance Considerations

Compared to buffers, data references are simple pointers and perform well. However, servicing page faults to bring data to the host incurs overhead in addition to the cost of transferring data. The impact on the application depends on the reference pattern. Sparse random access has the highest overhead and linear scans through data have lower impact from page faults.

Since all synchronization is explicit and under programmer control, concurrency is not an issue for a well designed program.

### 7.1.3 Device Allocations

The same program with device allocation can be found below. With device allocation, data can only be directly accessed on the device and must be explicitly copied to the host, as is done on line 21. All synchronization between device and host are explicit. Line 21 ends with a `wait` so the host code will not execute until the asynchronous copy finishes. The queue definition is not shown but uses an in-order queue so the `memcpy` on line 21 waits for the `parallel_for` on line 18 to complete.

**Listing 98:** /examples/usm/usm-buffer.cpp

```

1 void device_usm_data(int stride) {
2     // Allocate and initialize host data
3     float *host_data = new float[data_size];
4     init(host_data);
5
6     // Allocate device data
7     float *device_data = sycl::malloc_device<float>(data_size, q);
8
9     timer it;
10
11    for (int i = 0; i < time_steps; i++) {
12        // Copy data to device and compute
13        q.memcpy(device_data, host_data, sizeof(float) * data_size);
14        auto compute = [=](auto id) {
15            for (int k = 0; k < device_steps; k++)
16                device_data[id] += 1.0;
17        };
18        q.parallel_for(data_size, compute);
19
20        // Copy data to host and compute

```

(continues on next page)

(continued from previous page)

```

21     q.memcpy(host_data, device_data, sizeof(float) * data_size).wait();
22     for (int k = 0; k < data_size; k += stride)
23         host_data[k] += 1.0;
24     }
25     q.wait();
26     put_elapsed_time(it);
27
28     check(host_data);
29
30     sycl::free(device_data, q);
31     delete[] host_data;
32 } // device_usm_data

```

## Performance Considerations

Both data movement and synchronization are explicit and under the full control of the programmer. Array references are array references on host, so it has neither the page faults overhead of shared allocations, nor the overloading overhead associated with buffers. Shared allocations only transfer the data that the host actually references, with a memory page granularity. In theory, device allocations allow on-demand movement of any granularity. In practice, fine grained, asynchronous movement of data can be complex and most programmers simply move the entire data structure once. The requirement for explicit data movement and synchronization makes the code more complicated, but device allocations can provide the best performance.

## 7.2 Optimizing Memory Movement between Host and Accelerator

Buffers can be created using properties to control how they are allocated. One such property is `use_host_ptr`. This informs the runtime that if possible, the host memory should be directly used by the buffer instead of a copy. This will avoid the need to copy the content of the buffer back and forth between the host memory and the buffer memory, potentially saving time during the buffer creation and destruction. Another case when the GPU and CPU have shared memory, it is possible to avoid copies of memory through sharing of pages. But for page sharing to be possible, the allocated memory needs to have some properties like being aligned on page boundary. In case of discrete devices, the benefit may not be realized because any memory operation by the accelerator will have to go across PCIe or some other slower interface than the memory of the accelerator.

The following code shows how to print the memory addresses on the host, inside the buffer, and on the accelerator device inside the kernel.

### Listing 99:

/examples/memory-movement/vec-buffer-host.cpp

```

1 int VectorAdd0(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b,
2                 AlignedVector<int> &sum, int iter) {
3     sycl::range num_items{a.size()};
4
5     const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
6
7     for (int i = 0; i < iter; i++) {
8         sycl::buffer a_buf(a, props);

```

(continues on next page)

(continued from previous page)

```

9    sycl::buffer b_buf(b, props);
10   sycl::buffer sum_buf(sum.data(), num_items, props);
11   {
12     sycl::host_accessor a_host_acc(a_buf);
13     std::cout << "add0: buff memory address =" << a_host_acc.get_pointer()
14       << "\n";
15     std::cout << "add0: address of vector a = " << a.data() << "\n";
16   }
17   q.submit([&](auto &h) {
18     // Input accessors
19     sycl::accessor a_acc(a_buf, h, sycl::read_only);
20     sycl::accessor b_acc(b_buf, h, sycl::read_only);
21     // Output accessor
22     sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
23     sycl::stream out(1024 * 1024, 1 * 128, h);
24
25     h.parallel_for(num_items, [=](auto i) {
26       if (i[0] == 0)
27         out << "add0: dev addr = " << a_acc.get_pointer() << "\n";
28       sum_acc[i] = a_acc[i] + b_acc[i];
29     });
30   });
31 }
32 q.wait();
33 return (0);
34 }
```

When this program is run it can be seen that the addresses for all three (host, in the buffer, and on the accelerator) are the same when the property `use_host_ptr` is set for integrated GPU devices. But for discrete GPU devices the buffer and device addresses will be different. Also note that in line 1, none of the incoming arguments are declared to be `const`. If these are declared `const` then during buffer creation they are copied and new memory is allocated instead of reusing the memory in the host vectors. The code snippet below demonstrates this. When this code is executed, we see that the addresses associated with the incoming vectors are different from the memory present in the buffer and also the memory present in the accelerator device.

### **Listing 100:**

/examples/memory-movement/vec-buffer-host.cpp

```

1 int VectorAdd1(sycl::queue &q, const AlignedVector<int> &a,
2                 const AlignedVector<int> &b, AlignedVector<int> &sum, int iter) {
3   sycl::range num_items{a.size()};
4
5   const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
6
7   for (int i = 0; i < iter; i++) {
8     sycl::buffer a_buf(a, props);
9     sycl::buffer b_buf(b, props);
10    sycl::buffer sum_buf(sum.data(), num_items, props);
11    {
12      sycl::host_accessor a_host_acc(a_buf);
13      std::cout << "add1: buff memory address =" << a_host_acc.get_pointer()
```

(continues on next page)

(continued from previous page)

```

14     << "\n";
15     std::cout << "add1: address of vector aa = " << a.data() << "\n";
16 }
17 q.submit([&](auto &h) {
18     // Input accessors
19     sycl::accessor a_acc(a_buf, h, sycl::read_only);
20     sycl::accessor b_acc(b_buf, h, sycl::read_only);
21     // Output accessor
22     sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
23     sycl::stream out(16 * 1024, 16 * 1024, h);
24
25     h.parallel_for(num_items, [=](auto i) {
26         if (i[0] == 0)
27             out << "add1: dev addr = " << a_acc.get_pointer() << "\n";
28         sum_acc[i] = a_acc[i] + b_acc[i];
29     });
30 });
31 }
32 q.wait();
33 return (0);
34 }
```

The kernel `vectorAdd3` will not incur the cost of copying the memory contents from the buffer to the accelerator device because the `use_host_ptr` property is set while creating the buffers, and the buffers are aligned on a page boundary for an integrated GPU device. If memory pointed to by a buffer is not aligned on a page boundary, then new memory is allocated that aligns on a page boundary and the contents of the buffer are copied into that memory. This new memory from the buffer is then shared with the accelerator either by copying the contents from the buffer on host to the device (in case of accelerators that do not share any memory) or by using the page tables to avoid a physical copy of memory available on the device in case of shared memory.

### **Listing 101:**

/examples/memory-movement/vec-buffer-host.cpp

```

1 int VectorAdd2(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b,
2                 AlignedVector<int> &sum, int iter) {
3     sycl::range num_items{a.size()};
4
5     const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
6
7     auto start = std::chrono::steady_clock::now();
8     for (int i = 0; i < iter; i++) {
9         sycl::buffer a_buf(a, props);
10        sycl::buffer b_buf(b, props);
11        sycl::buffer sum_buf(sum.data(), num_items, props);
12        q.submit([&](auto &h) {
13            // Input accessors
14            sycl::accessor a_acc(a_buf, h, sycl::read_only);
15            sycl::accessor b_acc(b_buf, h, sycl::read_only);
16            // Output accessor
17            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
18        });
19    }
20 }
```

(continues on next page)

(continued from previous page)

```

19     h.parallel_for(num_items,
20                     [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
21   });
22 }
23 q.wait();
24 auto end = std::chrono::steady_clock::now();
25 std::cout << "Vector add2 completed on device - took "
26             << (end - start).count() << " u-secs\n";
27 return ((end - start).count());
28 }
```

The kernel below will incur the cost of copying memory contents between the host and buffer, and also from the buffer to the accelerator.

### **Listing 102:**

/examples/memory-movement/vec-buffer-host.cpp

```

1 int VectorAdd3(sycl::queue &q, const AlignedVector<int> &a,
2                 const AlignedVector<int> &b, AlignedVector<int> &sum, int iter) {
3     sycl::range num_items{a.size()};
4
5     auto start = std::chrono::steady_clock::now();
6     for (int i = 0; i < iter; i++) {
7         sycl::buffer a_buf(a);
8         sycl::buffer b_buf(b);
9         sycl::buffer sum_buf(sum.data(), num_items);
10        auto e = q.submit([&](auto &h) {
11            // Input accessors
12            sycl::accessor a_acc(a_buf, h, sycl::read_only);
13            sycl::accessor b_acc(b_buf, h, sycl::read_only);
14            // Output accessor
15            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
16
17            h.parallel_for(num_items,
18                           [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
19        });
20    }
21    q.wait();
22    auto end = std::chrono::steady_clock::now();
23    std::cout << "Vector add3 completed on device - took "
24                  << (end - start).count() << " u-secs\n";
25    return ((end - start).count());
26 }
```

Care must be taken to ensure that unnecessary copies are avoided during the creation of buffers and passing the memory from the buffers to the kernels. Even when the accelerator shares memory with the host, a few additional conditions must be satisfied to avoid these extra copies.

## 7.3 Avoid moving data back and forth between host and device

The cost of moving data between host and device is quite high especially in the case of discrete accelerators. So it is very important to avoid data transfers between host and device as much as possible. In some situations it may be required to bring the data that was computed by a kernel on the accelerator to the host and do some operation on it and send it back to the device for further processing. In such situation we will end up paying for the cost of device to host transfer and then again host to device transfer.

Consider the following example, where one kernel produces data through some operation(in this case vector add) into a new vector. This vector is then transformed into another vector by applying a function on each value and then fed as input into another kernel for some additional computation. This form of computation is quite common and occurs in many domains where algorithms are iterative and output from one computation needs to be fed as input into another computation. One classic example is machine learning models which are structured as layers of computation and output of one layer is input to the next layer.

**Listing 103:** /examples/host-device-memory/mem-move.cpp

```

1 double myFunc1(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b,
2                 AlignedVector<int> &c, AlignedVector<int> &d,
3                 AlignedVector<int> &res, int iter) {
4     sycl::range num_items{a.size()};
5     VectorAllocator<int> alloc;
6     AlignedVector<int> sum(a.size(), alloc);
7
8     const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
9     sycl::buffer a_buf(a, props);
10    sycl::buffer b_buf(b, props);
11    sycl::buffer c_buf(b, props);
12    sycl::buffer d_buf(b, props);
13    sycl::buffer res_buf(res, props);
14    sycl::buffer sum_buf(sum.data(), num_items, props);
15
16    Timer timer;
17    for (int i = 0; i < iter; i++) {
18        // kernel1
19        q.submit([&](auto &h) {
20            // Input accessors
21            sycl::accessor a_acc(a_buf, h, sycl::read_only);
22            sycl::accessor b_acc(b_buf, h, sycl::read_only);
23            // Output accessor
24            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
25
26            h.parallel_for(num_items,
27                           [=](auto id) { sum_acc[id] = a_acc[id] + b_acc[id]; });
28        });
29
30    {
31        sycl::host_accessor h_acc(sum_buf);
32        for (int j = 0; j < a.size(); j++)
33            if (h_acc[j] > 10)
34                h_acc[j] = 1;
35            else

```

(continues on next page)

(continued from previous page)

```

36     h_acc[j] = 0;
37 }
38
39 // kernel2
40 q.submit([&](auto &h) {
41     // Input accessors
42     sycl::accessor sum_acc(sum_buf, h, sycl::read_only);
43     sycl::accessor c_acc(c_buf, h, sycl::read_only);
44     sycl::accessor d_acc(d_buf, h, sycl::read_only);
45     // Output accessor
46     sycl::accessor res_acc(res_buf, h, sycl::write_only, sycl::no_init);
47
48     h.parallel_for(num_items, [=](auto id) {
49         res_acc[id] = sum_acc[id] * c_acc[id] + d_acc[id];
50     });
51 });
52 q.wait();
53 }
54 double elapsed = timer.Elapsed() / iter;
55 return (elapsed);
56 } // end myFunc1

```

Instead of bringing the data to the host and applying the function to the data and sending it back to the device in the second kernel, one can create a kernel to execute this function on the device itself. This has the advantage of avoiding the round trip of data from device to host. This technique is shown in the example below which is functionally same as the code before - we now introduce a third kernel **kernel3** which operates on the intermediate data in **accum\_buf** in between **kernel1** and **kernel2**.

**Listing 104:** /examples/host-device-memory/mem-move.cpp

```

1 double myFunc2(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b,
2                 AlignedVector<int> &c, AlignedVector<int> &d,
3                 AlignedVector<int> &res, int iter) {
4     sycl::range num_items{a.size()};
5     VectorAllocator<int> alloc;
6     AlignedVector<int> sum(a.size(), alloc);
7
8     const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
9     sycl::buffer a_buf(a, props);
10    sycl::buffer b_buf(b, props);
11    sycl::buffer c_buf(c, props);
12    sycl::buffer d_buf(d, props);
13    sycl::buffer res_buf(res, props);
14    sycl::buffer sum_buf(sum.data(), num_items, props);
15
16    Timer timer;
17    for (int i = 0; i < iter; i++) {
18        // kernel1
19        q.submit([&](auto &h) {
20            // Input accessors
21            sycl::accessor a_acc(a_buf, h, sycl::read_only);

```

(continues on next page)

(continued from previous page)

```

22     sycl::accessor b_acc(b_buf, h, sycl::read_only);
23     // Output accessor
24     sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
25
26     h.parallel_for(num_items,
27                     [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
28   });
29
30   // kernel3
31   q.submit([&](auto &h) {
32     sycl::accessor sum_acc(sum_buf, h, sycl::read_write);
33     h.parallel_for(num_items, [=](auto id) {
34       if (sum_acc[id] > 10)
35         sum_acc[id] = 1;
36       else
37         sum_acc[id] = 0;
38     });
39   });
40
41   // kernel2
42   q.submit([&](auto &h) {
43     // Input accessors
44     sycl::accessor sum_acc(sum_buf, h, sycl::read_only);
45     sycl::accessor c_acc(c_buf, h, sycl::read_only);
46     sycl::accessor d_acc(d_buf, h, sycl::read_only);
47     // Output accessor
48     sycl::accessor res_acc(res_buf, h, sycl::write_only, sycl::no_init);
49
50     h.parallel_for(num_items, [=](auto i) {
51       res_acc[i] = sum_acc[i] * c_acc[i] + d_acc[i];
52     });
53   });
54   q.wait();
55 }
56 double elapsed = timer.Elapsed() / iter;
57 return (elapsed);
58 } // end myFunc2

```

There are other ways to optimize this example - for instance the clipping operation in **kernel3** can be merged into the computation of **kernel1** as shown below. This is kernel fusion and has the added advantage of not launching a third kernel. DPCPP compiler cannot do this kind of an optimization. In some specific domains like machine learning, there are graph compilers which operate on the ML models and fuse the operations which has the same impact.

#### **Listing 105:** /examples/host-device-memory/mem-move.cpp

```

1 double myFunc3(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b,
2                 AlignedVector<int> &c, AlignedVector<int> &d,
3                 AlignedVector<int> &res, int iter) {
4     sycl::range num_items{a.size()};
5     VectorAllocator<int> alloc;

```

(continues on next page)

(continued from previous page)

```
6 AlignedVector<int> sum(a.size(), alloc);
7
8 const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
9 sycl::buffer a_buf(a, props);
10 sycl::buffer b_buf(b, props);
11 sycl::buffer c_buf(b, props);
12 sycl::buffer d_buf(b, props);
13 sycl::buffer res_buf(res, props);
14 sycl::buffer sum_buf(sum.data(), num_items, props);
15
16 Timer timer;
17 for (int i = 0; i < iter; i++) {
18     // kernel1
19     q.submit([&](auto &h) {
20         // Input accessors
21         sycl::accessor a_acc(a_buf, h, sycl::read_only);
22         sycl::accessor b_acc(b_buf, h, sycl::read_only);
23         // Output accessor
24         sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
25
26         h.parallel_for(num_items, [=](auto i) {
27             int t = a_acc[i] + b_acc[i];
28             if (t > 10)
29                 sum_acc[i] = 1;
30             else
31                 sum_acc[i] = 0;
32         });
33     });
34
35     // kernel2
36     q.submit([&](auto &h) {
37         // Input accessors
38         sycl::accessor sum_acc(sum_buf, h, sycl::read_only);
39         sycl::accessor c_acc(c_buf, h, sycl::read_only);
40         sycl::accessor d_acc(d_buf, h, sycl::read_only);
41         // Output accessor
42         sycl::accessor res_acc(res_buf, h, sycl::write_only, sycl::no_init);
43
44         h.parallel_for(num_items, [=](auto i) {
45             res_acc[i] = sum_acc[i] * c_acc[i] + d_acc[i];
46         });
47     });
48     q.wait();
49 }
50 double elapsed = timer.Elapsed() / iter;
51 return (elapsed);
52 } // end myFunc3
```

We can take this kernel fusion one level further and fuse both **kernel1** and **kernel2** as shown in the code below. This gives very good performance since it avoids the intermediate **accum\_buf** completely saving memory in addition to launching an additional kernel. Most of the performance benefit in this case is due to improvement in locality of memory references.

**Listing 106:** /examples/host-device-memory/mem-move.cpp

```

1 double myFunc4(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b,
2                 AlignedVector<int> &c, AlignedVector<int> &d,
3                 AlignedVector<int> &res, int iter) {
4     sycl::range num_items{a.size()};
5     VectorAllocator<int> alloc;
6
7     const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
8     sycl::buffer a_buf(a, props);
9     sycl::buffer b_buf(b, props);
10    sycl::buffer c_buf(b, props);
11    sycl::buffer d_buf(b, props);
12    sycl::buffer res_buf(res, props);
13
14    Timer timer;
15    for (int i = 0; i < iter; i++) {
16        // kernel
17        q.submit([&](auto &h) {
18            // Input accessors
19            sycl::accessor a_acc(a_buf, h, sycl::read_only);
20            sycl::accessor b_acc(b_buf, h, sycl::read_only);
21            sycl::accessor c_acc(c_buf, h, sycl::read_only);
22            sycl::accessor d_acc(d_buf, h, sycl::read_only);
23            // Output accessor
24            sycl::accessor res_acc(res_buf, h, sycl::write_only, sycl::no_init);
25
26            h.parallel_for(num_items, [=](auto i) {
27                int t = a_acc[i] + b_acc[i];
28                if (t > 10)
29                    res_acc[i] = c_acc[i] + d_acc[i];
30                else
31                    res_acc[i] = d_acc[i];
32            });
33        });
34        q.wait();
35    }
36    double elapsed = timer.Elapsed() / iter;
37    return (elapsed);
38 } // end myFunc4

```

## 7.4 Avoid Declaring Buffers in a Loop

When kernels are repeatedly launched inside a for-loop, you can prevent repeated allocation and freeing of a buffer by declaring the buffer outside the loop. Declaring a buffer inside the loop introduces repeated host-to-device and device-to-host memory copies.

In the following example, the kernel is repeatedly launched inside a for-loop. The buffer C is used as a temporary array, where it is used to hold values in an iteration, and the values assigned in one iteration are not used in any other iteration. Since the buffer C is declared inside the for-loop, it is allocated and freed in every loop iteration. In addition to the allocation and freeing of the buffer, the memory associated with the buffer is redundantly

transferred from host to device and device to host in each iteration.

**Listing 107:** /examples/buffers/buf-kern1.cpp

```
1 #include <CL/sycl.hpp>
2 #include <stdio.h>
3
4 constexpr int N = 25;
5 constexpr int STEPS = 100000;
6
7 int main() {
8
9     int AData[N];
10    int BData[N];
11    int CData[N];
12
13    sycl::queue Q;
14
15    // Create 2 buffers, each holding N integers
16    sycl::buffer<int> ABuf(&AData[0], N);
17    sycl::buffer<int> BBuf(&BData[0], N);
18
19    Q.submit([&](auto &h) {
20        // Create device accessors.
21        // The property no_init lets the runtime know that the
22        // previous contents of the buffer can be discarded.
23        sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
24        sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
25        h.parallel_for(N, [=](auto i) {
26            aA[i] = 10;
27            aB[i] = 20;
28        });
29    });
30
31    for (int j = 0; j < STEPS; j++) {
32        sycl::buffer<int> CBuf(&CData[0], N);
33        Q.submit([&](auto &h) {
34            // Create device accessors.
35            sycl::accessor aA(ABuf, h);
36            sycl::accessor aB(BBuf, h);
37            sycl::accessor aC(CBuf, h);
38            h.parallel_for(N, [=](auto i) {
39                aC[i] = (aA[i] < aB[i]) ? -1 : 1;
40                aA[i] += aC[i];
41                aB[i] -= aC[i];
42            });
43        });
44    } // end for
45
46    // Create host accessors.
47    const sycl::host_accessor haA(ABuf);
48    const sycl::host_accessor haB(BBuf);
49    printf("%d %d\n", haA[N / 2], haB[N / 2]);
```

(continues on next page)

(continued from previous page)

```
50     return 0;  
51 }  
52 }
```

A better approach would be to declare the buffer C before the for-loop, so that it is allocated and freed only once, resulting in improved performance by avoiding the redundant data transfers between host and device. The following kernel shows this change.

**Listing 108:** /examples/buffers/buf-kern2.cpp

```
1 #include <CL/sycl.hpp>  
2 #include <stdio.h>  
3  
4 constexpr int N = 25;  
5 constexpr int STEPS = 100000;  
6  
7 int main() {  
8  
9     int AData[N];  
10    int BData[N];  
11    int CData[N];  
12  
13    sycl::queue Q;  
14  
15    // Create 3 buffers, each holding N integers  
16    sycl::buffer<int> ABuf(&AData[0], N);  
17    sycl::buffer<int> BBuf(&BData[0], N);  
18    sycl::buffer<int> CBuf(&CData[0], N);  
19  
20    Q.submit([&](auto &h) {  
21        // Create device accessors.  
22        // The property no_init lets the runtime know that the  
23        // previous contents of the buffer can be discarded.  
24        sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);  
25        sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);  
26        h.parallel_for(N, [=](auto i) {  
27            aA[i] = 10;  
28            aB[i] = 20;  
29        });  
30    });  
31  
32    for (int j = 0; j < STEPS; j++) {  
33        Q.submit([&](auto &h) {  
34            // Create device accessors.  
35            sycl::accessor aA(ABuf, h);  
36            sycl::accessor aB(BBuf, h);  
37            sycl::accessor aC(CBuf, h);  
38            h.parallel_for(N, [=](auto i) {  
39                aC[i] = (aA[i] < aB[i]) ? -1 : 1;  
40                aA[i] += aC[i];  
41                aB[i] -= aC[i];  
42            });  
43        });  
44    }  
45}
```

(continues on next page)

(continued from previous page)

```
42     });
43 };
44 } // end for
45
46 // Create host accessors.
47 const sycl::host_accessor haA(ABuf);
48 const sycl::host_accessor hab(BBuf);
49 printf("%d %d\n", haA[N / 2], hab[N / 2]);
50
51 return 0;
52 }
```

## 7.5 Buffer Accessor Modes

In DPC++, a buffer provides an abstract view of memory that can be accessed by the host or a device. A buffer cannot be accessed directly through the buffer object. Instead, we must create an accessor object that allows us to access the buffer's data.

The access mode describes how we intend to use the memory associated with the accessor in the program. Accessor's access modes are used by the runtime to create an execution order for the kernels and perform data movement. This will ensure that kernels are executed in an order intended by the programmer. Depending on the capabilities of the underlying hardware, the runtime can execute kernels concurrently if the dependencies do not give rise to dependency violations or race conditions.

For better performance, make sure that the access modes of accessors reflect the operations performed by the kernel. The compiler will flag an error when a write done on an accessor which is declared as `read_only`. But the compiler does not change the declaration of an accessor from `read_write` to `read` if no write is done in the kernel.

In the following example, there are three kernels. The first kernel initializes the A, B, and C buffers, so we specify that the access modes for these buffers is `write_only`. The second kernel reads the A and B buffers, and reads and writes the C buffer, so we specify that the access mode for the A and B buffers is `read_only`, and the access mode for the C buffer is `read_write`.

The `read_only` access mode informs the runtime that the data needs to be available on the device before the kernel can begin executing, but the data need not be copied from the device to the host at the end of the computation.

If this second kernel were to use `read_write` for A and B, instead of `read_only`, then the memory associated with A and B is copied from the device to the host at the end of kernel execution, even though the data has not been modified by the device. Moreover, `read_write` creates unnecessary dependencies. If another kernel that reads A or B is submitted within this block, this new kernel cannot start until the second kernel has completed.

**Listing 109:** /examples/buffer-accessors/kern1.cpp

```
1 #include <CL/sycl.hpp>
2 #include <stdio.h>
3
4 constexpr int N = 100;
```

(continues on next page)

(continued from previous page)

```
6 int main() {
7
8     int AData[N];
9     int BData[N];
10    int CData[N];
11
12    sycl::queue Q;
13
14    // Kernel1
15    {
16        // Create 3 buffers, each holding N integers
17        sycl::buffer<int> ABuf(&AData[0], N);
18        sycl::buffer<int> BBuf(&BData[0], N);
19        sycl::buffer<int> CBuf(&CData[0], N);
20
21        Q.submit([&](auto &h) {
22            // Create device accessors.
23            // The property no_init lets the runtime know that the
24            // previous contents of the buffer can be discarded.
25            sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
26            sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
27            sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);
28
29            h.parallel_for(N, [=](auto i) {
30                aA[i] = 11;
31                aB[i] = 22;
32                aC[i] = 0;
33            });
34        });
35    } // end Kernel1
36
37    // Kernel2
38    {
39        // Create 3 buffers, each holding N integers
40        sycl::buffer<int> ABuf(&AData[0], N);
41        sycl::buffer<int> BBuf(&BData[0], N);
42        sycl::buffer<int> CBuf(&CData[0], N);
43
44        Q.submit([&](auto &h) {
45            // Create device accessors
46            sycl::accessor aA(ABuf, h, sycl::read_only);
47            sycl::accessor aB(BBuf, h, sycl::read_only);
48            sycl::accessor aC(CBuf, h);
49            h.parallel_for(N, [=](auto i) { aC[i] += aA[i] + aB[i]; });
50        });
51    } // end Kernel2
52
53    // Buffers are destroyed and so CData is updated and can be accessed
54    for (int i = 0; i < N; i++) {
55        printf("%d\n", CData[i]);
56    }
}
```

(continues on next page)

(continued from previous page)

```
57     return 0;
58 }
59 }
```

Specifying `read_only` accessor mode, instead of `read_write`, is especially useful when kernels are repeatedly launched inside a for-loop. If the access mode is `read_write`, the kernels launched will be serialized as one kernel should finish its computation and the data should be ready before the next kernel can be launched. On the other hand, if the access mode is `read_only`, then the runtime can launch the kernels in parallel.

Note that the buffer declarations and kernels are launched inside a block. This will cause the buffers to go out of scope at the end of first kernel completion. This will trigger a copy of the contents from the device to the host. The second kernel is inside another block where new buffers are declared to the same memory and this will trigger a copy of this same memory again from the host to the device. This back-and-forth between host and device can be avoided by declaring the buffers once, ensuring that they are in scope during the lifetime of the memory pointed to by these buffers. A better way to write the code that avoids these unnecessary memory transfers is shown below.

**Listing 110:** /examples/buffer-accessors/kern2.cpp

```
1 #include <CL/sycl.hpp>
2 #include <stdio.h>
3
4 constexpr int N = 100;
5
6 int main() {
7
8     int AData[N];
9     int BData[N];
10    int CData[N];
11
12    sycl::queue Q;
13
14    // Create 3 buffers, each holding N integers
15    sycl::buffer<int> ABuf(&AData[0], N);
16    sycl::buffer<int> BBuf(&BData[0], N);
17    sycl::buffer<int> CBuf(&CData[0], N);
18
19    // Kernel1
20    Q.submit([&](auto &h) {
21        // Create device accessors.
22        // The property no_init lets the runtime know that the
23        // previous contents of the buffer can be discarded.
24        sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
25        sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
26        sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);
27
28        h.parallel_for(N, [=](auto i) {
29            aA[i] = 11;
30            aB[i] = 22;
31            aC[i] = 0;
32        });
33    });
34}
```

(continues on next page)

(continued from previous page)

```

33 });
34
35 // Kernel2
36 Q.submit([&](auto &h) {
37     // Create device sycl::accessors
38     sycl::accessor aA(ABuf, h, sycl::read_only);
39     sycl::accessor aB(BBuf, h, sycl::read_only);
40     sycl::accessor aC(CBuf, h);
41     h.parallel_for(N, [=](auto i) { aC[i] += aA[i] + aB[i]; });
42 });
43
44 // The host accessor creation will ensure that a wait for kernel to finish
45 // is triggered and data from device to host is copied
46 sycl::host_accessor h_acc(CBuf);
47 for (int i = 0; i < N; i++) {
48     printf("%d\n", h_acc[i]);
49 }
50
51 return 0;
52 }
```

The following example shows another way to run the same code with different scope blocking. In this case, there will not be a copy of buffers from host to device at the end of kernel1 and from host to device at the beginning of kernel2. The copy of all three buffers happens at the end of kernel2 when these buffers go out of scope.

**Listing 111:** /examples/buffer-accessors/kern3.cpp

```

1 #include <CL/sycl.hpp>
2 #include <stdio.h>
3
4 constexpr int N = 100;
5
6 int main() {
7
8     int AData[N];
9     int BData[N];
10    int CData[N];
11
12    sycl::queue Q;
13
14 {
15     // Create 3 buffers, each holding N integers
16     sycl::buffer<int> ABuf(&AData[0], N);
17     sycl::buffer<int> BBuf(&BData[0], N);
18     sycl::buffer<int> CBuf(&CData[0], N);
19
20     // Kernel1
21     Q.submit([&](auto &h) {
22         // Create device accessors.
23         // The property no_init lets the runtime know that the
24         // previous contents of the buffer can be discarded.
```

(continues on next page)

(continued from previous page)

```

25     sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
26     sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
27     sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);
28
29     h.parallel_for(N, [=](auto i) {
30         aA[i] = 11;
31         aB[i] = 22;
32         aC[i] = 0;
33     });
34 });
35
36 // Kernel2
37 Q.submit([&](auto &h) {
38     // Create device accessors
39     sycl::accessor aA(ABuf, h, sycl::read_only);
40     sycl::accessor aB(BBuf, h, sycl::read_only);
41     sycl::accessor aC(CBuf, h);
42     h.parallel_for(N, [=](auto i) { aC[i] += aA[i] + aB[i]; });
43 });
44 }
45 // Since the buffers are going out of scope, they will have to be
46 // copied back from device to host and this will require a wait for
47 // all the kernels to finish and so no explicit wait is needed
48 for (int i = 0; i < N; i++) {
49     printf("%d\n", CData[i]);
50 }
51
52 return 0;
53 }
```

There is another way to write the kernel where a copy of the read-only variable on the host can be accessed on the device as part of variable capture in the lambda function defining the kernel, as shown below. The issue with this is that for every kernel invocation the data associated with vectors AData and BData have to be copied to the device.

**Listing 112:** /examples/buffer-accessors/kern4.cpp

```

1 #include <CL/sycl.hpp>
2 #include <stdio.h>
3
4 constexpr int N = 100;
5 constexpr int iters = 100;
6
7 int main() {
8
9     int AData[N];
10    int BData[N];
11    int CData[N];
12
13    sycl::queue Q;
14    sycl::buffer<int> CBuf(&CData[0], N);
```

(continues on next page)

(continued from previous page)

```

15
16 {
17     // Create 2 buffers, each holding N integers
18     sycl::buffer<int> ABuf(&AData[0], N);
19     sycl::buffer<int> BBuf(&BData[0], N);
20
21     // Kernel1
22     Q.submit([&](auto &h) {
23         // Create device accessors.
24         // The property no_init lets the runtime know that the
25         // previous contents of the buffer can be discarded.
26         sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
27         sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
28         sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);
29
30         h.parallel_for(N, [=](auto i) {
31             aA[i] = 11;
32             aB[i] = 22;
33             aC[i] = 0;
34         });
35     });
36 }
37
38 for (int it = 0; it < iters; it++) {
39     // Kernel2
40     Q.submit([&](auto &h) {
41         // Create device accessors
42         sycl::accessor aC(CBuf, h);
43         h.parallel_for(N, [=](auto i) { aC[i] += AData[i] + BData[i]; });
44     });
45 }
46
47 sycl::host_accessor h_acc(CBuf);
48 for (int i = 0; i < N; i++) {
49     printf("%d\n", h_acc[i]);
50 }
51
52 return 0;
53 }
```

It is better to use a buffer and a read-only accessor to that buffer so that the vector is copied from host to device only once. In the following kernel access to memory AData and BData is made through the ABuf and Bbuf on lines 38 and 39 and the declaration in lines 44 and 45 makes them read-only which prevents them from being copied back to the host from the device when they go out of scope.

**Listing 113:** /examples/buffer-accessors/kern5.cpp

```

1 #include <CL/sycl.hpp>
2 #include <stdio.h>
3
4 constexpr int N = 100;
```

(continues on next page)

(continued from previous page)

```
5 constexpr int iters = 100;
6
7 int main() {
8
9     int AData[N];
10    int BData[N];
11    int CData[N];
12
13    sycl::queue Q;
14    sycl::buffer<int> CBuf(&CData[0], N);
15
16    {
17        // Create 2 buffers, each holding N integers
18        sycl::buffer<int> ABuf(&AData[0], N);
19        sycl::buffer<int> BBuf(&BData[0], N);
20
21        // Kernel1
22        Q.submit([&](auto &h) {
23            // Create device accessors.
24            // The property no_init lets the runtime know that the
25            // previous contents of the buffer can be discarded.
26            sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
27            sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
28            sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);
29
30            h.parallel_for(N, [=](auto i) {
31                aA[i] = 11;
32                aB[i] = 22;
33                aC[i] = 0;
34            });
35        });
36    }
37
38    sycl::buffer<int> ABuf(&AData[0], N);
39    sycl::buffer<int> BBuf(&BData[0], N);
40    for (int it = 0; it < iters; it++) {
41        // Kernel2
42        Q.submit([&](auto &h) {
43            // Create device accessors
44            sycl::accessor aA(ABuf, h, sycl::read_only);
45            sycl::accessor aB(BBuf, h, sycl::read_only);
46            sycl::accessor aC(CBuf, h);
47            h.parallel_for(N, [=](auto i) { aC[i] += aA[i] + aB[i]; });
48        });
49    }
50
51    sycl::host_accessor h_acc(CBuf);
52    for (int i = 0; i < N; i++) {
53        printf("%d\n", h_acc[i]);
54    }
55}
```

(continues on next page)

(continued from previous page)

```
56     return 0;  
57 }
```



## 8.0 Host/Device Coordination

Significant computation and communication resources exist between the host and accelerator devices, and care must be taken to ensure that they are effectively utilized. In this section, we cover topics related to the coordination of host and accelerator processing.

### 8.1 Asynchronous and Overlapping Data Transfers between Host and Device

An accelerator is separate device from the host CPU and is attached with some form of bus like PCIe or CXL. This bus, depending on its type, has a certain bandwidth through which the host and devices can transfer data. An accelerator needs some data from host to do computation, and overall performance of the system is dependent on how quickly this transfer can happen.

#### 8.1.1 Bandwidth between Host and Accelerator

Most current accelerators are connected to the host system through PCIe. Different generations of PCIe have increased the bandwidth over time, as shown in the table below.

**Table 9:** PCIe bandwidth by generation

PCIe Version	Transfer Rate	Throughput
1.0	2.5 GT/s	0.250 GB/s
2.0	5.0 GT/s	0.500 GB/s
3.0	8.0 GT/s	0.985 GB/s
4.0	16.0 GT/s	1.969 GB/s
5.0	32.0 GT/s	3.938 GB/s

The local memory bandwidth of an accelerator is an order of magnitude higher than host-to-device bandwidth over a link like PCIe. For instance, HBM (High Bandwidth Memory) on modern GPUs can reach up to 900 GB/sec of bandwidth compared to an x16 PCIe, which can get 63 GB/s. So, it is imperative to keep data in local memory and avoid data transfer from host-to-device or device-to-host as much as possible. This means that it is better to execute all the kernels on the accelerator to avoid data movement between accelerators or between host and accelerator even it means some kernels are not very efficiently executed on these accelerators.

Any intermediate data structures should be created and used on the device, as opposed to creating them on the host and moving them back and forth between host and accelerator. This is illustrated by the kernels shown here for reduction operations where the intermediate results are created only on the device and never on the host. In kernel **ComputeParallel1**, a temporary accumulator on is created on the host and all work-items put their intermediate results in it. This accumulator is brought back to the host and then further reduced (at line 37).

**Listing 114:** /examples/overlap-data-transfers/reduction.cpp

```
1 float ComputeParallel1(sycl::queue &q, std::vector<float> &data) {
2     const size_t data_size = data.size();
3     float sum = 0;
4     static float *accum = 0;
5
6     if (data_size > 0) {
7         const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
8         int num_EUs =
9             q.get_device().get_info<sycl::info::device::max_compute_units>();
10        int vec_size =
11            q.get_device()
12                .get_info<sycl::info::device::native_vector_width_float>();
13        int num_processing_elements = num_EUs * vec_size;
14        int BATCH = (N + num_processing_elements - 1) / num_processing_elements;
15        sycl::buffer<float> buf(data.data(), data.size(), props);
16        sycl::buffer<float> accum_buf(accum, num_processing_elements, props);
17
18        if (!accum)
19            accum = new float[num_processing_elements];
20
21        q.submit([&](auto &h) {
22            sycl::accessor buf_acc(buf, h, sycl::read_only);
23            sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
24            h.parallel_for(num_processing_elements, [=](auto index) {
25                size_t glob_id = index[0];
26                size_t start = glob_id * BATCH;
27                size_t end = (glob_id + 1) * BATCH;
28                if (end > N)
29                    end = N;
30                float sum = 0.0;
31                for (size_t i = start; i < end; i++)
32                    sum += buf_acc[i];
33                accum_acc[glob_id] = sum;
34            });
35        });
36        q.wait();
37        sycl::host_accessor h_acc(accum_buf);
38        for (int i = 0; i < num_processing_elements; i++)
39            sum += h_acc[i];
40    }
41    return sum;
42 } // end ComputeParallel1
```

An alternative approach is to keep this temporary accumulator on the accelerator and launch another kernel with only one work-item, which will perform this final reduction operation on the device as shown in the following **ComputeParallel2** kernel on line 36. Note that this kernel does not have much parallelism and so it is executed by just one work-item. On some platforms this might be better than transferring the data back to the host and doing the reduction there.

**Listing 115:** /examples/overlap-data-transfers/reduction.cpp

```
1 float ComputeParallel2(sycl::queue &q, std::vector<float> &data) {
2     const size_t data_size = data.size();
3     float sum = 0;
4     static float *accum = 0;
5
6     if (data_size > 0) {
7         const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
8         int num_EUs =
9             q.get_device().get_info<sycl::info::device::max_compute_units>();
10        int vec_size =
11            q.get_device()
12                .get_info<sycl::info::device::native_vector_width_float>();
13        int num_processing_elements = num_EUs * vec_size;
14        int BATCH = (N + num_processing_elements - 1) / num_processing_elements;
15        sycl::buffer<float> buf(data.data(), data.size(), props);
16        sycl::buffer<float> accum_buf(accum, num_processing_elements, props);
17        sycl::buffer<float> res_buf(&sum, 1, props);
18        if (!accum)
19            accum = new float[num_processing_elements];
20
21        q.submit([&](auto &h) {
22            sycl::accessor buf_acc(buf, h, sycl::read_only);
23            sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
24            h.parallel_for(num_processing_elements, [=](auto index) {
25                size_t glob_id = index[0];
26                size_t start = glob_id * BATCH;
27                size_t end = (glob_id + 1) * BATCH;
28                if (end > N)
29                    end = N;
30                float sum = 0.0;
31                for (size_t i = start; i < end; i++)
32                    sum += buf_acc[i];
33                accum_acc[glob_id] = sum;
34            });
35        });
36
37        q.submit([&](auto &h) {
38            sycl::accessor accum_acc(accum, h, sycl::read_only);
39            sycl::accessor res_acc(res_buf, h, sycl::write_only, sycl::no_init);
40            h.parallel_for(1, [=](auto index) {
41                res_acc[index] = 0;
42                for (size_t i = 0; i < num_processing_elements; i++)
43                    res_acc[index] += accum_acc[i];
44            });
45        });
46    }
47    // Buffers go out of scope and data gets transferred from device to host
48    return sum;
49 } // end ComputeParallel2
```

## 8.1.2 Overlapping Data Transfer from Host to Device with Computation on Device

Some GPUs provide specialized engines for copying data from host to device. Effective utilization of them will ensure that the host-to-device data transfer can be overlapped with execution on the device. In the following example, a block of memory is divided into chunks and each chunk is transferred to the accelerator (line 35), processed (line 38), and the result (line 44) is brought back to the host. These chunks of three tasks are independent so they can be processed in parallel depending on availability of hardware resources. In systems where there is a copy engine that can be used to transfer data between host and device, we can see that the kernels from different loop iterations can execute in parallel. The parallel execution can manifest in two ways:

- Between two memory copies, where one is executed by the GPU EU's and one by the copy engine
- Between a memory copy and a compute kernel, where the memory copy is executed by the copy engine and the compute kernel by the GPU EU's.

**Listing 116:** /examples/overlap-data-transfers/overlap.cpp

```
1 #include <CL/sycl.hpp>
2
3 class Timer {
4 public:
5     Timer() : start_(std::chrono::steady_clock::now()) {}
6
7     double Elapsed() {
8         auto now = std::chrono::steady_clock::now();
9         return std::chrono::duration_cast<Duration>(now - start_).count();
10    }
11
12 private:
13     using Duration = std::chrono::duration<double>;
14     std::chrono::steady_clock::time_point start_;
15 };
16
17 int main() {
18     const int num_chunks = 10;
19     const int chunk_size = 1000000;
20     const int iter = 10;
21
22     sycl::queue q;
23
24     // Allocate and initialize host data
25     float *host_data[num_chunks];
26     for (int c = 0; c < num_chunks; c++) {
27         host_data[c] = new float[chunk_size];
28         float val = c;
29         for (int i = 0; i < chunk_size; i++)
30             host_data[c][i] = val;
31     }
32     std::cout << "Allocated host data\n";
33
34     // Allocate and initialize device memory
35     float *device_data[num_chunks];
36     for (int c = 0; c < num_chunks; c++) {
```

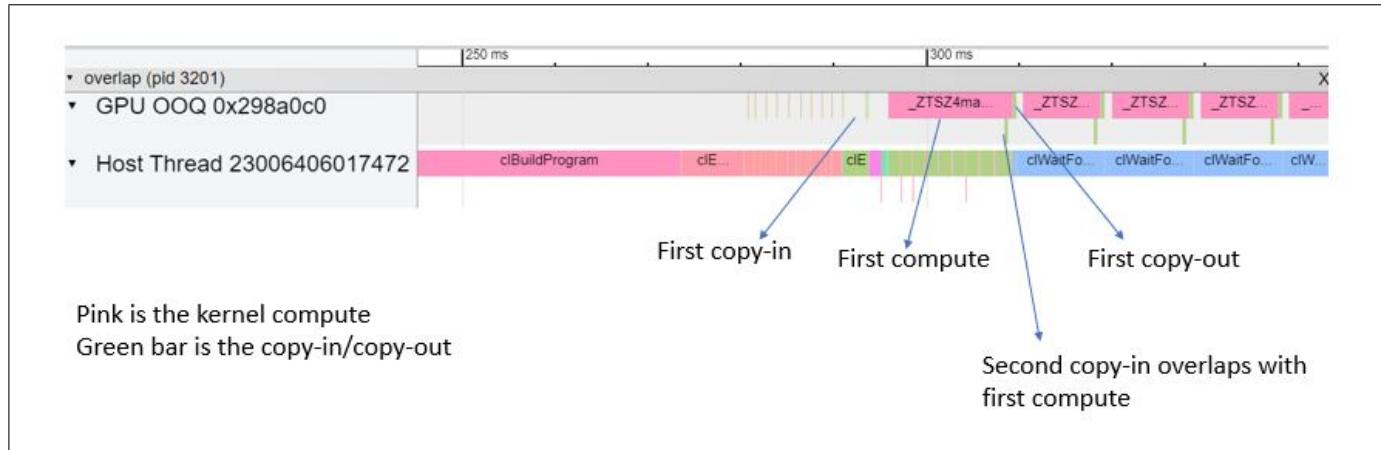
(continues on next page)

(continued from previous page)

```

37 device_data[c] = sycl::malloc_device<float>(chunk_size, q);
38     float val = 1000.0;
39     q.fill<float>(device_data[c], val, chunk_size);
40 }
41 q.wait();
42 std::cout << "Allocated device data\n";
43
44 Timer timer;
45 for (int it = 0; it < iter; it++) {
46     for (int c = 0; c < num_chunks; c++) {
47         auto add_one = [=](auto id) {
48             for (int i = 0; i < 10000; i++)
49                 device_data[c][id] += 1.0;
50         };
51         // Copy-in not dependent on previous event
52         auto copy_in =
53             q.memcpy(device_data[c], host_data[c], sizeof(float) * chunk_size);
54         // Compute waits for copy_in
55         auto compute = q.parallel_for(chunk_size, copy_in, add_one);
56         auto cg = [=](auto &h) {
57             h.depends_on(compute);
58             h.memcpy(host_data[c], device_data[c], sizeof(float) * chunk_size);
59         };
60         // Copy out waits for compute
61         auto copy_out = q.submit(cg);
62     }
63
64     q.wait();
65 }
66 auto elapsed = timer.Elapsed() / iter;
67 for (int c = 0; c < num_chunks; c++) {
68     for (int i = 0; i < chunk_size; i++) {
69         if (host_data[c][i] != (float)((c + 10000 * iter))) {
70             std::cout << "Mismatch for chunk: " << c << " position: " << i
71                 << " expected: " << c + 10000 << " got: " << host_data[c][i]
72                 << "\n";
73             break;
74         }
75     }
76 }
77 std::cout << "Time = " << elapsed << " usecs\n";
78 }
```

In the timeline picture below, which is collected using **clIntercept**, we can see that copy-in is overlapping with the execution of compute kernel.



**Fig. 17:** clIntercept showing copy-in overlap with execution of compute kernel

In the example above, we cannot have two kernels (even though they are independent) executing concurrently because we only have one GPU (it is possible to partition the GPU into smaller chunks and execute different kernels concurrently on them).

## 9.0 Using multiple heterogeneous devices

Most accelerators typically reside in a server that has a significant amount of compute resources in it. For instance, a typical server can have up to eight sockets with each socket containing over 50 cores. DPC++ provides the ability to treat the CPUs and the accelerators uniformly to distribute work among them. It is the responsibility of the programmer to ensure a balanced distribution of work among the heterogeneous compute resources in the platform.

### 9.1 Overlapping Compute on Various Accelerators in the Platform

DPC++ provides access to different kinds of devices through abstraction of device selectors. Queues can be created for each of the devices, and kernels can be submitted to them for execution. All kernel submits in DPC++ are non-blocking, which means that once the kernel is submitted to a queue for execution, the host does not wait for it to finish unless waiting on the queue is explicitly requested. This allows the host to do some work itself or initiate work on other devices while the kernel is executing on the accelerator.

The host CPU can be treated as an accelerator and the DPCPP can submit kernels to it for execution. This is completely independent and orthogonal to the job done by the host to orchestrate the kernel submission and creation. The underlying operating system manages the kernels submitted to the CPU accelerator as another process and uses the same `openCL/Level0` runtime mechanisms to exchange information with the host device.

In the following example, a simple vector add operation is shown that works on a single GPU device.

**Listing 117:** /examples/multiple-devices/overlap.cpp

```

1 int VectorAdd1(sycl::queue &q, const IntArray &a, const IntArray &b,
2                 IntArray &sum, int iter) {
3     sycl::range num_items{a.size()};
4
5     sycl::buffer a_buf(a);
6     sycl::buffer b_buf(b);
7     sycl::buffer sum_buf(sum.data(), num_items);
8     auto start = std::chrono::steady_clock::now();
9     for (int i = 0; i < iter; i++) {
10
11         auto e = q.submit([&](auto &h) {
12             // Input accessors
13             sycl::accessor a_acc(a_buf, h, sycl::read_only);
14             sycl::accessor b_acc(b_buf, h, sycl::read_only);
15             // Output accessor
16             sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
17
18             h.parallel_for(num_items,
19                           [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
20         });
21     }
22     q.wait();

```

(continues on next page)

(continued from previous page)

```

23 auto end = std::chrono::steady_clock::now();
24 std::cout << "Vector add1 completed on device - took "
25         << (end - start).count() << " u-secs\n";
26 return ((end - start).count());
27 } // end VectorAdd1

```

In the following kernel the input vector is split into two parts and computation is done on two different accelerators (one is a CPU and the other is a GPU) that can execute concurrently. Care must be taken to ensure that the kernels, in addition to being submitted, are actually launched on the devices to get this parallelism. The actual time that a kernel is launched can be substantially later than when it was submitted by the host. The implementation decides the time to launch the kernels based on some heuristics to maximize some metric like utilization, throughput or latency. For instance, in the case of the OpenCL backend, on certain platforms one needs to explicitly issue a `clFlush` (as shown on line 41) on the queue to launch the kernels on the accelerators.

**Listing 118:** /examples/multiple-devices/overlap.cpp

```

1 int VectorAdd2(sycl::queue &q1, sycl::queue &q2, const IntArray &a,
2                 const IntArray &b, IntArray &sum, int iter) {
3     sycl::range num_items{a.size() / 2};
4
5     auto start = std::chrono::steady_clock::now();
6     {
7         sycl::buffer a1_buf(a.data(), num_items);
8         sycl::buffer b1_buf(b.data(), num_items);
9         sycl::buffer sum1_buf(sum.data(), num_items);
10
11         sycl::buffer a2_buf(a.data() + a.size() / 2, num_items);
12         sycl::buffer b2_buf(b.data() + a.size() / 2, num_items);
13         sycl::buffer sum2_buf(sum.data() + a.size() / 2, num_items);
14         for (int i = 0; i < iter; i++) {
15
16             q1.submit([&](auto &h) {
17                 // Input accessors
18                 sycl::accessor a_acc(a1_buf, h, sycl::read_only);
19                 sycl::accessor b_acc(b1_buf, h, sycl::read_only);
20                 // Output accessor
21                 sycl::accessor sum_acc(sum1_buf, h, sycl::write_only, sycl::no_init);
22
23                 h.parallel_for(num_items,
24                               [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
25             });
26             // do the work on host
27             q2.submit([&](auto &h) {
28                 // Input accessors
29                 sycl::accessor a_acc(a2_buf, h, sycl::read_only);
30                 sycl::accessor b_acc(b2_buf, h, sycl::read_only);
31                 // Output accessor
32                 sycl::accessor sum_acc(sum2_buf, h, sycl::write_only, sycl::no_init);
33
34                 h.parallel_for(num_items,
35                               [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
36             });
37         }
38     }
39 }

```

(continues on next page)

(continued from previous page)

```
36     });
37 }
38 // On some platforms this explicit flush of queues is needed
39 // to ensure the overlap in execution between the CPU and GPU
40 // cl_command_queue cq = q1.get();
41 // clFlush(cq);
42 // cq=q2.get();
43 // clFlush(cq);
44 }
45 q1.wait();
46 q2.wait();
47 auto end = std::chrono::steady_clock::now();
48 std::cout << "Vector add2 completed on device - took "
49     << (end - start).count() << " u-secs\n";
50 return ((end - start).count());
51 } // end VectorAdd2
```

Checking the running time of the above two kernels it can be seen that the application runs almost twice as fast as before since it has more hardware resources dedicated to solving the problem. In order to achieve good balance one will have to split the work proportional to the capability of the accelerator instead of distributing it evenly as was done in the above example.



## 10.0 Compilation

In oneAPI, there are multiple types of compilation. The main source to the application is compiled. The offloaded kernels are compiled. For the kernels, this might be ahead-of-time (AOT) or just-in-time (JIT).

In this section we cover topics related to this compilation and how it can impact the efficiency of the execution.

### 10.1 Just-In-Time Compilation in DPC++

The Intel® oneAPI DPC++ Compiler converts a DPC++ program into an intermediate language called SPIR-V and stores that in the binary produced by the compilation process. The advantage of producing this intermediate file instead of the binary is that this code can be run on any hardware platform by translating the SPIR-V code into the assembly code of the platform at runtime. This process of translating the intermediate code present in the binary is called JIT compilation (just-in-time compilation). JIT compilation can happen on demand at runtime. There are multiple ways in which this JIT compilation can be controlled. By default, all the SPIR-V code present in the binary is translated upfront at the beginning of the execution of the first offloaded kernel.

**Listing 119:** /examples/jitting/jit.cpp

```

1 #include <CL/sycl.hpp>
2 #include <array>
3 #include <chrono>
4 #include <iostream>
5
6 // Array type and data size for this example.
7 constexpr size_t array_size = (1 << 16);
8 typedef std::array<int, array_size> IntArray;
9
10 void VectorAddl(sycl::queue &q, const IntArray &a, const IntArray &b,
11                  IntArray &sum) {
12     sycl::range num_items{a.size()};
13
14     sycl::buffer a_buf(a);
15     sycl::buffer b_buf(b);
16     sycl::buffer sum_buf(sum.data(), num_items);
17
18     auto e = q.submit([&](auto &h) {
19         // Input accessors
20         sycl::accessor a_acc(a_buf, h, sycl::read_only);
21         sycl::accessor b_acc(b_buf, h, sycl::read_only);
22         // Output accessor
23         sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
24
25         h.parallel_for(num_items,
26                         [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
27     });
28     q.wait();

```

(continues on next page)

(continued from previous page)

```
29 }
30
31 void VectorAdd2(sycl::queue &q, const IntArray &a, const IntArray &b,
32                 IntArray &sum) {
33     sycl::range num_items{a.size()};
34
35     sycl::buffer a_buf(a);
36     sycl::buffer b_buf(b);
37     sycl::buffer sum_buf(sum.data(), num_items);
38
39     auto e = q.submit([&](auto &h) {
40         // Input accessors
41         sycl::accessor a_acc(a_buf, h, sycl::read_only);
42         sycl::accessor b_acc(b_buf, h, sycl::read_only);
43         // Output accessor
44         sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
45
46         h.parallel_for(num_items,
47                         [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
48     });
49     q.wait();
50 }
51
52 void InitializeArray(IntArray &a) {
53     for (size_t i = 0; i < a.size(); i++)
54         a[i] = i;
55 }
56
57 int main() {
58     sycl::default_selector d_selector;
59
60     IntArray a, b, sum;
61
62     InitializeArray(a);
63     InitializeArray(b);
64
65     sycl::queue q(d_selector, sycl::property::queue::enable_profiling{});
66
67     std::cout << "Running on device: "
68             << q.get_device().get_info<sycl::info::device::name>() << "\n";
69     std::cout << "Vector size: " << a.size() << "\n";
70     auto start = std::chrono::steady_clock::now();
71     VectorAdd1(q, a, b, sum);
72     auto end = std::chrono::steady_clock::now();
73     std::cout << "Initial Vector add1 successfully completed on device - took "
74             << (end - start).count() << " nano-secs\n";
75
76     start = std::chrono::steady_clock::now();
77     VectorAdd1(q, a, b, sum);
78     end = std::chrono::steady_clock::now();
79     std::cout << "Second Vector add1 successfully completed on device - took "
```

(continues on next page)

(continued from previous page)

```

80     << (end - start).count() << " nano-secs\n";
81
82     start = std::chrono::steady_clock::now();
83     VectorAdd2(q, a, b, sum);
84     end = std::chrono::steady_clock::now();
85     std::cout << "Initial Vector add2 successfully completed on device - took "
86             << (end - start).count() << " nano-secs\n";
87
88     start = std::chrono::steady_clock::now();
89     VectorAdd2(q, a, b, sum);
90     end = std::chrono::steady_clock::now();
91     std::cout << "Second Vector add2 successfully completed on device - took "
92             << (end - start).count() << " nano-secs\n";
93     return 0;
94 }
```

When the program above is compiled using the command below (assuming that the name of the source file is `example.cpp`):

```
dpcpp -O3 -o example example.cpp
```

and run, the output generated will show that the first call to `VectorAdd1` takes much longer than the calls to other kernels in the program due to the cost of JIT compilation, which gets invoked when `vectorAdd1` is executed for the first time.

The overhead of JIT compilation at runtime can be avoided by ahead-of-time (AOT) compilation (it is enabled by appropriate switches on the compile-line). With AOT compile, the binary will contain the actual assembly code of the platform that was selected during compilation instead of the SPIR-V intermediate code. The advantage is that we do not need to JIT compile the code from SPIR-V to assembly during execution, which makes the code run faster. The disadvantage is that now the code cannot run anywhere other than the platform for which it was compiled.

The example above can be compiled on a Gen9 GPU using the following command with AOT code-generation:

```
dpcpp -O3 -o example example.cpp -fsycl-targets=spir64_gen-unknown-unknown-sycldevice -Xsycl-
    ↵target-backend=spir64_gen-unknown-unknown-sycldevice "-device skl"
```

When this compiled program is run, it can be seen from the output that the time it takes to execute all the calls to the kernels takes the same amount of time, unlike before where the first kernel takes a lot more time because of JIT compilation.

If the application contains multiple kernels, one can force eager JIT compilation or lazy JIT compilation using compile-time switches. Eager JIT compilation will invoke the JITter on all the kernels in the binary at the beginning of execution, while lazy JIT compilation will enable the JITter only when the kernel is actually called during execution. In situations where certain kernels are not called, this has the advantage of not translating code that is never actually executed, which avoids unnecessary JIT compilation. This mode can be enabled during compilation using the following option:

```
-fsycl-device-code-split=<value>
```

where value is

- **per\_kernel**: generates code to do JIT compilation of a kernel only when it is called
- **per\_source**: generates code to do JIT compilation of all kernels in the source file when any of the kernels in the source file are called
- **off**: the default, which does eager JIT compilation of all kernels in the application

If the above program is compiled with this option:

```
dpcpp -O3 -o example vec1.cpp vec2.cpp main.cpp -fsycl-device-code-split=per_kernel
```

and run, then from the timings of the kernel executions it can be seen that the first invocations of VectorAdd1 and VectorAdd2 take longer, while the second invocations will take less time because they do not pay the cost of JIT compilation.

In the example above, we can put VectorAdd1 and VectorAdd2 in separate files and compile them with and without the **per\_source** option to see the impact on the execution times of the kernels. When compiled with

```
dpcpp -O3 -o example vec1.cpp vec2.cpp main.cpp -fsycl-device-code-split=per_source
```

and run, the execution times of the kernels will show that the JIT compilation cost is paid at the first kernel invocation, while the subsequent kernel invocations do not pay the JIT compilation cost. But when the program is compiled with

```
dpcpp -O3 -o example vec1.cpp vec2.cpp main.cpp
```

and run, the execution times of the kernels will show that the JIT compilation cost is paid upfront at the first invocation of the kernel and all subsequent kernels do not pay the cost of JIT compilation.

## 10.2 Specialization Constants

DPC++ has a feature called **specialization constants** that can explicitly trigger JIT compilation to generate code from the intermediate SPIR-V code based on the run-time values of these specialization constants. These JIT compilation actions are done during the execution of the program when the values of these constants are known. This is different from the JIT compilation, which is triggered based on the options provided to **-fsycl-device-code-split**.

In the example below, the call to `set_specialization_constant` binds the value returned by the call to function `get_value`, defined on line 10, to the SYCL kernel bundle. When the kernel bundle is initially compiled, this value is not known and so cannot be used for optimizations. At runtime, after function `get_value` is executed, the value is known, so it is used by command groups handler to trigger JIT compilation of the specialized kernel with this value.

**Listing 120:** /examples/jitting/spec-const1.cpp

```
1 #include <CL/sycl.hpp>
2 #include <vector>
3
4 class specialized_kernel;
```

(continues on next page)

(continued from previous page)

```

6 // const static identifier of specialization constant
7 const static sycl::specialization_id<float> value_id;
8
9 // Fetch a value at runtime.
10 float get_value() { return 10; };
11
12 int main() {
13     sycl::queue queue;
14
15     std::vector<float> vec(1);
16     {
17         sycl::buffer<float> buffer(vec.data(), vec.size());
18         queue.submit([&](auto &cgh) {
19             sycl::accessor acc(buffer, cgh, sycl::write_only, sycl::no_init);
20
21             // Set value of specialization constant.
22             cgh.template set_specialization_constant<value_id>(get_value());
23
24             // Runtime builds the kernel with specialization constant
25             // replaced by the literal value provided in the preceding
26             // call of `set_specialization_constant<value_id>`
27             cgh.template single_task<specialized_kernel>(
28                 [=](sycl::kernel_handler kh) {
29                     const float val = kh.get_specialization_constant<value_id>();
30                     acc[0] = val;
31                 });
32         });
33     }
34     queue.wait_and_throw();
35
36     std::cout << vec[0] << std::endl;
37
38     return 0;
39 }
```

The specialized kernel at line 24 will eventually become the code shown below:

```
cgh.single_task<specialized_kernel>(
    [=]() { acc[0] = 10; });
```

This JIT compilation also has an impact on the amount of time it takes to execute a kernel. This is illustrated by the example below:

**Listing 121:** /examples/jitting/spec-const2.cpp

```

1 #include <CL/sycl.hpp>
2 #include <chrono>
3 #include <vector>
4
5 class specialized_kernel;
6 class literal_kernel;
```

(continues on next page)

(continued from previous page)

```
7 // const static identifier of specialization constant
8 const static sycl::specialization_id<float> value_id;
9
10 // Fetch a value at runtime.
11 float get_value() { return 10; }
12
13 int main() {
14     sycl::queue queue;
15
16     // Get kernel ID from kernel class qualifier
17     sycl::kernel_id specialized_kernel_id =
18         sycl::get_kernel_id<specialized_kernel>();
19
20     // Construct kernel bundle with only specialized_kernel in the input state
21     sycl::kernel_bundle kb_src =
22         sycl::get_kernel_bundle<sycl::bundle_state::input>(
23             queue.get_context(), {specialized_kernel_id});
24     // set specialization constant value
25     kb_src.set_specialization_constant<value_id>(get_value());
26
27     auto start = std::chrono::steady_clock::now();
28     // build the kernel bundle for the set value
29     sycl::kernel_bundle kb_exe = sycl::build(kb_src);
30     auto end = std::chrono::steady_clock::now();
31     std::cout << "specialization took - " << (end - start).count()
32             << " nano-secs\n";
33
34     std::vector<float> vec{0, 0, 0, 0, 0};
35     sycl::buffer<float> buffer1(vec.data(), vec.size());
36     sycl::buffer<float> buffer2(vec.data(), vec.size());
37     start = std::chrono::steady_clock::now();
38
39     {
40         queue.submit([&](auto &cgh) {
41             sycl::accessor acc(buffer1, cgh, sycl::write_only, sycl::no_init);
42
43             // use the precompiled kernel bundle in the executable state
44             cgh.use_kernel_bundle(kb_exe);
45
46             cgh.template single_task<specialized_kernel>(
47                 [=](sycl::kernel_handler kh) {
48                     float v = kh.get_specialization_constant<value_id>();
49                     acc[0] = v;
50                 });
51         });
52         queue.wait_and_throw();
53     }
54     end = std::chrono::steady_clock::now();
55
56     {
57         sycl::host_accessor host_acc(buffer1, sycl::read_only);
```

(continues on next page)

(continued from previous page)

```
58     std::cout << "result1 (c): " << host_acc[0] << " " << host_acc[1] << " "
59         << host_acc[2] << " " << host_acc[3] << " " << host_acc[4]
60         << std::endl;
61     }
62     std::cout << "execution took : " << (end - start).count() << " nano-secs\n";
63
64     start = std::chrono::steady_clock::now();
65     {
66         queue.submit([&](auto &cgh) {
67             sycl::accessor acc(buffer2, cgh, sycl::write_only, sycl::no_init);
68             cgh.template single_task<literal_kernel>([=]() { acc[0] = 20; });
69         });
70         queue.wait_and_throw();
71     }
72     end = std::chrono::steady_clock::now();
73
74     {
75         sycl::host_accessor host_acc(buffer2, sycl::read_only);
76         std::cout << "result2 (c): " << host_acc[0] << " " << host_acc[1] << " "
77             << host_acc[2] << " " << host_acc[3] << " " << host_acc[4]
78             << std::endl;
79     }
80     std::cout << "execution took - " << (end - start).count() << " nano-secs\n";
81 }
```

Looking the runtimes reported by each of the timing messages it can be seen that the initial translation of the kernel takes a long time, while the actual execution of the JIT-compiled kernel takes less time. The same kernel which had not been precompiled to the executable state takes longer time because this kernel will have been JIT-compiled by the runtime before actually executing it.



## 11.0 Debugging and Profiling

Understanding the behavior of your system is critical to making informed decisions about optimization choices. Some tools like profilers, analyzers, or debuggers are full featured. Other tools can be lighter weight like interval timers, kernel timers, and print statements. But all of them serve an important purpose in aiding the optimization process.

In this section we cover topics related to these tools usages for software optimization.

### 11.1 GPU Analysis with Intel® VTune™ Profiler

Intel® VTune™ Profiler is a performance analysis tool for serial and multi-threaded applications. It helps you analyze algorithm choices and identify where and how your application can benefit from available hardware resources. Use it to locate or determine:

- Sections of code that don't effectively utilize available processor resources
- The best sections of code to optimize for both sequential and threaded performance
- Synchronization objects that affect the application performance
- Whether, where, and why your application spends time on input/output operations
- Whether your application is CPU-bound or GPU-bound and how effectively it offloads code to the GPU
- The performance impact of different synchronization methods, different numbers of threads, or different algorithms
- Thread activity and transitions
- Hardware-related issues in your code such as data sharing, cache misses, branch misprediction, and others
- Profiling a DPC++ Application running on a GPU

The tool also has new features to support GPU analysis:

- GPU Offload Analysis (technical preview)
- GPU Compute/Media Hotspots Analysis (technical preview)

#### GPU Offload Analysis (Preview)

Use this analysis type to analyze code execution on the CPU and GPU cores of your platform, correlate CPU and GPU activity, and identify whether your application is GPU-bound or CPU-bound. The tool infrastructure automatically aligns clocks across all cores in the system so you can analyze some CPU-based workloads together with GPU-based workloads within a unified time domain. This analysis lets you:

- Identify how effectively your application uses DPC++ or OpenCL™ kernels.
- Analyze execution of Intel® Media SDK tasks over time (for Linux targets only).

- Explore GPU usage and analyze a software queue for GPU engines at each moment of time.

### GPU Compute/Media Hotspots Analysis (Preview)

Use this tool to analyze the most time-consuming GPU kernels, characterize GPU usage based on GPU hardware metrics, identify performance issues caused by memory latency or inefficient kernel algorithms, and analyze GPU instruction frequency for certain instruction types. The GPU Compute/Media Hotspots analysis allows you to:

- Explore GPU kernels with high GPU utilization, estimate the efficiency of this utilization, and identify possible reasons for stalls or low occupancy.
- Explore the performance of your application per selected GPU metrics over time.
- Analyze the hottest DPC++ or OpenCL kernels for inefficient kernel code algorithms or incorrect work item configuration.
- Run GPU Offload Analysis on a DPC++ Application.

### How to use VTune Profiler to analyze GPU applications

1. Launch VTune Profiler and click **New Project** from the Welcome page. The Create a Project dialog box opens.
2. Specify a project name and a location for your project and click **Create Project**. The Configure Analysis window opens.
3. Make sure the Local Host is selected in the WHERE pane.
4. In the WHAT pane, make sure the Launch Application target is selected and specify the `matrix_multiply` binary as an Application to profile.
5. In the HOW pane, select GPU Offload analysis type from the Accelerators group.

This is the least intrusive analysis for applications running on platforms with Intel Graphics as well as on other third-party GPUs supported by VTune Profiler. Click the **Start** button to launch the analysis.

### Run Analysis from Command Line:

To run the analysis from the command line:

On Linux\* OS:

1. Set VTune Profiler environment variables by sourcing the script:

```
source <install_dir>/env/vars.sh
```

2. Run the analysis command:

```
vtune -collect gpu-offload -- ./matrix.dpcpp
```

On Windows\* OS:

1. Set VTune Profiler environment variables by running the batch file:

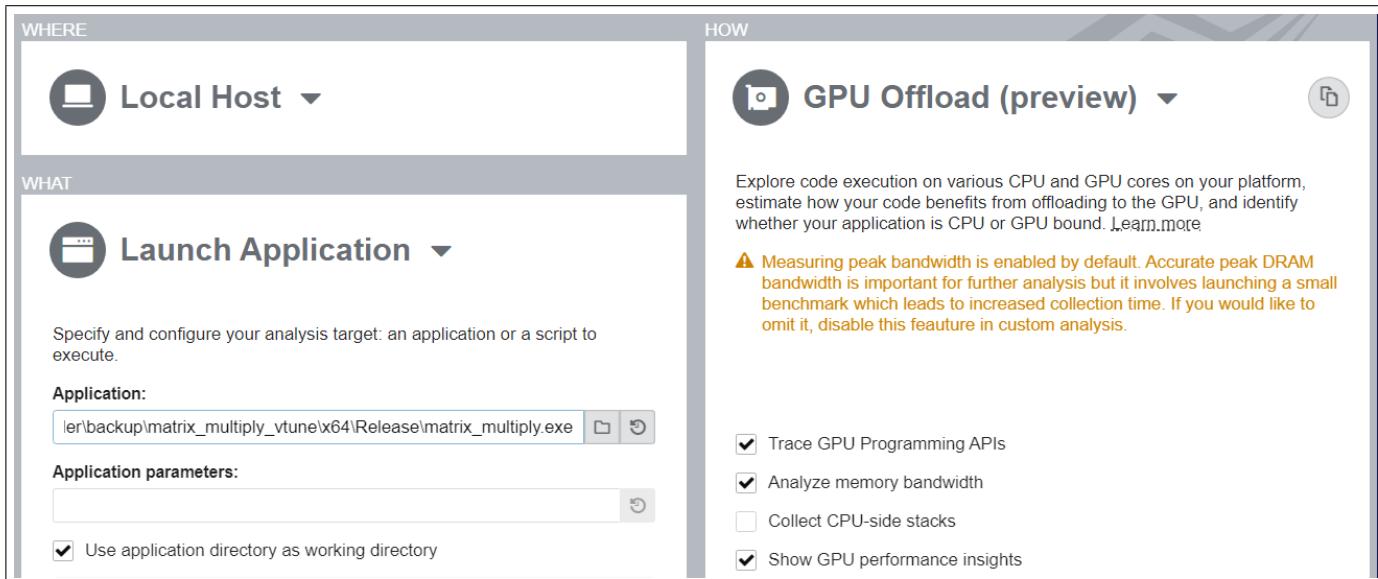
```
export <install_dir>\env\vars.bat
```

2. Run the analysis command:

```
vtune.exe -collect gpu-offload -- matrix_multiply.exe
```

### Analyze Collected Data

Start your analysis with the GPU Offload viewpoint. In the Summary window, you can see statistics on CPU and GPU resource usage to determine if your application is GPU-bound, CPU-bound, or not effectively utilizing the compute capabilities of the system. In this example, the application should use the GPU for intensive computation. However, the result summary informs that GPU usage is actually low.



**Fig. 18:** GPU Offload viewpoint

Switch to the Platform window. Here, you can see basic CPU and GPU metrics that help analyze GPU usage on a software queue. This data is correlated with CPU usage on the timeline. The information in the Platform window can help you make some inferences.

## Elapsed Time : 7.840s

### GPU Utilization : 7.3%

Use this section to understand whether the GPU was utilized properly and which of the engines were utilized. Identify the amount of gaps in the GPU utilization that potentially could be loaded with some work. This metric is calculated for the engines that had at least one piece of work scheduled to them.

#### GPU Utilization

GPU Utilization breakdown by GPU engines and work types.

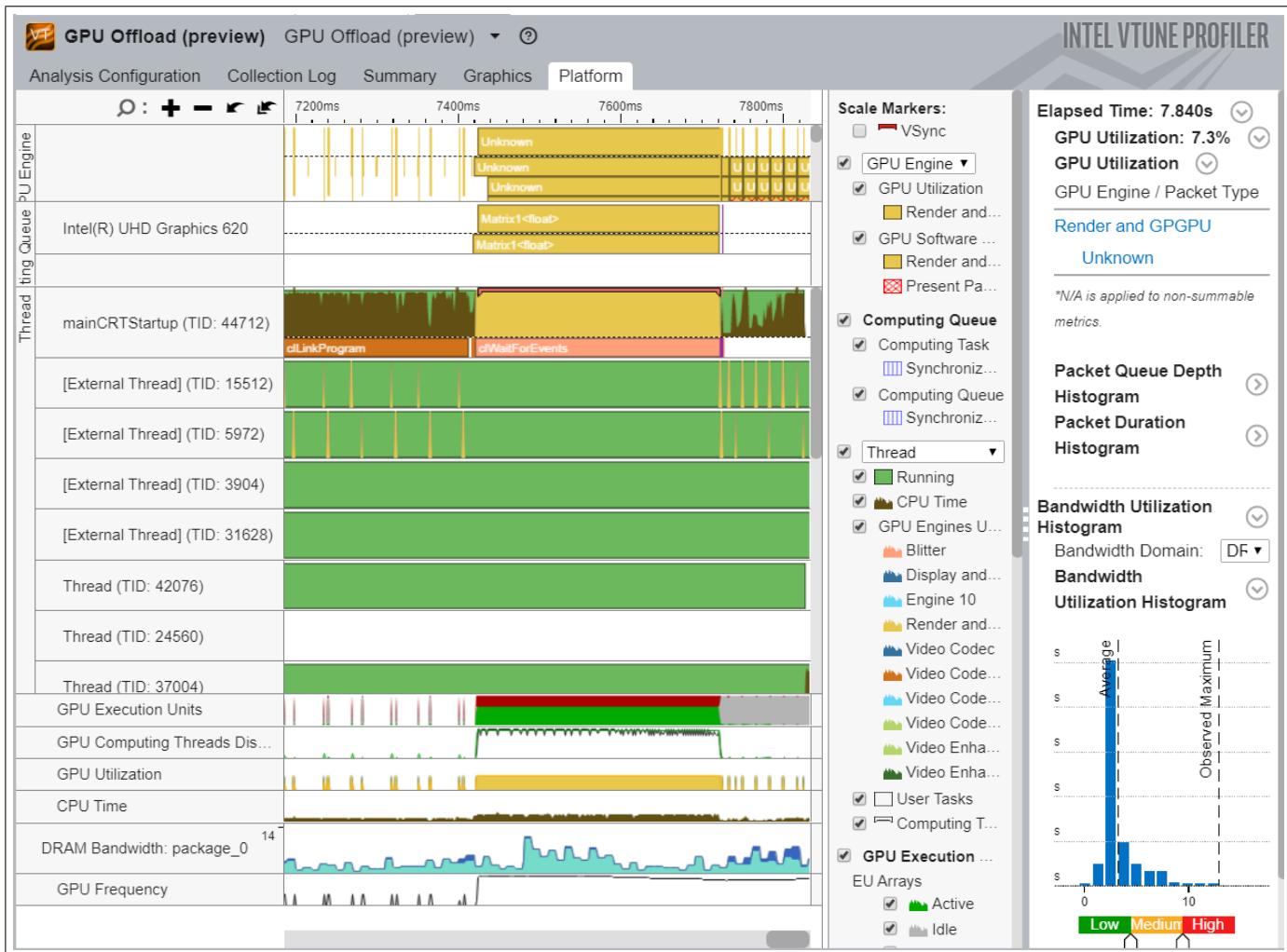
GPU Engine / Packet Type	GPU Time	GPU Utilization 
Render and GPGPU	0.570s	7.3% 
Unknown	0.570s	7.3%

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

**Fig.19:** GPU Utilization

Most applications may not present obvious situations as described above. A detailed analysis is important to understand all dependencies. For example, GPU engines that are responsible for video processing and rendering are loaded in turns. In this case, they are used in a serial manner. When the application code runs on the CPU, this can cause an ineffective scheduling on the GPU. The behavior can mislead you to interpret the application to be GPU bound.

Identify the GPU execution phase based on the computing task reference and GPU Utilization metrics. Then, you can define the overhead for creating the task and placing it into a queue.



**Fig. 20:** GPU Offload Platform window

To investigate a computing task, switch to the Graphics window to examine the type of work (rendering or computation) running on the GPU per thread. Select the Computing Task grouping and use the table to study the performance characterization of your task. To further analyze your computing task, run the GPU Compute/Media Hotspots analysis type.

Grouping: Computing Task										
Computing Task	Computing Task					EU Array		EU Threads Occupancy	Computing Threads Started	
	Total Time	Average T...	Insta...	SIMD Width	SVM Usage ...	Active	Stalled ▼	Idle		
Matrix1<float>	297.870ms	297.870ms	1	8		63.2%	36.7%	0.0%	95.1%	131,072
[Outside any task]	0ms					2.5%	7.2%	90.3%	6.4%	0
clEnqueueReadBufferR	0.007ms	0.007ms	1			0.0%	0.0%	100.0%	0.0%	0

**Fig. 21:** Computing Task grouping

## Run GPU Compute/Media Hotspots Analysis

To run the analysis:

1. In the Accelerators group, select the GPU Compute/Media Hotspots analysis type.

2. Configure analysis options as described in the previous section.
3. Click the **Start** button to run the analysis.

The screenshot shows the Intel VTUNE Profiler interface for GPU Compute/Media Hotspots analysis. At the top right, it says "INTEL VTUNE PROFILER". Below that, a "HOW" section title is followed by a large icon of a location pin and the text "GPU Compute/Media Hotspots (preview)". To the right of the icon is a dropdown arrow and a refresh/circular arrow icon. The main content area has a heading "Analyze the most time-consuming GPU kernels, characterize GPU utilization based on GPU hardware metrics, identify performance issues caused by memory latency or inefficient kernel algorithms, and analyze GPU instruction frequency per certain instruction types. [Learn more](#)".  
The "Characterization" tab is selected, indicated by a solid black dot next to the radio button. It contains:

- A dropdown menu set to "Overview".
- A "GPU sampling interval, ms" input field containing the value "1".
- Two checkboxes:
  - An empty checkbox labeled "Analyze memory bandwidth".
  - A checked checkbox labeled "Trace GPU Programming APIs".

  
The "Source Analysis" tab is unselected, indicated by an empty circle next to the radio button.  
A table titled "Computing task of interest" is present:

Computing task of interest	Instance step
Enter computing task of interest	Default

  
Below the table is a "Details" button with a right-pointing arrow.  
At the bottom of the interface are four large blue circular icons: a play button, a microphone, a file folder, and a right-pointing arrow.

**Fig. 22:** GPU Compute/Media Hotspots analysis

## Run Analysis from the Command line

On Linux OS:

```
vtune -collect gpu-hotspots -- ./matrix.dpcpp
```

On Windows OS:

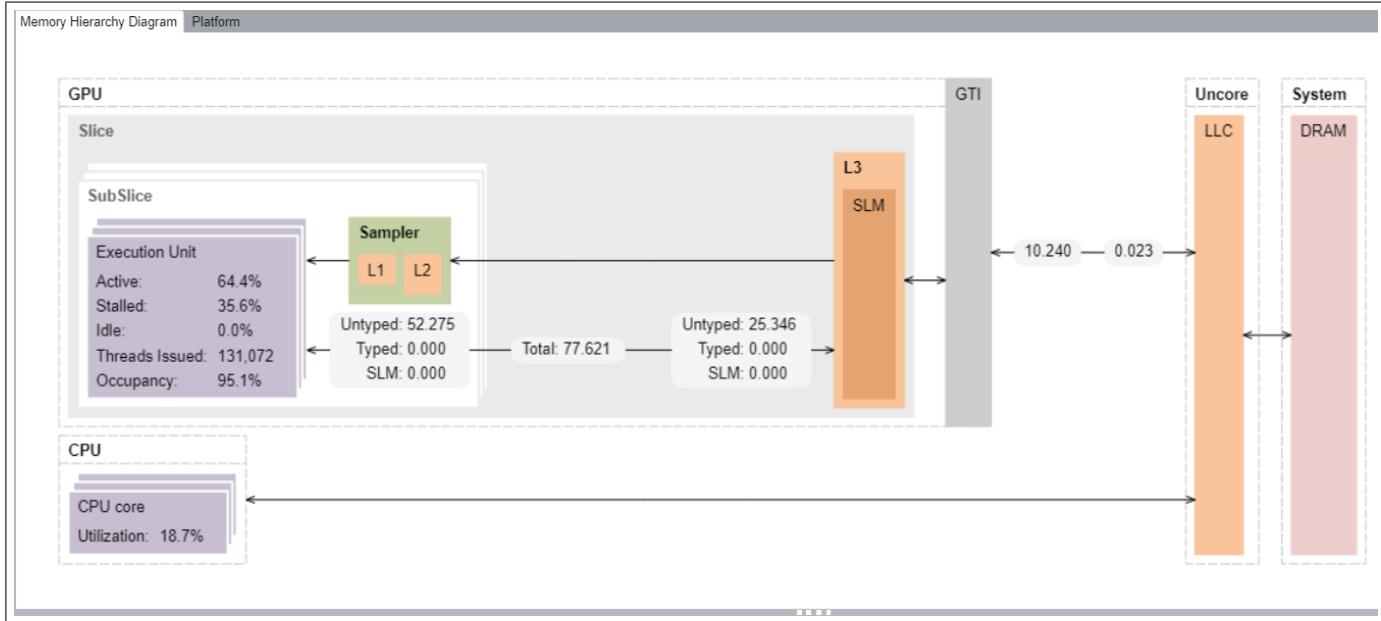
```
vtune.exe -collect gpu-hotspots -- matrix_multiply.exe
```

## Analyze Your Compute Tasks

Computing Task	L3 Bandwidth, GB/sec	Untyped Memory Ba...		Shared Local Me...		Typed Memory Ba...		GPU Memory Bandwidth, GB...	
		Read	Write	Read	Write	Read	Write	Read	Write
Matrix1<float>	77.621	52.275	25.346	0.000	0.000	0.000	0.000	10.240	0.023
clEnqueueReadBufferRect	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
[Outside any task]	0.000	0.015	0.000	0.000	0.000	0.000	0.014	0.033	0.024

**Fig. 23:** Characterization profile

The default analysis configuration invokes the Characterization profile with the Overview metric set. In addition to individual compute task characterization that is available through the GPU Offload analysis, VTune Profiler provides memory bandwidth metrics that are categorized by different levels of GPU memory hierarchy.



**Fig. 24:** VTune Profiler memory bandwidth metrics

You can analyze compute tasks at source code level too. For example, to count GPU clock cycles spent on a particular task or due to memory latency, use the Source Analysis option.

Analyze the most time-consuming GPU kernels, characterize GPU utilization based on GPU hardware metrics, identify performance issues caused by memory latency or inefficient kernel algorithms, and analyze GPU instruction frequency per certain instruction types. [Learn more](#)

- Characterization [?](#)
- Source Analysis [?](#)

Memory Latency

Basic Blocks Latency

Memory Latency

Enter computing task of interest Default

Overhead

**Fig. 25:** GPU Compute/Media Hotspots analysis, Source Analysis

In our matrix example

Once you have ported your code to DPC++

**Listing 122:** /examples/matrix/multiply.cpp

```

1 // Basic matrix multiply
2 void multiplyl(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
3                 TYPE c[][NUM], TYPE t[][NUM]) {
4     int i, j, k;
5
6     // Declare a deviceQueue
7     sycl::default_selector device;
8     sycl::queue q(device, exception_handler);
9     cout << "Running on " << q.get_device().get_info<sycl::info::device::name>()
10      << "\n";
11     // Declare a 2 dimensional range
12     sycl::range<2> matrix_range{NUM, NUM};
13
14     // Declare 3 buffers and Initialize them
15     sycl::buffer<TYPE, 2> bufferA((TYPE *)a, matrix_range);
16     sycl::buffer<TYPE, 2> bufferB((TYPE *)b, matrix_range);
17     sycl::buffer<TYPE, 2> bufferC((TYPE *)c, matrix_range);

```

(continues on next page)

(continued from previous page)

```

18 // Submit our job to the queue
19 q.submit([&](auto &h) {
20     // Declare 3 accessors to our buffers. The first 2 read and the last
21     // read_write
22     sycl::accessor accessorA(bufferA, h, sycl::read_only);
23     sycl::accessor accessorB(bufferB, h, sycl::read_only);
24     sycl::accessor accessorC(bufferC, h);
25
26     // Execute matrix multiply in parallel over our matrix_range
27     // ind is an index into this range
28     h.parallel_for(matrix_range, [=](sycl::id<2> ind) {
29         int k;
30         for (k = 0; k < NUM; k++) {
31             // Perform computation ind[0] is row, ind[1] is col
32             accessorC[ind[0]][ind[1]] +=
33                 accessorA[ind[0]][k] * accessorB[k][ind[1]];
34         }
35     });
36     }).wait_and_throw();
37 } // multiply1

```

If we analyze our GPU-offload report from the command-line we can get some detailed recommendation on how to optimize the application.

```

Elapsed Time: 2.805s
GPU Utilization: 3.3%
GPU utilization is low. Consider offloading more work to the GPU to
increase overall application performance.

GPU Utilization
GPU Engine      Packet Type   GPU Time    GPU Utilization(%)
-----
Render and GPGPU Unknown       0.091s        3.3%

Hottest GPU Computing Tasks
Computing Task  Total Time  Execution Time  % of Execution(%)  Instance Count
-----
Matrix1<float>    0.183s       0.086s        47.0%                1

Recommendations:
GPU Utilization: 3.3%
GPU utilization is low. Switch to the for in-depth analysis of host
activity. Poor GPU utilization can prevent the application from
offloading effectively.

Transfer Time: 0.097s
Execution time on the device is less than memory transfer time. Make sure
your offload schema is optimal. Use Intel Advisor tool to get an insight
into possible causes for inefficient offload.

```

We can also examine how efficient our GPU kernel is running using GPU-hotspots. How often our execution units are stalled can be a good indication of GPU performance. Another important metric is whether we are L3

Bandwidth bound, in our case VTune is indicating that our L3 bandwidth was high while we stalled.

```
Elapsed Time: 1.849s
GPU Time: 0.090s

EU Array Stalled/Idle: 6.2%
GPU L3 Bandwidth Bound: 65.2%

L3 bandwidth was high when EUs were stalled or idle.
Consider improving cache reuse.

FPU Utilization: 76.4%
```

For more ways to optimize GPU performance using VTune Profiler, see [Software Optimization for Intel® GPUs](#) in the Intel® VTune™ Profiler Performance Analysis Cookbook and [Optimize Applications for Intel® GPUs with Intel® VTune™ Profiler](#).

## 11.2 Intel® Advisor GPU Analysis

Intel Advisor has two features that can help you analyze your performance of your application running on a GPU:

- Use Offload Modeling to identify kernels in your CPU-based code and predict their performance when run on a GPU. It also allows you to explore different GPU configurations for GPUs that do not exist yet.
- Use GPU Roofline Insights to see how your application is performing when compared to the limitations of your GPU.

Identify Regions to Offload to GPU with Offload Modeling

The Offload Modeling feature, a part of Intel Advisor, can be used to:

- Identify the portions of a code that are profitable to be offloaded to a GPU.
- Predict the code's performance if run on a GPU.
- Experiment with accelerator configuration parameters.

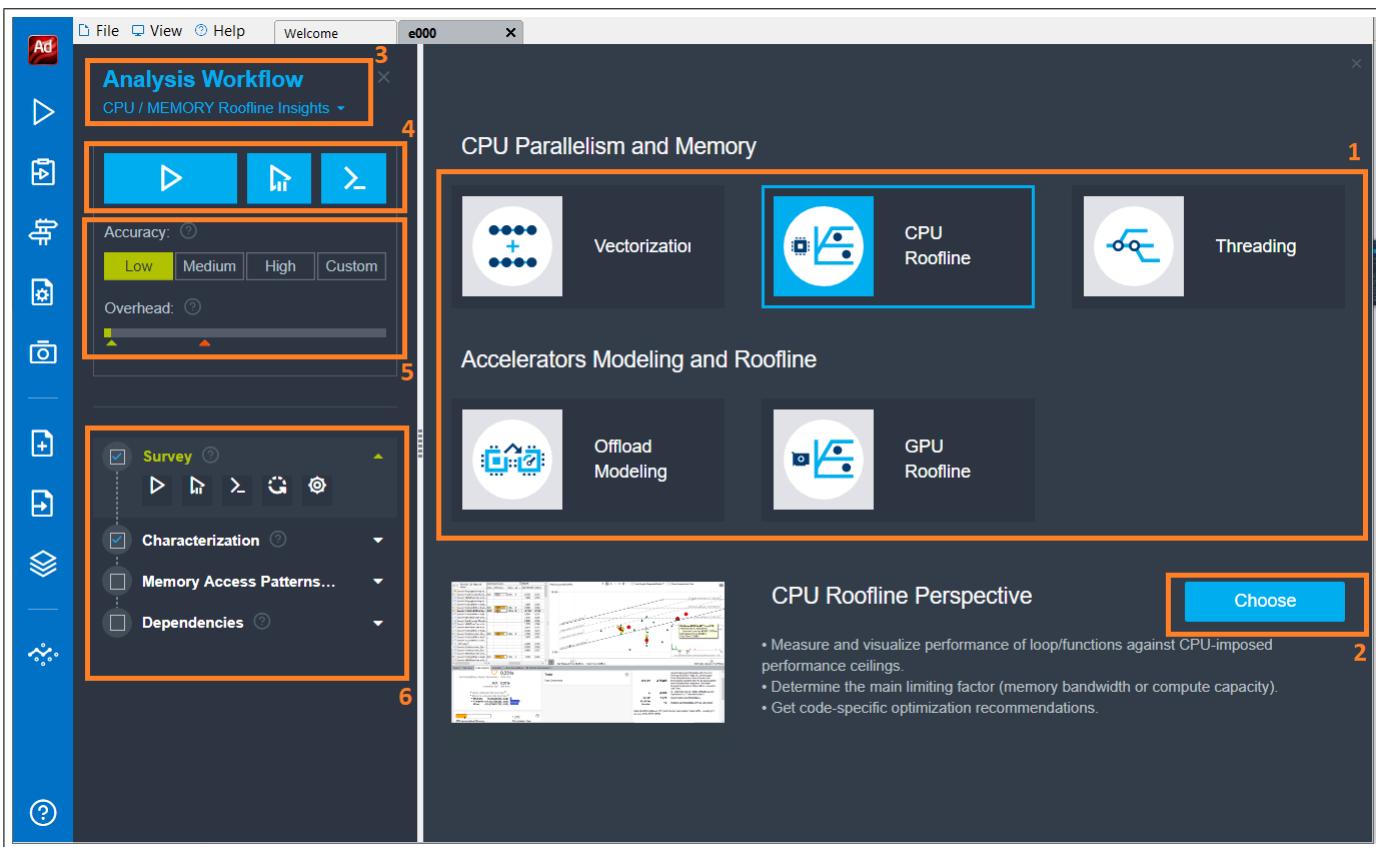
Offload Modeling produces upper-bound speedup estimates using a bounds-and-bottlenecks performance model. It takes measured x86 CPU metrics and application characteristics as an input and applies an analytical model to estimate execution time and characteristics on a target GPU.

You can run the Offload Modeling perspective from the Intel Advisor GUI, using the `advisor` command line interface, or using the dedicated Python\* scripts delivered with the Intel Advisor. This topic describes how to run Offload modeling with the scripts. For detailed description of other ways to run the perspective, see the [Intel Advisor User Guide](#).

Prerequisites: Set up Intel Advisor environment variables:

- On Linux\* OS: `source <install-dir>/advisor-vars.sh`
- On Windows\* OS: `<install-dir>/advisor-vars.bat`

To run Offload Modeling for a C++ Matrix Multiply application on Linux\* OS:



**Fig. 26:** Offload Modelling for a C++ Matrix application on Linux\*OS

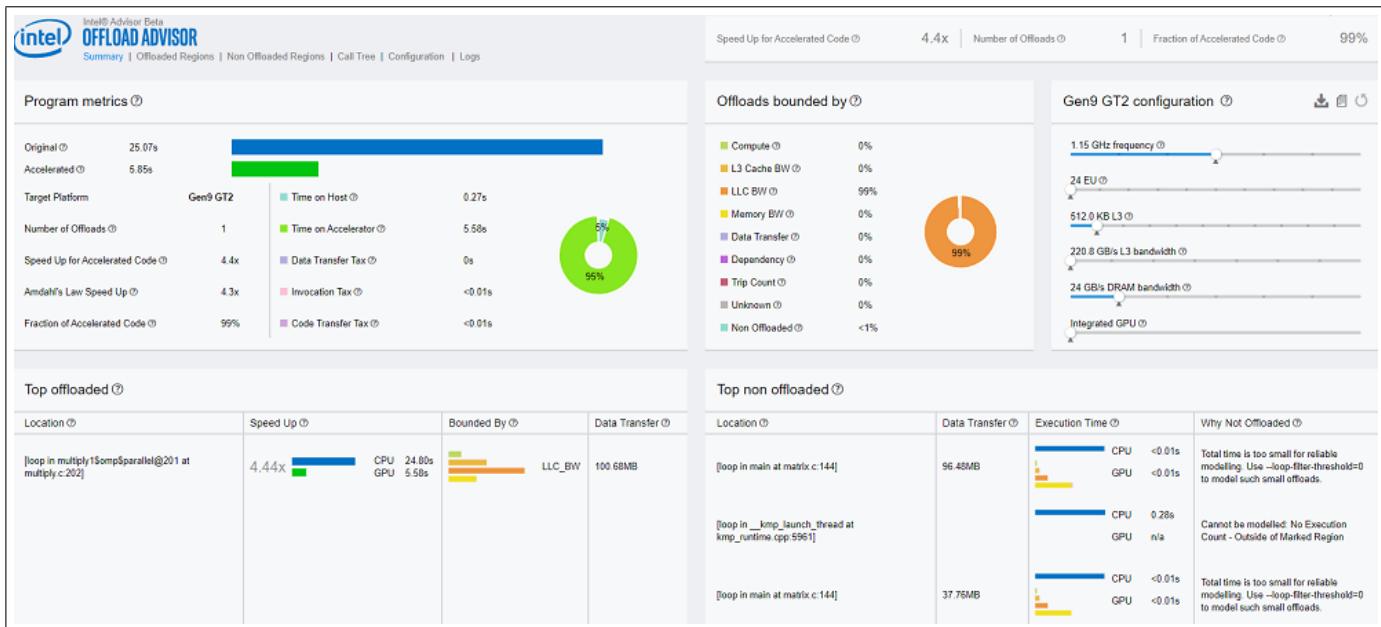
1. Collect application performance metrics with `collect.py`:

```
advisor-python $APM/collect.py ./advisor_project --config gen9_gt2 -- matrix_multiply
```

2. Model your application performance on a GPU with `analyze.py`:

```
advisor-python $APM/analyze.py ./advisor_project --config gen9_gt2
```

Once you have run the performance modeling, you can open the results in the Intel Advisor GUI or see CSV metric reports and an interactive HTML report generated in the `advisor_project/e000/pp000/data.0`



**Fig. 27:** Intel Advisor GUI, Offload Advisor

For example, in the Summary section of the report, review the following:

- The original execution time on a CPU, the predicted execution time on a GPU accelerator, the number of offloaded regions, and the estimated speedup in the Program metrics pane. For Matrix Multiply, Intel Advisor reports 4.4x potential speedup.
- What the offloads are bounded by. This pane reports the main limiting factors that prevent your application from achieving a better performance on a target device. The Matrix Multiply application is 99% bounded by last level cache (LLC) cache bandwidth.
- Exact source lines of the **Top Offloaded** code regions that can benefit from offloading to the GPU and estimated performance of each code region. For Matrix Multiply, there is one code region recommended for offloading.
- Exact source lines of the **Top Non-Offloaded** code regions that are not recommended for offloading and specific reasons for it.

Go to the Offloaded Regions tab to view the detailed measured and estimated metrics for the code regions recommended for offloading. It also reports estimated amount of data transferred for the code regions and the corresponding offload taxes.

Use the data in the report to decide what regions of your code to port your code to DPC++. For example, you can port the C++ Matrix Multiply application to DPC++ as follows:

**Listing 123:** /examples/matrix/multiply.cpp

```

1 // Basic matrix multiply
2 void multiply1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
3                 TYPE c[][NUM], TYPE t[][NUM]) {
4     int i, j, k;

```

(continues on next page)

(continued from previous page)

```

5   // Declare a deviceQueue
6   sycl::default_selector device;
7   sycl::queue q(device, exception_handler);
8   cout << "Running on " << q.get_device().get_info<sycl::info::device::name>()
9   << "\n";
10  // Declare a 2 dimensional range
11  sycl::range<2> matrix_range{NUM, NUM};
12
13
14  // Declare 3 buffers and Initialize them
15  sycl::buffer<TYPE, 2> bufferA((TYPE *)a, matrix_range);
16  sycl::buffer<TYPE, 2> bufferB((TYPE *)b, matrix_range);
17  sycl::buffer<TYPE, 2> bufferC((TYPE *)c, matrix_range);
18  // Submit our job to the queue
19  q.submit([&](auto &h) {
20      // Declare 3 accessors to our buffers. The first 2 read and the last
21      // read_write
22      sycl::accessor accessorA(bufferA, h, sycl::read_only);
23      sycl::accessor accessorB(bufferB, h, sycl::read_only);
24      sycl::accessor accessorC(bufferC, h);
25
26      // Execute matrix multiply in parallel over our matrix_range
27      // ind is an index into this range
28      h.parallel_for(matrix_range, [=](sycl::id<2> ind) {
29          int k;
30          for (k = 0; k < NUM; k++) {
31              // Perform computation ind[0] is row, ind[1] is col
32              accessorC[ind[0]][ind[1]] +=
33                  accessorA[ind[0]][k] * accessorB[k][ind[1]];
34          }
35      });
36  }).wait_and_throw();
37 } // multiplyl

```

## Run a GPU Roofline Analysis

To estimate performance of an application running on a GPU against hardware limitations, you can use the GPU Roofline Insights feature. Intel Advisor can generate a Roofline model for kernels running on Intel® GPUs. The GPU Roofline model offers a very efficient way to characterize your kernels and visualize how far you are from ideal performance. For details about the GPU Roofline, see the [Intel Advisor User Guide](#).

**Prerequisites:** It is recommended to run the GPU Roofline with root privileges on Linux\* OS or as an Administrator on Windows\* OS.

If you do not have root permissions on Linux, configure your system to enable collecting GPU metrics for non-root users:

1. Add your username to the video group. To check if you are already in the video group:

```
groups | grep video
```

If you are not part of the video group, add your username to it:

```
sudo usermod -a -G video <username>
```

Set the value of the dev.i915.perf\_stream\_paranoid sysctl option to 0:

```
sysctl -w dev.i915.perf_stream_paranoid=0
```

2. Disable time limit in order to run OpenCL kernel for longer period:

```
sudo sh -c "echo N> /sys/module/i915/parameters/enable_hangcheck"
```

For all users:

1. Make sure that your DPC++ code runs correctly on the GPU. To check which hardware you are running on, add the following to your DPC++ code and run it:

```
sycl::default_selector selector;
sycl::queue queue(selector);
auto d = queue.get_device();
std::cout<<Running on :<<d.get_info<cl::sycl::info::device::name>()<<std::endl;
```

2. Set up the Intel Advisor environment for Linux OS:

```
source <advisor_install_dir>/env/vars.sh
```

and for Windows OS:

```
<install-dir>/advisor-vars.bat
```

To run the GPU Roofline analysis in the Intel Advisor CLI:

1. Run the Survey analysis with the profile-gpu option:

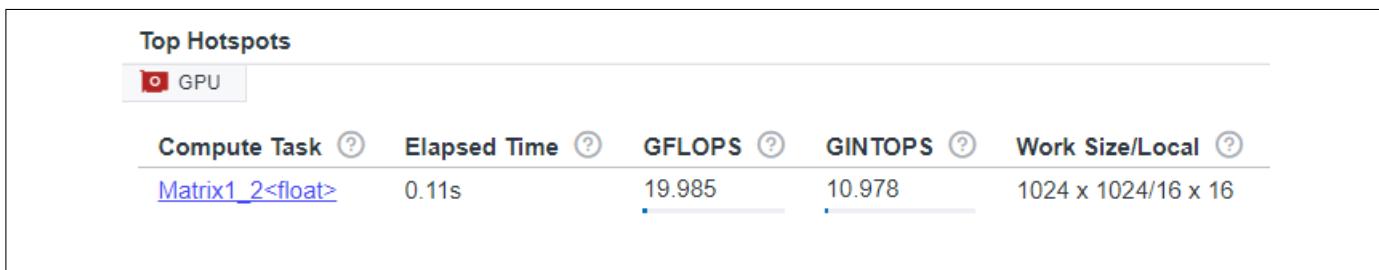
```
advisor --collect=survey --profile-gpu --project-dir=./advisor-project --search-dir src:r=./matrix_multiply -- matrix_multiply
```

2. Run the Trip Count and FLOP analysis with --profile-gpu::

```
advisor --collect=tripcounts --stacks --flop --profile-gpu --project-dir=./advisor-project --search-dir src:r./matrix_multiply -- matrix_multiply
```

3. Open the generated GPU Roofline report in the Intel Advisor GUI. Review the following metrics for the DPC++ Matrix Multiply application:

- In the Summary tab, view top hotspots and the memory layout in the Top Hotspots pane.



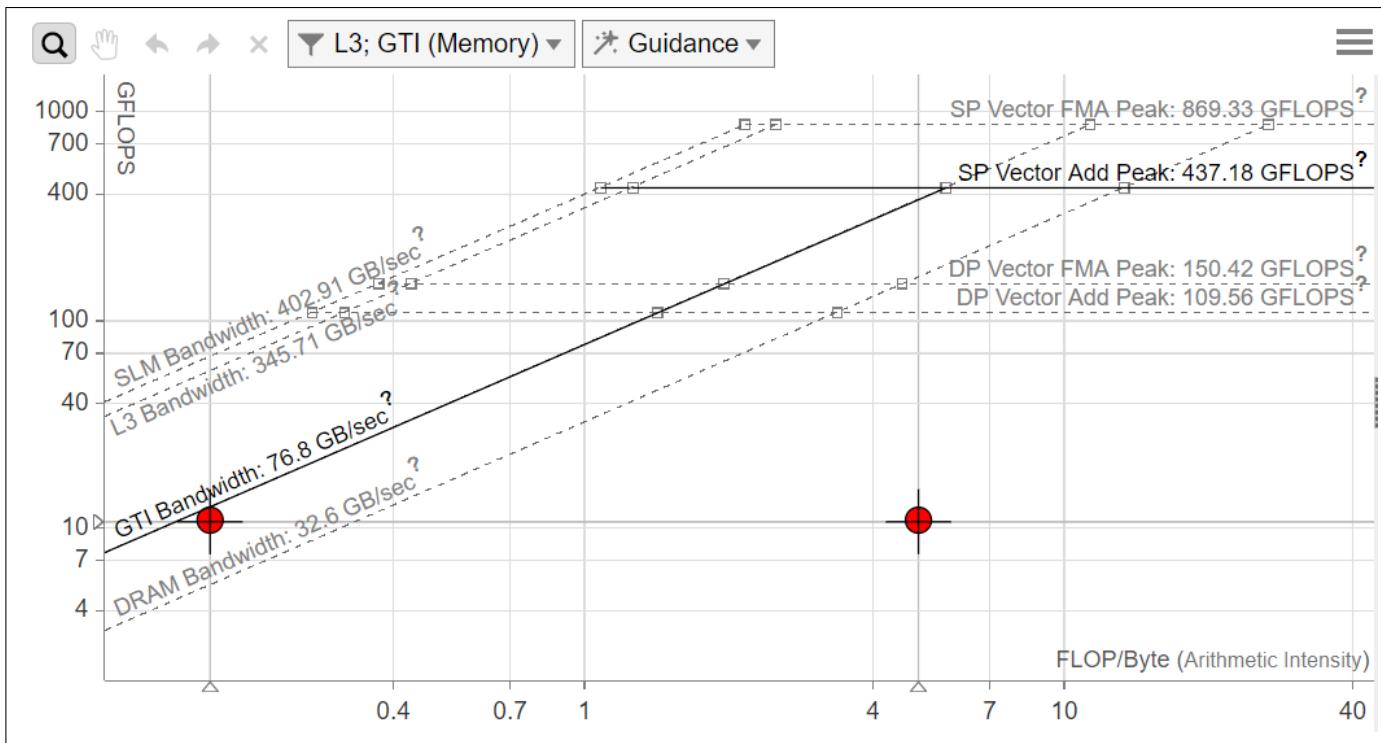
**Fig. 28:** Top Hotspots pane

See how efficiently your application uses execution units in the Performance Characteristics pane.



**Fig. 29:** Performance Characteristics pane

In the GPU Roofline Regions tab, see the GPU Roofline chart and performance metrics.



**Fig. 30:** GPU Roofline chart and performance metrics

- The Matrix Multiply application gets 10.98 GFLOPS. It uses global memory and is not optimized for a local (SLM) memory. Since the application uses a global accessor, this makes sense.
- The application is far from the maximum bandwidth of the GTI as represented by the red dot on the right.
- The dot on the left represents the L3 bandwidth. As the chart shows, it is far from the L3 bandwidth maximum.

As the GPU Roofline chart suggests, there are several possible optimizations that we can do to get better usage of memory bandwidth:

- Use local memory (SLM).
- Use cache blocking technique to better use SLM/L3 cache.

The following code is the optimized version of the Matrix Multiply application. In this version, we declare two tiles and define them as `sycl::access::target::local`, we also modify the kernel to process these tiles in some inner loops.

**Listing 124:** /examples/matrix/multiply.cpp

```
1 // Replaces accessorC reference with a local variable
2 void multiplyl_1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
3                  TYPE c[][NUM], TYPE t[][NUM]) {
4
5     int i, j, k;
6
7     // Declare a deviceQueue
8     sycl::default_selector device;
9     sycl::queue q(device, exception_handler);
10    cout << "Running on " << q.get_device().get_info<sycl::info::device::name>()
11      << "\n";
12
13    // Declare a 2 dimensional range
14    sycl::range<2> matrix_range{NUM, NUM};
15
16    // Declare 3 buffers and Initialize them
17    sycl::buffer<TYPE, 2> bufferA((TYPE *)a, matrix_range);
18    sycl::buffer<TYPE, 2> bufferB((TYPE *)b, matrix_range);
19    sycl::buffer<TYPE, 2> bufferC((TYPE *)c, matrix_range);
20
21    // Submit our job to the queue
22    q.submit([&](auto &h) {
23        // Declare 3 accessors to our buffers. The first 2 read and the last
24        // read_write
25        sycl::accessor accessorA(bufferA, h, sycl::read_only);
26        sycl::accessor accessorB(bufferB, h, sycl::read_only);
27        sycl::accessor accessorC(bufferC, h);
28
29        // Execute matrix multiply in parallel over our matrix_range
30        // ind is an index into this range
31        h.parallel_for(matrix_range, [=](sycl::id<2> ind) {
32            int k;
33            TYPE acc = 0.0;
34            for (k = 0; k < NUM; k++) {
35                // Perform computation ind[0] is row, ind[1] is col
36                acc += accessorA[ind[0]][k] * accessorB[k][ind[1]];
37            }
38            accessorC[ind[0]][ind[1]] = acc;
39        });
40    }).wait_and_throw();
41 }
42
43 // Replaces accessorC reference with a local variable and adds matrix tiling
44 void multiplyl_2(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
45                  TYPE c[][NUM], TYPE t[][NUM]) {
46     int i, j, k;
```

(continues on next page)

(continued from previous page)

```

48 // Declare a deviceQueue
49 sycl::default_selector device;
50 sycl::queue q(device, exception_handler);
51 cout << "Running on " << q.get_device().get_info<sycl::info::device::name>()
52     << "\n";
53
54 // Declare a 2 dimensional range
55 sycl::range<2> matrix_range{NUM, NUM};
56 sycl::range<2> tile_range{MATRIXTILESIZE, MATRIXTILESIZE};
57
58 // Declare 3 buffers and Initialize them
59 sycl::buffer<TYPE, 2> bufferA((TYPE *)a, matrix_range);
60 sycl::buffer<TYPE, 2> bufferB((TYPE *)b, matrix_range);
61 sycl::buffer<TYPE, 2> bufferC((TYPE *)c, matrix_range);
62
63 // Submit our job to the queue
64 q.submit([&](auto &h) {
65     // Declare 3 accessors to our buffers. The first 2 read and the last
66     // read_write
67     sycl::accessor accessorA(bufferA, h, sycl::read_only);
68     sycl::accessor accessorB(bufferB, h, sycl::read_only);
69     sycl::accessor accessorC(bufferC, h);
70
71     // Create matrix tiles
72     sycl::accessor<TYPE, 2, sycl::access::mode::read_write,
73         sycl::access::target::local>
74     aTile(sycl::range<2>(MATRIXTILESIZE, MATRIXTILESIZE), h);
75     sycl::accessor<TYPE, 2, sycl::access::mode::read_write,
76         sycl::access::target::local>
77     bTile(sycl::range<2>(MATRIXTILESIZE, MATRIXTILESIZE), h);
78     // Execute matrix multiply in parallel over our matrix_range
79     // ind is an index into this range
80     h.parallel_for(sycl::nd_range<2>(matrix_range, tile_range),
81         [=](cl::sycl::nd_item<2> it) {
82             int k;
83             const int numTiles = NUM / MATRIXTILESIZE;
84             const int row = it.get_local_id(0);
85             const int col = it.get_local_id(1);
86             const int globalRow =
87                 MATRIXTILESIZE * it.get_group(0) + row;
88             const int globalCol =
89                 MATRIXTILESIZE * it.get_group(1) + col;
90             TYPE acc = 0.0;
91             for (int t = 0; t < numTiles; t++) {
92                 const int tiledRow = MATRIXTILESIZE * t + row;
93                 const int tiledCol = MATRIXTILESIZE * t + col;
94                 aTile[row][col] = accessorA[globalRow][tiledCol];
95                 bTile[row][col] = accessorB[tiledRow][globalCol];
96                 it.barrier(sycl::access::fence_space::local_space);
97                 for (k = 0; k < MATRIXTILESIZE; k++) {
98                     // Perform computation ind[0] is row, ind[1] is col

```

(continues on next page)

(continued from previous page)

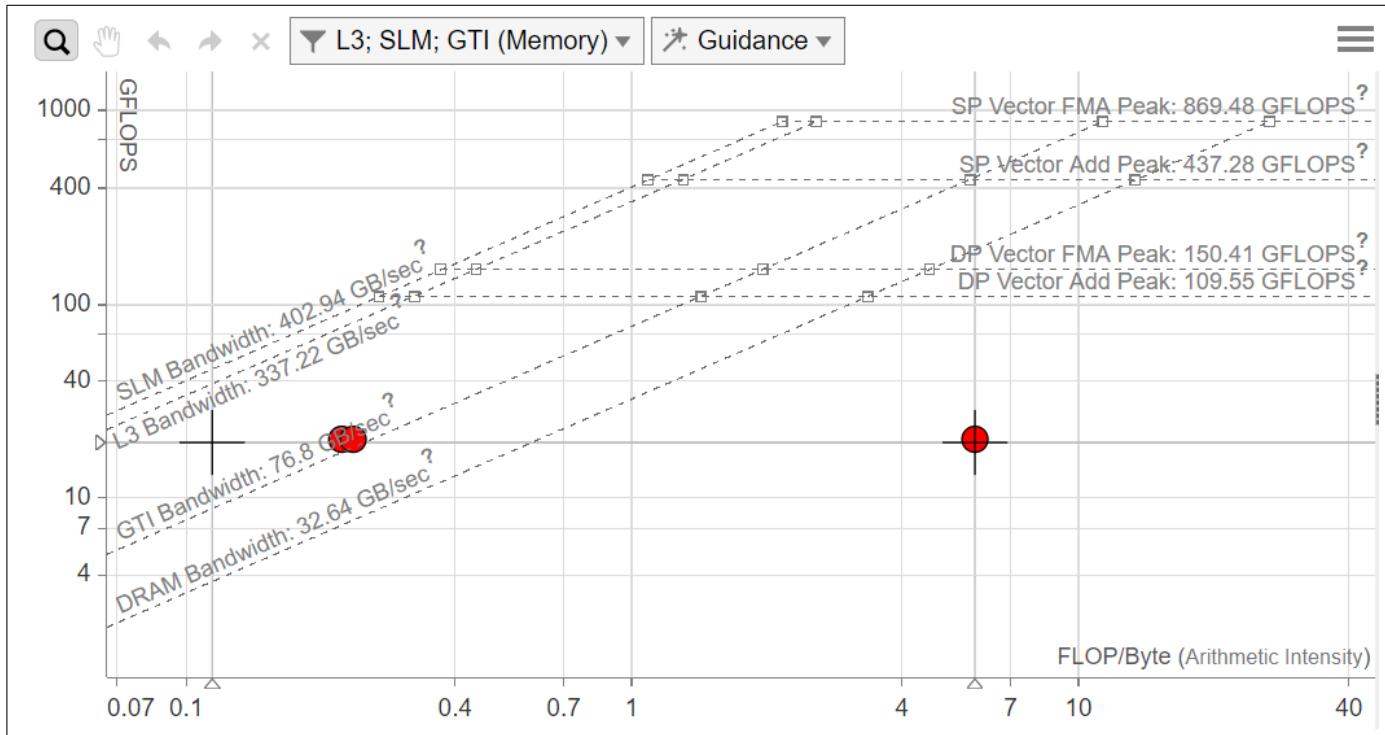
```

99         acc += aTile[row][k] * bTile[k][col];
100    }
101    it.barrier(sycl::access::fence_space::local_space);
102    }
103    accessorC[globalRow][globalCol] = acc;
104  });
105 }).wait_and_throw();
} // multiply_1_2

```

Save the optimized version as `multiply_1_2` and rerun the GPU Roofline. As the new chart shows:

- The optimized application gets 19.985 GFLOPS.
- The application uses global and SLM memory, which represents the 16x16 tile. This increases the memory bandwidth.



**Fig. 31:** GPU Roofline new chart

## 11.3 Doing IO in the kernel

Print statement is the most fundamental capability that is needed for looking the results of the program. In accelerators, printing is surprisingly hard and also fairly expensive in terms of overhead. DPC++ provides some capabilities to help make this task similar to standard I/O C/C++ programs but there are some quirks that one needs understand because of the way accelerators work. File I/O is not possible from DPC++ kernels.

SYCL provides **stream** class to allow users to print information to the console from within kernels, providing an easy way to debug simple issues without resorting to a debugger. The **stream** class provides functionality that

is very similar to the C++ STL **ostream** class, and its usage is similar to the STL class. Below we describe how to use SYCL **stream** class to output information from within an enqueued kernel.

To use the class we must first instantiate it. The signature of the **stream** constructor is as follows:

```
stream(size_t BufferSize, size_t MaxStatementSize, handler &CGH);
```

The constructor takes three parameters:

- BufferSize: the total number of characters that may be printed over the entire kernel range
- MaxStatementSize: the maximum number of characters in any one call to the stream class
- CGH: reference to the **sycl::handler** parameter in the **sycl::queue::submit** call

Usage is very similar to that of the C++ STL **ostream std::cout** class. The message or data that needs to be printed is sent to the SYCL **stream** instance via the appropriate **operator<<** method. SYCL provides implementations for all the builtin data types (such as **int**, **char** and **float**) as well as some common classes (such as **sycl::nd\_range** and **sycl::group**).

Example usage of a SYCL **stream** instance:

**Listing 125:** /examples/io-kernel/out.cpp

```
1 void out1() {
2     constexpr int N = 16;
3     sycl::queue q;
4     q.submit([&](auto &cgh) {
5         sycl::stream str(8192, 1024, cgh);
6         cgh.parallel_for(N, [=](sycl::item<1> it) {
7             int id = it[0];
8             /* Send the identifier to a stream to be printed on the console */
9             str << "ID=" << id << sycl::endl;
10        });
11    }).wait();
12 } // end out1
```

The use of **sycl::endl** is analogous to the use of the C++ STL **std::endl** **ostream** reference—it serves to insert a new line as well as flush the stream.

Compiling and executing the above kernel gives the following output:

```
ID=0
ID=1
ID=2
ID=3
ID=4
ID=5
ID=6
ID=7
ID=8
ID=9
ID=10
```

(continues on next page)

(continued from previous page)

```
ID=11  
ID=12  
ID=13  
ID=14  
ID=15
```

Care must be taken in choosing the appropriate **BufferSize** and **MaxStatementSize** parameters. Sizes that are insufficient may cause statements to either not be printed, or to be printed with less information than expected. Consider the following kernel

**Listing 126:** /examples/io-kernel/out.cpp

```
1 void out2() {  
2     sycl::queue q;  
3     q.submit([&](auto &cgh) {  
4         sycl::stream str(8192, 4, cgh);  
5         cgh.parallel_for(1, [=](sycl::item<1> it) {  
6             str << "ABC" << sycl::endl; // Print statement 1  
7             str << "ABCDEFG" << sycl::endl; // Print statement 2  
8         });  
9     }).wait();  
10 } // end out2
```

Compiling and running the above kernel gives the following output:

```
ABC
```

The first statement was successfully printed out since the number of characters to be printed is 4 (including the newline introduced by `sycl::endl`) and the maximum statement size (as specified by the **MaxStatementSize** parameter to the `sycl::stream` constructor) is also 4. However, only the newline from the second statement is printed.

The following kernel shows the impact of increasing the allowed maximum character size:

**Listing 127:** /examples/io-kernel/out.cpp

```
1 void out3() {  
2     sycl::queue q;  
3     q.submit([&](auto &cgh) {  
4         sycl::stream str(8192, 10, cgh);  
5         cgh.parallel_for(1, [=](sycl::item<1> it) {  
6             str << "ABC" << sycl::endl; // Print statement 1  
7             str << "ABCDEFG" << sycl::endl; // Print statement 2  
8         });  
9     }).wait();  
10 } // end out3
```

Compiling and running the above kernel gives the expected output:

```
ABC  
ABCDEFG
```

The examples above used simple kernels with a single work item. More realistic kernels will typically include multiple work items. In these cases, no guarantee is made as to the specific order of the statements printed to the console and users should expect statements from different work items to be interleaved. Consider the following kernel.

**Listing 128:** /examples/io-kernel/out.cpp

```

1 void out4() {
2     sycl::queue q;
3     q.submit([&](auto &cgh) {
4         sycl::stream str(8192, 1024, cgh);
5         cgh.parallel_for(sycl::nd_range<1>(32, 4), [=](sycl::nd_item<1> it) {
6             int id = it.get_global_id();
7             str << "ID=" << id << sycl::endl;
8         });
9     }).wait();
10 } // end out4

```

One run can produce the following output.

```

ID=0
ID=1
ID=2
ID=3
ID=4
ID=5
[snip]
ID=26
ID=27
ID=28
ID=29
ID=30
ID=31

```

When this program is run again we might get the output in a totally different order which depends on the order in which threads are executed.

```

ID=4
ID=5
ID=6
ID=7
ID=0
ID=1
[snip]
ID=14
ID=15
ID=28
ID=29
ID=30
ID=31

```

The output from `sycl::stream` is printed after the kernel has completed execution. In most cases this is of no consequence. However, should the kernel fault or throw an exception then no statement will be printed. To

illustrate this, consider the following kernel which raises an exception:

**Listing 129:** /examples/io-kernel/out.cpp

```

1 void out4() {
2     sycl::queue q;
3     q.submit([&](auto &cgh) {
4         sycl::stream str(8192, 1024, cgh);
5         cgh.parallel_for(sycl::nd_range<1>(32, 4), [=](sycl::nd_item<1> it) {
6             int id = it.get_global_id();
7             str << "ID=" << id << sycl::endl;
8         });
9     }).wait();
10 } // end out4

```

Compiling and executing the above code generates a segmentation fault due the write to a null pointer.

Segmentation fault (core dumped)

None of the print statements are actually printed to the console; instead, the user sees an error message about a segmentation fault. This is unlike traditional C/C++ streams.

## 11.4 Using the timers

The standard C++ chrono library can be used for tracking times with varying degrees of precision in DPC++. The following example shows how to use chrono timer class to time a kernel execution from host side.

```

#include <iostream>
using namespace sycl;

// Array type and data size for this example.
constexpr size_t array_size = (1 << 16);
typedef std::array<int, array_size> IntArray;

double VectorAdd(queue &q, const IntArray &a, const IntArray &b, IntArray &sum) {
    range<1> num_items{a.size()};

    buffer a_buf(a);
    buffer b_buf(b);
    buffer sum_buf(sum.data(),num_items);

    event e = q.submit([&](handler &h) {
        // Input accessors
        auto a_acc = a_buf.get_access<access::mode::read>(h);
        auto b_acc = b_buf.get_access<access::mode::read>(h);
        // Output accessor
        auto sum_acc = sum_buf.get_access<access::mode::write>(h);

        h.parallel_for(num_items, [=](id<1> i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
    });
    q.wait();
}

```

(continues on next page)

(continued from previous page)

```

return(e.template get_profiling_info<info::event_profiling::command_end>() -
       e.template get_profiling_info<info::event_profiling::command_start>());
}

void InitializeArray(IntArray &a) {
    for (size_t i = 0; i < a.size(); i++) a[i] = i;
}

int main() {
    default_selector d_selector;

    IntArray a, b, sum;

    InitializeArray(a);
    InitializeArray(b);

    queue q(d_selector, property::queue::enable_profiling{});

    std::cout << "Running on device: "
          << q.get_device().get_info<info::device::name>() << "\n";
    std::cout << "Vector size: " << a.size() << "\n";

    double t = VectorAdd(q, a, b, sum);

    std::cout << "Vector add successfully completed on device - took " << t << " u-secs\n";
    return 0;
}

```

It must be noted that this timing is purely from the host side. The actual execution of the kernel on the device may start much later after the submission of the kernel by the host. DPC++ provides a profiling capability which allows one to keep track of the time it took to execute kernels.

```

#include <CL/sycl.hpp>
#include <array>
#include <iostream>
using namespace sycl;

// Array type and data size for this example.
constexpr size_t array_size = (1 << 16);
typedef std::array<int, array_size> IntArray;

double VectorAdd(queue &q, const IntArray &a, const IntArray &b, IntArray &sum) {
    range<1> num_items{a.size()};

    buffer a_buf(a);
    buffer b_buf(b);
    buffer sum_buf(sum.data(),num_items);

    event e = q.submit([&](handler &h) {
        // Input accessors
        auto a_acc = a_buf.get_access<access::mode::read>(h);

```

(continues on next page)

(continued from previous page)

```
auto b_acc = b_buf.get_access<access::mode::read>(h);
// Output accessor
auto sum_acc = sum_buf.get_access<access::mode::write>(h);

h.parallel_for(num_items, [=](id<1> i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
});

q.wait();
return(e.template get_profiling_info<info::event_profiling::command_end>() -
e.template get_profiling_info<info::event_profiling::command_start>());
}

void InitializeArray(IntArray &a) {
    for (size_t i = 0; i < a.size(); i++) a[i] = i;
}

int main() {
    default_selector d_selector;

    IntArray a, b, sum;

    InitializeArray(a);
    InitializeArray(b);

    queue q(d_selector, property::queue::enable_profiling{});

    std::cout << "Running on device: "
        << q.get_device().get_info<info::device::name>() << "\n";
    std::cout << "Vector size: " << a.size() << "\n";

    double t = VectorAdd(q, a, b, sum);

    std::cout << "Vector add successfully completed on device - took " << t << " nano-secs\n";
    return 0;
}
```

When these examples are run, it is quite possible that the time with the chrono timer is much larger than the time with the DPC++ profiling class. This is due to the fact that the DPC++ profiling does not include any data transfer times between the host and the offload device.

## 11.5 How to Use the Intercept Layer for OpenCL™ Applications

Linux\* and OS X\*: Linux OS X Build Status | Windows\*: Windows Build Status

The Intercept Layer for OpenCL Applications is a tool that can intercept and modify OpenCL calls for debugging and performance analysis. Using the Intercept Layer for OpenCL Applications requires no application or driver modifications.

To operate, the Intercept Layer for OpenCL Applications masquerades as the OpenCL ICD loader (usually) or as an OpenCL implementation (rarely) and is loaded when the application intends to load the real OpenCL ICD loader. As part of the Intercept Layer for OpenCL Application's initialization, it loads the real OpenCL ICD loader

and gets function pointers to the real OpenCL entry points. Then, whenever the application makes an OpenCL call, the call is intercepted and can be passed through to the real OpenCL with or without changes.

To access the OpenCL Intercept Layer git:

```
git clone https://github.com/intel/opencl-intercept-layer
```

All controls are documented here <https://github.com/intel/opencl-intercept-layer/blob/master/docs/controls.md>

**See [intercept documentation](#) for information about controls.**

To run use the following setup:

```
export CLI_OpenCLFileName=/opt/intel/inteloneapi/compiler/latest/linux/lib/libOpenCL.so.1
export LD_LIBRARY_PATH=/home/opencl-intercept-layer/build/intercept:$LD_LIBRARY_PATH

export SYCL_BE=PI_OPENCL

CLI_ReportToStderr=0 CLI_ReportToFile=1 CLI_HostPerformanceTiming=1 CLI_
↪DevicePerformanceTiming=1 CLI_DumpDir=. ./matrix.dpcpp
```

This will generate a file called `clintercept_report.txt`. The file will include the following data and tables shown below.

- Total Enqueues: 2
- Total Time (ns): 1604325652

**Table 10:** Host Performance Timing Results

Function Name	Calls	Time (ns)	Time (%)	Average (ns)	Min (ns)	Max (ns)
clBuildProgram	1	337069812	21.01%	337069812	337069812	337069812
clCreateBuffer	3	3393909	0.21%	1131303	140325	2036170
clCreateCommandQueueWithProperties	1	5221	0.00%	5221	5221	5221
clCreateContext	1	33639	0.00%	33639	33639	33639
clCreateKernel	1	11713	0.00%	11713	11713	11713
clCreateProgramWithIL	1	153337	0.01%	153337	153337	153337
clEnqueueNDRangeKernel(_ZTS9Matrix1_2IfE)	3	3102488	0.19%	3102488	3102488	3102488
clEnqueueReadBufferRect	1	1099684	0.07%	1099684	1099684	1099684
clGetContextInfo	8	4720	0.00%	590	160	1997
clGetDeviceIDs	12	53004	0.00%	4417	504	14853
clGetDeviceInfo	30	85695	0.01%	2856	133	19920
clGetExtensionFunctionAddressForPlatform	3	6446	0.00%	2148	1317	3687
clGetKernelInfo	2	716	0.00%	358	169	547
clGetPlatformIDs	2	1198290216	74.69%	599145108	715	1198289501
clGetPlatformInfo	12	22538	0.00%	1878	404	7326
clReleaseCommandQueue	1	1744	0.00%	1744	1744	1744
clReleaseContext	1	331	0.00%	331	331	331
clReleaseDevice	6	6365	0.00%	1060	491	1352
clReleaseEvent	2	2398	0.00%	1199	992	1406
clReleaseKernel	1	2733	0.00%	2733	2733	2733
clReleaseMemObject	3	45464	0.00%	15154	10828	22428
clReleaseProgram	1	51380	0.00%	51380	51380	51380
clRetainDevice	6	8680	0.00%	1446	832	2131
clSetKernelArg	20	6976	0.00%	348	180	1484
clSetKernelExecInfo	3	1588	0.00%	529	183	1149
clWaitForEvents	6	60864855	3.79%	10144142	928	60855555

**Table 11:** Device Performance Timing Results for Intel(R) Gen9 HD Graphics NEO (24CUs, 1200MHz)

Function Name	Calls	Time (ns)	Time (%)	Average (ns)	Min (ns)	Max (ns)
_ZTS9Matrix1_2IfE	1	58691515	99.98%	58691515	58691515	58691515
clEnqueueReadBufferRect	1	13390	0.02%	13390	13390	13390

The report includes detailed timing data on both your host and device.



## 12.0 NDA Prerelease Systems Tuning Guide

This chapter contains Intel Confidential information subjecting to the terms under a Non-Disclosure Agreement (NDA) between Intel and the party with access. The hardware and software features are subject to changes in the final released products.

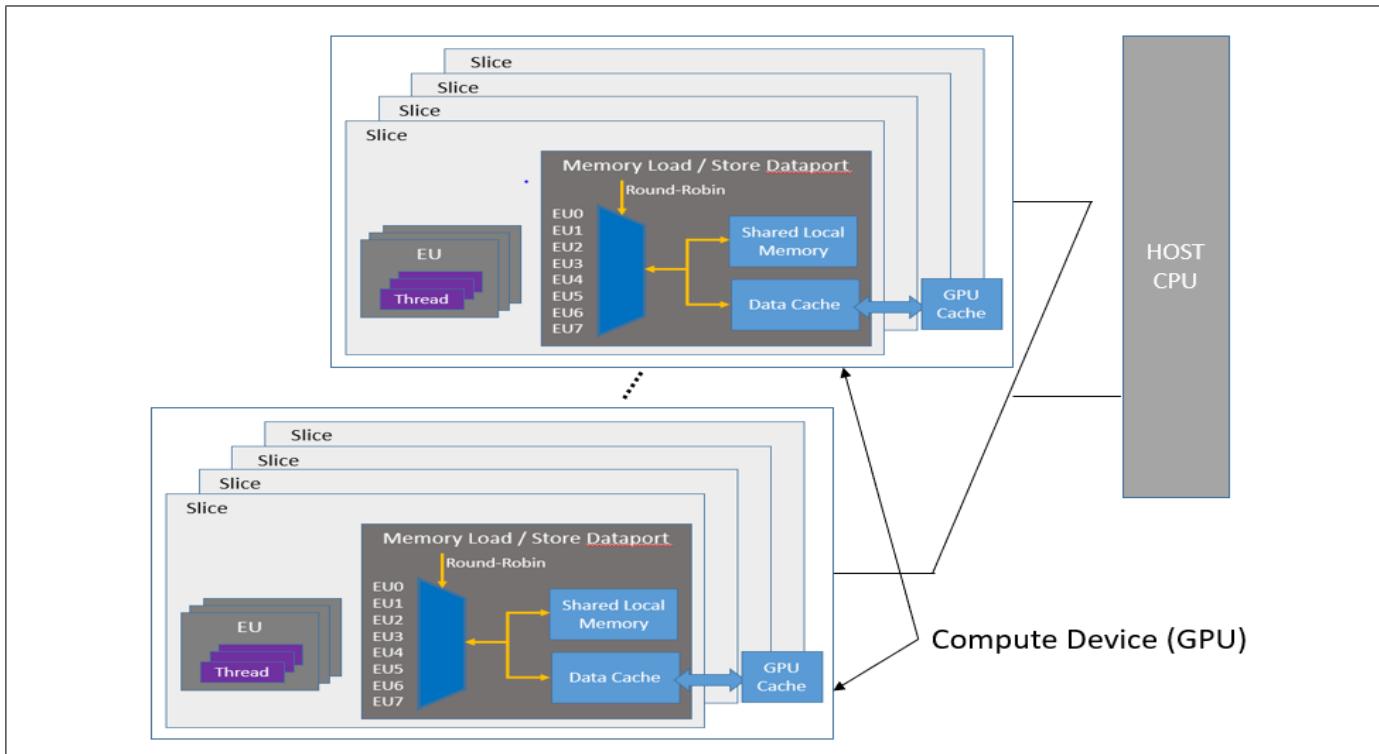
### 12.1 Multi-GPU, Multi-Tile and Multi-C-Slice Architecture and Programming

Intel® Iris® Xe-HP Arctic Sound (ATS) and Intel® Iris® Xe-HPC Ponte Vecchio (PVC) families of GPUs use a multi-tile GPU architecture with 1 to 4 tiles. The new GPU architecture and products enable Multi-GPU, Multi-tile and Multi-C-slice computing.

In this chapter, we introduce the following topics in Multi-GPU, Multi-tile and Multi-C-slice heterogeneous computing:

#### 12.1.1 GPU Execution Model Overview

The General Purpose GPU (GPGPU) compute model consists of a host connected to one or more compute devices. Each compute device consists of several Execution Units (EUs), also known as Cores, caches, shared local memory high-bandwidth memory (HBM), etc. as shown in the figure **General Purpose Compute Model**. Applications are then architected as a combination of host software (per the host framework), and kernels that are submitted by the host to run on the EUs with a pre-defined decoupling point.



**Fig. 32:** General Purpose Compute Model

The GPGPU compute architecture contains two distinct units of execution: a host program and a set of kernels that execute within the context set by the host. The host interacts with these kernels through a command queue. Each device may have its own command queue. When a command is submitted into the command queue, the command is checked for dependencies and then executed on an EU inside the compute unit clusters. Once the command has finished executing, the kernel communicates an end of life cycle through “end of thread” message.

The GP execution model determines how to schedule and execute the kernels. When a kernel-enqueue command submits a kernel for execution, the command defines an index space or N-dimensional range. A kernel-instance consists of the kernel, the argument values associated with the kernel, and the parameters that define the index space. When a compute device executes a kernel-instance, the kernel function executes for each point in the defined index space or N-dimensional range.

An executing kernel function is called a work-item, and a collection of these work-items is called a work-group. A compute device manages work-items using work-groups. Individual work-items are identified by either a global ID, or a combination of the work-group ID and a local ID inside the work-group.

The work-group concept, which essentially runs the same kernel on several unit items in a group, captures the essence of data parallel computing. The EUs can organize work-items in SIMD vector format and run the same kernel on the SIMD vector, hence speeding up the compute for all such applications.

A device can compute each work-group in any arbitrary order. Also, the work-items within a single work-group execute concurrently, with no guarantee on the order of progress. A high level work-group function, like Barriers, applies to each work-item in a work-group, to facilitate the required synchronization points. Such a work-group function must be defined so that all work-items in the work-group encounter precisely the same work-group function.

Synchronization can also occur at the command level, where the synchronization can happen between commands in host command-queues. In this mode, one command can depend on execution points in another command or multiple commands.

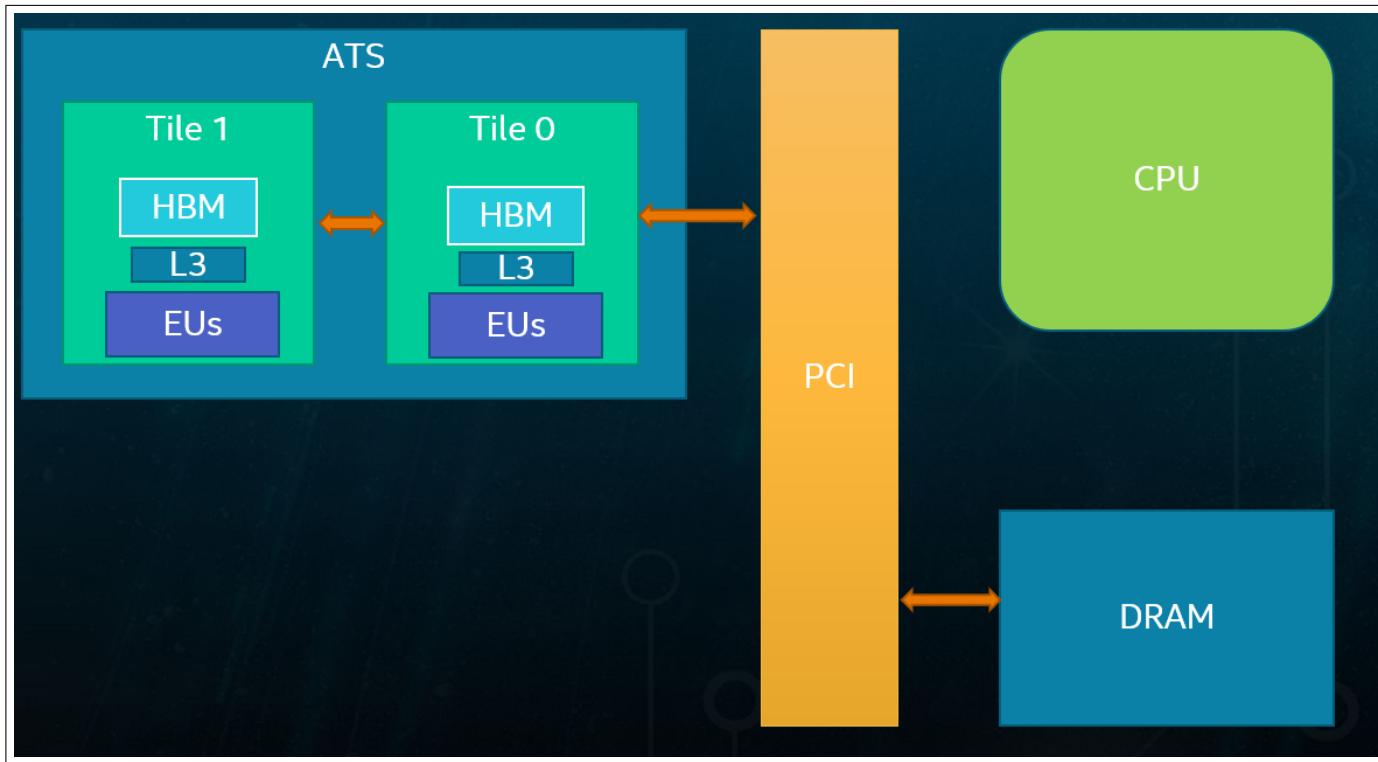
Other types of synchronization based on memory-order constraints inside a program include Atomics and Fences. These synchronization types control how a memory operation of any particular work-item is made visible to another, which offers micro-level synchronization points in the data-parallel compute model.

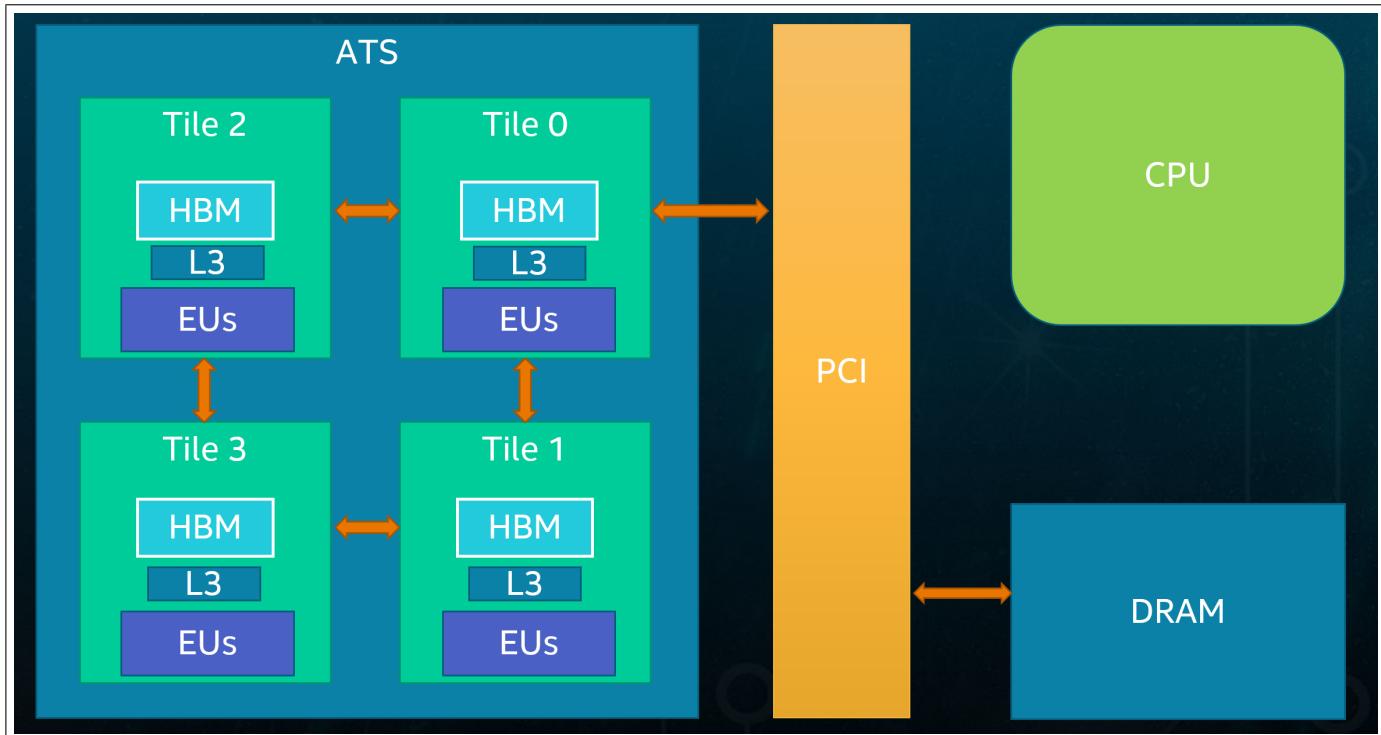
Note that an Intel GPU device is equipped with several Execution Units (EUs), while each EU is a multi-threaded SIMD processor. Compiler generates SIMD code to map several work-items to be executed simultaneously within a given hardware thread. The SIMD-width for kernel is a heuristic driven compiler choice. SIMD-8, SIMD-16, SIMD-32 are common SIMD-width examples.

For a given SIMD-width, if all kernel instances within a thread are executing the same instruction, then the SIMD lanes can be maximally utilized. If one or more of the kernel instances choose a divergent branch, then the thread executes the two paths of the branch and merges the results by mask. The EUs branch unit keeps track of such branch divergence and branch nesting.

### 12.1.2 Intel® Iris® Xe Multi-tile GPU Architecture

Intel® Iris® Xe-HP Arctic Sound (ATS) and Intel® Iris® Xe-HPC Ponte Vecchio (PVC) families of GPUs use a multi-tile GPU architecture with 1 to 4 tiles.





**Fig. 33:** Intel® Iris® Xe GPU Multi-tile Architecture

The figure illustrates a 2 tile and 4 tile ATS, each with its own dedicated resources:

**Execution Units (EUs)** Computation units belong to the tile

**High Bandwidth Memory (HBM)** HBM directly connected to the tile

**Level 3 Cache (L3)** Level 3 cache belongs to the tile

For general applications, the multi-tile GPU is represented as a single GPU device. Applications do not care that internally GPU is constructed out of smaller tiles, which simplifies the programming model and allows existing applications to run without any code changes. Intel GPU driver, DPC++ and OpenMP parallel language runtimes work together to automatically dispatch the workloads across the tiles.

Tiles are connected with fast interconnect that allows efficient communication between tiles using HBM. The following operations are possible:

**Any tile is capable of reading and writing to any HBM memory** For example, tile 0 may read the local HBM memory of tile 1. In this case, the interconnect between tile 0 and tile 1 is used for communication.

Tile 0 and tile 3 are not directly connected. It takes two hops (through tile 1 or tile 2) to access HBMs between them. The same rule applies to HBM access between tile 1 and tile 2.

**Tile 0 is connected to the PCI, but any tile can read and write system memory** The same inter-tile inter-connects are used to transfer the data. Hence, tile 0 has the shortest path to system memory among all the tiles.

Reading and writing to system memory do not require CPU involvement, GPU can perform DMA (Direct Memory Access) over PCI to system memory.

**Each tile is an independent entity** The tile can execute workloads on its own.

Because access to a tile's local HBM does not involve inter-tile interconnect, it is more efficient than cross-tile HBM access, with lower latency and lower inter-tile bandwidth consumption. Advanced developers can take advantage of memory locality to achieve higher performance.

To properly utilize multi-tile GPU, we introduced two application programming modes:

**Implicit scaling mode** Driver and language runtimes are responsible for work distribution and multi-tile memory placement. Application sees the GPU as one monolithic device and does not care about multi-tile architecture.

**Explicit scaling mode** User is responsible for work distribution and multi-tile memory placement. Driver and language runtimes provide tools that expose each tile as a separate subdevice that can be programmed independently of all the others.

Both modes utilize a **device** to steer the driver execution. There are two types of devices.

**Root Device** Represents a multi-tile part, containing subdevices (tiles)

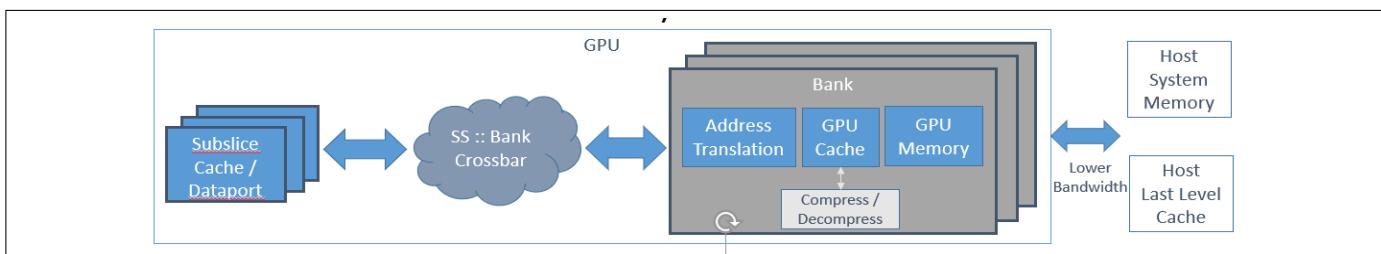
**Subdevice** Represents each tile

Driver and language runtimes utilize **device** abstraction for command dispatching and memory resource allocation. Since both root and subdevices are in fact **devices**, the same code may be used to target different modes by simply switching what **device** means in each mode.

### 12.1.3 GPU Memory System

The memory for a general-purpose engine is partitioned into host-side memory and device-side memory as shown in [GPU Memory System](#), using Unified Shared Memory (USM) to move objects between the two sides. Each address hashes to a unique bank. Approximate uniform distribution for sequential, strided, and random addresses.

- Full bandwidth when same number of banks as subslices (reduced by queuing congestion, hot-spotting, bank congestion).
- TLB misses on virtual address translation increase memory latency. May trigger more memory accesses and cache evictions.
- High miss rate on GPU cache may decrease GPU Memory controller efficiency when presented with highly distributed accesses.
- Compression unit compresses adjacent cachelines. Also supports read/write of fast-clear surfaces. Improves bandwidth but also adds latency.



**Fig. 34:** GPU Memory System

GPU Memory accesses measured at EU:

- Sustained fabric bandwidth ~90% of peak
- GPU cache hit ~150 cycles, cache miss ~300 cycles. TLB miss adds 50-150 cycles
- GPU cacheline read after write to same cacheline adds ~30 cycles

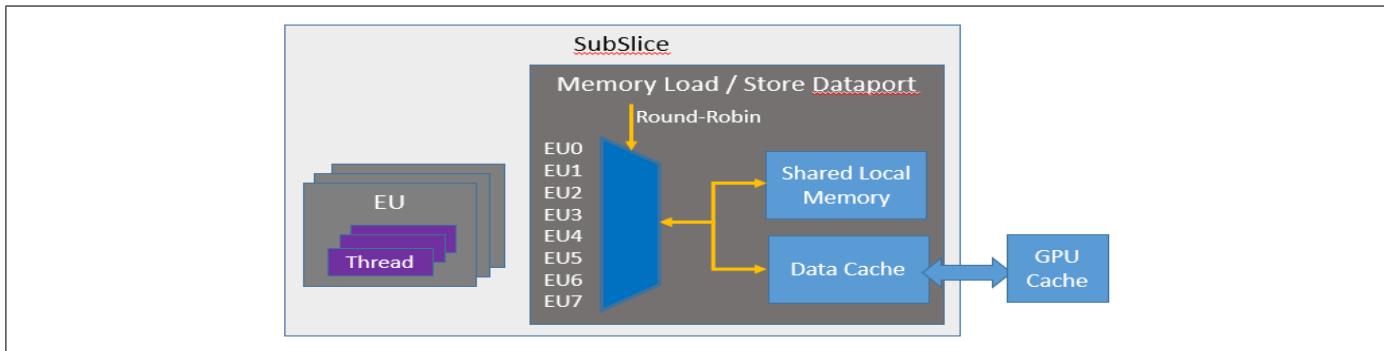
Tiles accessing device memory on a different tile utilize a new GAM-to-GAM High bandwidth interface ("GT Link") between Tiles. The bandwidth of this interface closely matches the memory bandwidth that can be handled by the device memory sub-system of any single Tile.

### Loads/Stores/Atomics in EU Threads

EU threads make memory accesses by sending messages to a data-port, the load instruction sends address and receive data, The store instructions send address and data. All EU in Subslice share one Memory Load/Store

data-port as shown in [EU thread and Memory Access](#)

- Inside subslice: ~128-256 Bytes per cycle
- Outside subslice: ~64 Bytes per cycle
- Read bandwidth sometimes higher than write bandwidth



**Fig. 35:** EU thread and Memory Access

A new memory access can be started every cycle, typical 32b SIMD16 SEND operations complete in 6 cycles plus their memory latency (4-element vectors complete in 12 cycles plus memory latency), and Independent addresses are merged to minimize memory bandwidth. Keep it mind on memory latencies:

**Table 12:** Memory latency table

Access type	Latency
Shared local memory	~30 cycles
Subslice data cache hit	~50 cycles
GPU cache hit	~150 - ~200 cycles
GPU cache miss	~300 - ~500 cycles

All Loads/Stores are relaxed ordering (ISO C11 memory model, Read and Write are in-order for the same address from the same thread. Different addresses in same thread may complete out-of-order, Read/Write ordering is not maintained between threads nor EUs nor Subslices, so code needs to use atomic operations or Fence messages to guarantee additional ordering.

An atomic operation may involve both reading from and then writing to a memory location. Atomic operations apply only to either unordered access views) or thread-group shared memory. It is guaranteed that when a thread issues an atomic operation on a memory address, no write to the same address from outside the current atomic operation by any thread can occur between the atomic read and write.

If multiple atomic operations from different threads target the same address, the operations are serialized in an undefined order. This serialization occurs due to L3 serialization rules to the same address. Atomic operations do not imply a memory or thread fence. If the program author/compiler does not make appropriate use of fences, it is not guaranteed that all threads see the result of any given memory operation at the same time, or in any particular order with respect to updates to other memory addresses. However atomic operations are always stated on a global level (except on shared local memory), when the atomic operation is complete, the final result is always visible to all thread groups. Each generation since Gen7 has increased the capability and performance of atomic operations. Gen12.5 (ATS) atomics has introduced these changes:

- Single-precision floating point atomic ADD and SUB operations (global memory only)
- Support for cross-tile global memory atomics
- Removal of PREDEC operation for both SLM and Global memory atomic
- 8-byte operands are only supported for A64 addressing mode

The following DPC++ code example performs 1024 same address atomic operations per work item. Each work item use a different (unique) address, compiler generates SIMD32 kernel for each EU thread, which will perform 2 SIMD16 atomic operations on 2 cache-lines, and compiler unrolls loop ~8 times to reduce register dependency stalls as well.

**Listing 130:** /examples/explicit-scaling/01\_memory\_order/memory\_order.cpp

```
// clang-format off
#include <CL/sycl.hpp>
#include <chrono>
#include <iostream>
#include <string>
#include <unistd.h>
#include <vector>

#ifndef SCALE
#define SCALE 1
#endif

#define N 1024*SCALE
#define SG_SIZE 32

// Number of repetitions
constexpr int repetitions = 16;
```

(continues on next page)

(continued from previous page)

```
constexpr int warm_up_token = -1;

static auto exception_handler = [] (sycl::exception_list eList) {
    for (std::exception_ptr const &e : eList) {
        try {
            std::rethrow_exception(e);
        } catch (std::exception const &e) {
            std::cout << "Failure" << std::endl;
            std::terminate();
        }
    }
};

class Timer {
public:
    Timer() : start_(std::chrono::steady_clock::now()) {}

    double Elapsed() {
        auto now = std::chrono::steady_clock::now();
        return std::chrono::duration_cast<Duration>(now - start_).count();
    }

private:
    using Duration = std::chrono::duration<double>;
    std::chrono::steady_clock::time_point start_;
};

#endif FLUSH_CACHE
void flush_cache(sycl::queue &q, sycl::buffer<int> &flush_buf) {
    auto flush_size = flush_buf.get_size()/sizeof(int);
    auto ev = q.submit([&] (auto &h) {
        sycl::accessor flush_acc(flush_buf, h, sycl::write_only, sycl::noinit);
        h.parallel_for(flush_size, [=] (auto index) { flush_acc[index] = 1; });
    });
    ev.wait_and_throw();
}
#endif

void atomicLatencyTest(sycl::queue &q, sycl::buffer<int> inbuf,
                      sycl::buffer<int> flush_buf, int &res, int iter) {
    const size_t data_size = inbuf.get_size()/sizeof(int);

    sycl::buffer<int> sum_buf(&res, 1);

    double elapsed = 0;

    for (int k = warm_up_token; k < iter; k++) {
#ifndef FLUSH_CACHE
        flush_cache(q, flush_buf);
#endif
    }
}
```

(continues on next page)

(continued from previous page)

```

Timer timer;

q.submit([&](auto &h) {
    sycl::accessor buf_acc(inbuf, h, sycl::write_only, sycl::noinit);

    h.parallel_for(sycl::nd_range<1>(sycl::range<>{N}, sycl::range<>{SG_SIZE}), [=](sycl::nd_
->item<1> item) [[intel::reqd_sub_group_size(SG_SIZE)]] {
        int i = item.get_global_id(0);
        for (int ii = 0; ii < 1024; ++ii) {
            auto v =
            #ifdef ATOMIC_RELAXED
                sycl::ONEAPI::atomic_ref<int, sycl::ONEAPI::memory_order::relaxed,
                sycl::ONEAPI::memory_scope::device,
                sycl::access::address_space::global_space>(buf_acc[i]);
            #else
                sycl::ONEAPI::atomic_ref<int, sycl::ONEAPI::memory_order::acq_rel,
                sycl::ONEAPI::memory_scope::device,
                sycl::access::address_space::global_space>(buf_acc[i]);
            #endif
            v.fetch_add(1);
        }
    });
    q.wait();
    elapsed += (iter == warm_up_token) ? 0 : timer.Elapsed();
}
std::cout << "SUCCESS: Time atomicLatency = " << elapsed << "s" << std::endl;
}

int main(int argc, char *argv[]) {
    sycl::queue q{sycl::gpu_selector{}, exception_handler};
    std::cout << q.get_device().get_info<sycl::info::device::name>() << std::endl;

    std::vector<int> data(N);
    std::vector<int> extra(N);

    for (size_t i = 0; i < N ; ++i) {
        data[i] = 1;
        extra[i] = 1;
    }
    int res=0;

    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
    sycl::buffer<int> buf(data.data(), data.size(), props);
    sycl::buffer<int> flush_buf(extra.data(), extra.size(), props);
    atomicLatencyTest(q, buf, flush_buf, res, 16);
}

```

Based on ATS hardware design for atomic, the acq\_rel-atomics should be ~2x faster than relaxed-atomics in ATS, The effective rate is ~8 ops/cycle for atomic acq\_rel operations, and ~4ops/cycles for atomic relaxed operations, assuming a uniform subslice -> bank distribution, so the single bank request pattern is as (M request-

$s/SS) * (N SS) / (N Banks)$ . Arbitration at EU, Subslice, Bank, and memory fabric is more complicated than this simple pattern shown in the code example, when the problem size is small (<16K), atomic relaxed operation shows a better performance.

**Table 13:** Atomics performance of acquire-release vs relaxed memory order

	1K	8K	16K	1M	8M	16M	32M
atomic relaxed	0.0017	0.0027	0.0047	0.1765	1.3931	2.7816	5.5599
atomic acq_rel	0.0039	0.0040	0.0042	0.1059	0.8317	1.6614	3.3177
acq_rel gain/loss	44.90%	67.42%	112.23%	166.69%	167.50%	167.43%	167.58%

In real workloads with atomics, users need to understand memory access behaviors and data set size when select a beneficial atomic operation to achieve optimal bandwidth (8 ops/cycle).

#### 12.1.4 Unified Shared Memory Allocations

Unified Shared Memory (USM) allows a program to use C/C++ pointers for memory access. There are three ways to allocate memory in DPC++:

`malloc_device`:

- Allocation can only be accessed by the specified device but not by other devices in the context nor by host.
- The data stays on the device all the time and thus is the fastest choice for kernel execution.
- Explicit copy is needed to transfer data to the host or other devices in the context.

`malloc_host`:

- Allocation can be accessed by the host and any other device in the context.
- The data stays on the host all the time and is accessed via PCI from the devices.
- No explicit copy is needed for synchronizing of the data with the host or devices.

`malloc_shared`:

- Allocation can be accessed by the host and the specified device only.
- The data can migrate (operated by the Level-Zero driver) between the host and the device for faster access.
- No explicit copy is necessary for synchronizing between the host and the device, but it is needed for other devices in the context.

The three kinds of memory allocations and their characteristics are summarized in the table below.

**Table 14:** Memory allocation types and characteristics

Memory allocation types	Description	Host accessible	Device accessible	Location
host	allocated in host memory	yes	yes, remotely through PCIe or fabric link	host
device	allocated in device memory	no	yes	device
shared	allocated shared between host and device	yes	yes	dynamically migrate between host and device

In a multi-tile, multi-C-slice GPU environment, it is important to note that device and shared USM allocations are associated with the root device. Hence, they are accessible by all the tiles and C-slices on the same device. A program should use root device for malloc\_device and malloc\_shared allocations to avoid confusion.

## OpenMP USM Allocation API

To align with DPC++ USM model, we added 3 new OpenMP APIs as Intel extensions for users to perform memory allocations based on application, memory size and performance requirements. Their semantics and performance characteristics are detailed in the following subsections.

### Host memory allocation

This host allocation is easier to use than device allocations since we do not have to manually copy data between the host and the device. Host allocations are allocations in host memory that are accessible on both the host and the device. These allocations, while accessible on the device, cannot migrate to the device's attached memory. Instead, offloading regions that read from or write to this memory do it **remotely** through either PCIe bus or fabric link. This tradeoff between convenience and performance is something that we must take into consideration. Despite the higher access costs that host allocations can incur, there are still valid reasons to use them. Examples include rarely accessed data or large data sets that cannot fit inside device attached memory. The API to perform host memory allocation is:

```
extern void *omp_target_alloc_host(size_t size, int device_num)
```

### Device memory allocation

This kind of allocation is what users need in order to have a pointer into a device's attached memory, such as (G)DDR, or HBM on the device. Device allocations can be read from or written to by offloading regions running on a device, but they cannot be directly accessed from code executing on the host. Trying to access a device allocation on the host can result in either incorrect data or a program crashing. The API to perform device memory allocation is:

```
extern void *omp_target_alloc_device(size_t size, int device_num)
```

## Shared memory allocation

Like host allocations, shared allocations are accessible on both the host and the device. The difference between them is that shared allocations are free to migrate between host memory and device attached memory, automatically, without our intervention. If an allocation has migrated to the device, any offloading region executing on that device accessing it will do so with greater performance than remotely accessing it from the host. However, shared allocations do not give us all the benefits without any drawbacks such as page migration cost and ping-pong effects:

```
extern void *omp_target_alloc_shared(size_t size, int device_num)
```

## USM support for `omp_target_alloc` API

The OpenMP API for target memory allocation is implemented based on “`#pragma omp requires unified shared memory`” and the environment **LIBOMPTARGET\_USM\_HOST\_MEM** to support three kinds of memory allocation. The implement logic is described as below.

**Listing 131:** /examples/explicit-scaling/08\_openmp\_usm/openmp\_usm.cpp

```
1 if (exists("#pragma omp requires unified_shared_memory")) {
2     if (LIBOMPTARGET_USM_HOST_MEM == 1)
3         return "host memory";
4     else
5         return "shared memory";
6 } else {
7     return "device memory";
8 }
```

### 12.1.5 Terminology

**Table 15:** Terminologies

Term	Abbreviation	Definition
Blitter	BLT	Block Image Transferrer

continues on next page

Table 15 – continued from previous page

Term	Abbreviation	Definition
Child Thread		A branch-node or a leaf-node thread that is created by another thread. It is a kind of thread associated with the media fixed function pipeline. A child thread is originated from a thread (the parent) executing on an EU and forwarded to the Thread Dispatcher by the TS unit. A child thread may or may not have child threads depending on whether it is a branch-node or a leaf-node thread. All pre-allocated resources such as URB and scratch memory for a child thread are managed by its parent thread. See also <b>Parent Thread</b> .
Command		Directive fetched from a ring buffer in memory by the Command Streamer and routed down a pipeline. Should not be confused with instructions which are fetched by the instruction cache subsystem and executed on an EU.
Command Streamer	CS or CSI	Functional unit of the Graphics Processing Engine that fetches commands, parses them, and routes them to the appropriate pipeline.
Core		Alternative name for an EU in the multi-processor system. See <b>EU</b> .
Dual Sub-slice	DSS	The collection of Execution Units (EU) that have a common set of shared function units, such as sampler, dataport, and pixel port.
End of Thread	EOT	A message sideband signal on the Output message bus signifying that the message requester thread is terminated. A thread must have at least one SEND instruction with the EOT bit in the message descriptor field set to properly terminate.

continues on next page

Table 15 – continued from previous page

Term	Abbreviation	Definition
Exception		Type of (normally rare) interruption to EU execution of a thread's instructions. An exception occurrence causes the EU thread to begin executing the System Routine, which is designed to handle exceptions.
Execution Channel		Single lane of a SIMD operand.
Execution Size	ExecSize	Execution Size indicates the number of data elements processed by an SIMD instruction. It is one of the instruction fields and can be changed per instruction.
Execution Unit	EU	An EU is a multi-threaded processor within the multi-processor system. Each EU is a fully-capable processor containing instruction fetch and decode, register files, source operand swizzle and SIMD ALU, etc. An EU is also referred to as a Core.
Execution Unit Identifier	EUID	A 4-bit field within a thread state register (SRO) that identifies the row and column location of the EU where a thread is located. A thread can be uniquely identified by the EUID and TID.
Execution Width	ExecWidth	The width of each of several data elements that may be processed by a single SIMD instruction.
General Register File	GRF	Large read/write register file shared by all the EUs for operand sources and destinations. This is the most commonly used read-write register space organized as an array of 256-bit registers for a thread.

continues on next page

Table 15 – continued from previous page

Term	Abbreviation	Definition
General State Base Address		The Graphics Address of a block of memory-resident “state data”, which includes state blocks, scratch space, constant buffers, and kernel programs. The contents of this memory block are referenced via offsets from the contents of the General State Base Address register. See <b>Graphics Processing Engine</b> .
Graphics Processing Engine	GPE	Collective name for the Subsystem, the 3D and Media pipelines, and the Command Streamer.
Memory-mapped Input/Output	MMIO	A method for performing input/output between the CPU/GPU and peripheral devices.
Message		Messages are data packages transmitted from a thread to another thread, another shared function, or another fixed function. Message passing is the primary communication mechanism of Intel GPU architecture.
Parent Thread		A thread corresponding to a root-node or a branch-node in thread generation hierarchy. A parent thread may be a root thread or a child thread depending on its position in the thread generation hierarchy.
Resource Streamer	RS	Functional unit of the Graphics Processing Engine that examines the commands in the ring buffer in an attempt to pre-process certain long latency items for the remainder of the graphics processing.

continues on next page

Table 15 – continued from previous page

<b>Term</b>	<b>Abbreviation</b>	<b>Definition</b>
Root Thread		A root-node thread. A thread corresponds to a root-node in a thread generation hierarchy. It is a kind of thread associated with the media fixed function pipeline. A root thread is originated from the VFE unit and forwarded to the Thread Dispatcher by the TS unit. A root thread may or may not have child threads. A root thread may have scratch memory managed by TS. A root thread with children has its URB resource managed by the VFE.
Single Instruction Multiple Data	SIMD	A parallel processing architecture that exploits data parallelism at the instruction level. It can also be used to describe the instructions in such an architecture or to describe the amount of data parallelism in a particular instruction (SIMD8 for example).
Spawn		To initiate a thread for execution on an EU. Done by the thread spawner as well as most FF units in the 3D Pipeline.
Sub-Register		Subfield of a SIMD register. A SIMD register is an aligned fixed size register for a register file or a register type. For example, a GRF register, <b>r2</b> , is a 256-bits wide, 256-bit aligned register. A sub-register, <b>r2.3:d</b> , is the fourth dword of GRF register <b>r2</b> .
Subsystem		The name given to the resources shared by the FF units, including shared functions and EUs.
Surface		A rendering operand or destination, including textures, buffers, and render targets.
Surface State		State associated with a render surface.

continues on next page

Table 15 – continued from previous page

Term	Abbreviation	Definition
Surface State Base Pointer		Base address used when referencing binding table and surface state data.
Synchronized Root Thread		A root thread that is dispatched by TS upon a ‘dispatch root thread’ message.
System IP	SIP	There is one global System IP register for all the threads. From a thread’s point of view, this is a virtual read only register. Upon an exception, hardware performs some bookkeeping and then jumps to SIP.
System Routine		Sequence of instructions that handles exceptions. SIP is programmed to point to this routine, and all threads encountering an exception will call it.
Thread		An instance of a kernel program executed on an EU. The life cycle for a thread starts from the executing the first instruction after being dispatched from Thread Dispatcher to an EU to the execution of the last instruction - a <b>send</b> instruction with EOT that signals the thread termination. Threads in the system may be independent from each other or communicate with each other through Message Gateway share function.
Thread Dispatcher	TD, TDL	Functional unit that arbitrates thread initiation requests from Fixed Functions units and instantiates the threads on EUs.
Thread Identifier	TID	The field within a thread state register (SRO) that identifies which thread slots on an EU a thread occupies. A thread can be uniquely identified by the EUID and TID.

continues on next page

Table 15 – continued from previous page

Term	Abbreviation	Definition
Thread Payload		Before a thread starting execution, some amount of data is pre-loaded into the thread's GRF (starting at r0). This data is typically a combination of control information provided by the spawning entity (FF Unit) and data read from the URB.
Thread Spawner	TS	The second and the last fixed function stage of the media pipeline that initiates new threads on behalf of generic/media processing.
Unsynchronized Root Thread		A root thread that is automatically dispatched by TS.

## 12.2 Implicit Scaling on Intel® Multi-Tile GPU

A root-device is built using multiple sub-devices, also known as tiles. These tiles form a shared memory space which allows to treat a root-device as a monolithic device without the requirement of explicit communication between tiles. This section covers multi-tile programming principles using implicit scaling. When using implicit scaling, the root-device driver is responsible for distributing work to all tiles when application code launches a kernel.

### 12.2.1 Implicit Scaling Introduction

#### Usage

Implicit scaling can be enabled by exporting below environment variable:

```
export EnableWalkerPartition=1
```

This environment variable changes meaning of a DPC++/OpenMP device to root-device. No change in application code is required. A kernel submitted to DPC++/OpenMP device will utilize all tiles. Similarly, memory allocation on DPC++/OpenMP device will span across all tiles. The driver behavior is described in [Work Scheduling and Memory Distribution](#).

Note:

- **EnableWalkerPartition** is enabled by default when using OpenCL or DPC++/OpenMP with OpenCL-backend. For Level0 backends, implicit scaling is not enabled by default.
- Implicit scaling does not work when DPC++/OpenMP sub-device semantics are used.
- Do not use sub-device syntax in **ZE\_AFFINITY\_MASK**. I.e. instead of exposing tile-0 from root-device-0 (**ZE\_AFFINITY\_MASK=0.0**), you must expose root-device to driver via **ZE\_AFFINITY\_MASK=0** or by unsetting **ZE\_AFFINITY\_MASK**.

## Performance Expectations

Implicit scaling exposes resources of all tiles to a single kernel launch. For root-device with 2 tiles, a kernel has access to 2x compute peak, 2x memory bandwidth and 2x memory capacity. In the ideal case, workload performance increases by 2x. However, cache size and cache bandwidth are increased by 2x as well which can lead to better-than-linear scaling if workload fits in increased cache capacity.

Each tile is equivalent to a NUMA domain and therefore memory access pattern and memory allocation are a crucial part to achieve optimal implicit scaling performance. Workloads with a concept of locality are expected to work best with this programming model as cross-tile memory accesses are naturally minimized. Note that compute bound kernels are not impacted by NUMA domains, thus are expected to easily scale to multiple tiles with implicit scaling.

MPI applications are more efficient with implicit scaling compared to an explicit scaling approach. A single rank can utilize the entire root-device which eliminates explicit synchronization and communication between tiles. Implicit scaling automatically overlaps local memory accesses and cross-tile memory accesses in a single kernel launch.

Implicit scaling improves kernel execution time only. Serial bottlenecks will not speed up. Applications will observe no speed-up with implicit scaling if large serial bottleneck is present. Common serial bottlenecks are:

- high CPU usage
- kernel launch latency
- PCIe transfers

These will become more pronounced as kernel execution time reduces. Note that only tile-0 has PCIe connection to the host. On ATS with implicit scaling enabled, kernel launch latency increases by about 10 usec.

### 12.2.2 Work Scheduling and Memory Distribution

The root-device driver uses deterministic heuristics to distribute work-groups and memory pages to all tiles when implicit scaling is used. These heuristics are described in the next two sections.

#### Memory Coloring

Any allocation in DPC++/OpenMP that corresponds to a shared or device allocation is colored across all tiles, meaning that allocation is divided in number-of-tiles chunks and distributed round-robin between tiles. Consider this root-device allocation:

OpenMP:

```
int *a = (int*)omp_target_alloc( sizeof(int)*N, device_id );
```

DPC++:

```
int *a = sycl::malloc_device<int>(N, q);
```

For a 2-tile root-device, the first half, i.e. elements  $a[0]$  to  $a[N/2-1]$ , is physically allocated on tile-0. The remaining half, i.e. elements  $a[N/2]$  to  $a[N-1]$ , is located on tile-1. In the future, we will introduce memory allocation APIs that allow user-defined memory coloring.

Note:

- Memory coloring is applied at page size granularity.
- Allocations smaller or equal than page size are resident on tile-0 only.
- Using a memory pool that is based on a single allocation will break memory coloring logic. It is recommended that applications create one allocation per object to allow that object data is distributed to all tiles.

## Static Partitioning

Scheduling of work-groups to tiles is deterministic and referred to as static partitioning. The partitioning follows a simple rule: the slowest moving dimension is divided in number-of-tiles chunks and distributed round-robin between tiles. Let's look at 1-dimensional kernel launch on root-device:

OpenMP:

**Listing 132:** /examples/implicit-scaling/01\_wg\_partitioning/wg\_partitioning\_1D.cpp

```
#pragma omp target teams distribute parallel for simd
for (int i = 0; i < N; ++i)
{
    //
}
```

DPC++:

**Listing 133:** /examples/implicit-scaling/01\_wg\_partitioning/wg\_partitioning\_1D\_sycl.cpp

```
q.parallel_for(N, [=](auto i) {
    //
});
```

Since there is only a single dimension it is automatically slowest dimension and partitioned between tiles by driver. For a 2-tile root-device, iterations 0 to N/2-1 are scheduled to tile-0. The remaining iterations N/2 to N-1 are executed on tile-1.

For OpenMP, the slowest moving dimension is outer most loop when collapse clause is used. For DPC++, the slowest moving dimension is the first element of global range. E.g. consider this 3D kernel launch:

OpenMP:

**Listing 134:** /examples/implicit-scaling/01\_wg\_partitioning/wg\_partitioning\_3D.cpp

```
#pragma omp target teams distribute parallel for simd collapse(3)
for (int z = 0; z < nz; ++z)
{
    for (int y = 0; y < ny; ++y)
    {
        for (int x = 0; x < nx; ++x)
        {
```

(continues on next page)

(continued from previous page)

```

        //
    }
}
}
```

DPC++:

**Listing 135:** /examples/implicit-scaling/01\_wg\_partitioning/wg\_partitioning\_3D\_sycl.cpp

```

range<3> global{nz, ny, nx};
range<3> local{1, 1, 16};

cgh.parallel_for(nd_range<3>(global, local), [=](nd_item<3> item) {
    //
});
```

The slowest dimension is z and partitioned between tiles, i.e. for 2-tile root-device, all iterations from z=0 to z=nz/2-1 are executed on tile-0. The remaining iterations with z=nz/2 to z=nz-1 are scheduled to tile-1.

In case slowest moving dimension can't be divided evenly between tiles and creates an remainder imbalance larger than 5%, driver will partition next dimension if it leads to less load imbalance. This impacts kernels with odd dimensions smaller than 19 only. Examples for different kernel launches can be seen in below table (assuming local range {1,1,16}):

**Table 16:** Work group partition to tiles

nz	ny	nx	Partitioned Dimension
512	512	512	z
21	512	512	z
19	512	512	y
18	512	512	z
19	19	512	x

In case of multi-dimensional local range in DPC++, the partitioned dimension can change. E.g. for global range {38,512,512} with local range {2,1,8} driver would partition y-dimension while for local range {1,1,16} driver would partition z-dimension. OpenMP can only have a 1-dimensional local range which is created from inner most loop and thus does not impact static partitioning heuristics. OpenMP kernels created with collapse level larger than 3 correspond to 1-dimensional kernel with all for loops linearized. The linearized loop will be portioned following 1D kernel launch heuristics.

Note:

- Static partitioning happens at work-group granularity.
  - This implies that all work-items in a work-group are scheduled to same tile.
- A kernel with a single work-group is resident on tile-0 only.

### 12.2.3 Simple Examples

For given kernel:

OpenMP:

**Listing 136:** /examples/implicit-scaling/02\_mem\_partitioning/simple\_example.cpp

```
int *a = (int *)omp_target_alloc(sizeof(int) * N, device_id);

#pragma omp target teams distribute parallel for simd
for (int i = 0; i < N; ++i)
{
    a[i] = i;
}
```

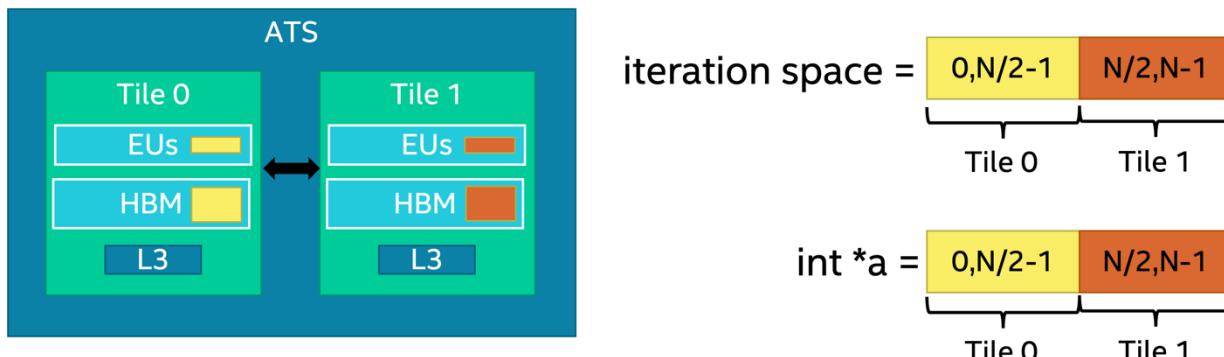
DPC++:

**Listing 137:** /examples/implicit-scaling/02\_mem\_partitioning/simple\_example\_sycl.cpp

```
int *a = sycl::malloc_device<int>(N, q);

q.parallel_for(N, [=](auto i) {
    a[i] = i;
});
```

implicit scaling guarantees 100% local memory accesses. The behavior of static partitioning and memory coloring is visualized below:



In the following section, we demonstrate implicit scaling performance for STREAM triad benchmark using 1D and 3D kernel launches on ATS.

## STREAM

Let's consider STREAM triad benchmark written in OpenMP. The main kernel is on line 40-43:

**Listing 138:** /examples/implicit-scaling/03\_stream/stream.cpp

```

1 // clang-format off
2 // Code for STREAM:
3 #include <iostream>
4 #include <omp.h>
5
6 // compile via:
7 // icpx -O2 -fopenmp -fopenmp-targets=spir64 ./stream.cpp
8
9 int main()
10 {
11     constexpr int64_t N = 256 * 1e6;
12     constexpr int64_t bytes = N * sizeof(int);
13
14     int *a = static_cast<int *>(malloc(bytes));
15     int *b = static_cast<int *>(malloc(bytes));
16     int *c = static_cast<int *>(malloc(bytes));
17
18 #pragma omp target enter data map(alloc:a[0:N])
19 #pragma omp target enter data map(alloc:b[0:N])
20 #pragma omp target enter data map(alloc:c[0:N])
21
22     for (int i = 0; i < N; ++i)
23     {
24         a[i] = i + 1;
25         b[i] = i - 1;
26     }
27
28 #pragma omp target update to(a[0:N])
29 #pragma omp target update to(b[0:N])
30
31     const int no_max_rep = 100;
32     double time;
33     for (int irep = 0; irep < no_max_rep + 10; ++irep)
34     {
35         if (irep == 10)
36             time = omp_get_wtime();
37
38 #pragma omp target teams distribute parallel for \
39             simd simdlen(32) thread_limit(256)
40         for (int i = 0; i < N; ++i)
41         {
42             c[i] = a[i] + b[i];
43         }
44     }
45     time = omp_get_wtime() - time;
46     time = time / no_max_rep;

```

(continues on next page)

(continued from previous page)

```

47
48 #pragma omp target update from(c[0:N])
49
50     for (int i = 0; i < N; ++i)
51     {
52         if (c[i] != 2 * i)
53         {
54             std::cout << "wrong results!" << std::endl;
55             exit(1);
56         }
57     }
58
59     const int64_t streamed_bytes = 3 * N * sizeof(int);
60
61     std::cout << "bandwidth = " << (streamed_bytes / time) * 1E-9
62         << " GB/s" << std::endl;
63 }
```

The benchmark runs on entire root-device just by enabling implicit scaling through `EnableWalkerPartition=1`. No code changes are required. The heuristics of static partitioning and memory coloring guarantee that each tile accesses local memory only. On 2-tile ATS system we measure 2x speed-up for STREAM triad compared to single tile. Measured bandwidth is reported in below table:

**Table 17:** Measured bandwidth with 1D kernel launch

Array Size [MB]	1-tile Bandwidth [GB/s]	Implicit Scaling Bandwidth [GB/s]	Implicit Scaling over 1-tile Speed-up
256	335	671	2x
512	337	677	2x
1024	340	678	2x

## 3D STREAM

The STREAM triad benchmark can be modified to use 3D kernel launch via collapse clause in OpenMP. The intent here is to show performance in case driver heuristics are used to partition 3D kernel launch between tiles. The kernel is on line 56-66:

**Listing 139:** /examples/implicit-scaling/04\_stream\_3D/stream\_3D.cpp

```

1 // clang-format off
2 // Code for 3D STREAM
3 #include <iostream>
4 #include <omp.h>
5 #include <cassert>
6
7 // compile via:
8 // icpx -O2 -fopenmp -fopenmp-targets=spir64 ./stream_3D.cpp
9
```

(continues on next page)

(continued from previous page)

```

10 int main(int argc, char **argv)
11 {
12     const int device_id = omp_get_default_device();
13     const int desired_total_size = 32 * 512 * 16384;
14     const std::size_t bytes = desired_total_size * sizeof(int);
15
16     std::cout << "memory footprint = " << 3 * bytes * 1E-9 << " GB"
17         << std::endl;
18
19     int *a = static_cast<int*>(omp_target_alloc_device(bytes, device_id));
20     int *b = static_cast<int*>(omp_target_alloc_device(bytes, device_id));
21     int *c = static_cast<int*>(omp_target_alloc_device(bytes, device_id));
22
23     const int min = 64;
24     const int max = 32768;
25
26     for (int lx = min; lx < max; lx *= 2)
27     {
28         for (int ly = min; ly < max; ly *= 2)
29         {
30             for (int lz = min; lz < max; lz *= 2)
31             {
32                 const int total_size = lx * ly * lz;
33                 if (total_size != desired_total_size)
34                     continue;
35
36                 std::cout << "lx=" << lx << " ly=" << ly << " lz="
37                     << lz << ", ";
38
39 #pragma omp target teams distribute parallel for simd
40     for (int i = 0; i < total_size; ++i)
41     {
42         a[i] = i + 1;
43         b[i] = i - 1;
44         c[i] = 0;
45     }
46
47     const int no_max_rep = 40;
48     const int warmup = 10;
49     double time;
50     for (int irep = 0; irep < no_max_rep + warmup; ++irep)
51     {
52         if (irep == warmup) time = omp_get_wtime();
53
54 #pragma omp target teams distribute parallel for \
55             simd simdlen(32) thread_limit(64) collapse(3)
56         for (int iz = 0; iz < lz; ++iz)
57         {
58             for (int iy = 0; iy < ly; ++iy)
59             {
60                 for (int ix = 0; ix < lx; ++ix)
61                 {

```

(continues on next page)

(continued from previous page)

```

61
62         {
63             const int index = ix + iy * lx + iz * lx * ly;
64             c[index] = a[index] + b[index];
65         }
66     }
67 }
68 time = omp_get_wtime() - time;
69 time = time / no_max_rep;

70
71 const int64_t streamed_bytes = 3 * total_size * sizeof(int);

72 std::cout << "bandwidth = " << (streamed_bytes / time) * 1E-9
73 << " GB/s" << std::endl;

74
75 #pragma omp target teams distribute parallel for simd
76 for (int i = 0; i < total_size; ++i)
77 {
78     assert(c[i] == 2 * i);
79 }
80 }
81 }
82 }
83 }

84 omp_target_free(a, device_id);
85 omp_target_free(b, device_id);
86 omp_target_free(c, device_id);
87 }
```

Note that inner-most loop has stride-1 memory access pattern. If z- or y-loop were inner-most loop, performance would decrease due to generation of scatter loads and stores leading to poor cache line utilization. On 2-tile ATS with 1GB array size, we measure below performance:

**Table 18:** Measured bandwidth with 3D kernel launch

<b>nx</b>	<b>ny</b>	<b>nz</b>	<b>1-tile Bandwidth [GB/s]</b>	<b>Implicit Scaling Bandwidth [GB/s]</b>	<b>Implicit Scaling Speed-up over 1-tile</b>
64	256	16834	336	668	1.99x
16834	64	256	332	655	1.97x
256	16834	64	332	654	1.97x

As described in [Static Partitioning](#), for these loop bounds driver partitions slowest moving dimension, i.e. z, between both tiles. This guarantees that each tile accesses local memory only leading to 2x implicit scaling compared to single tile.

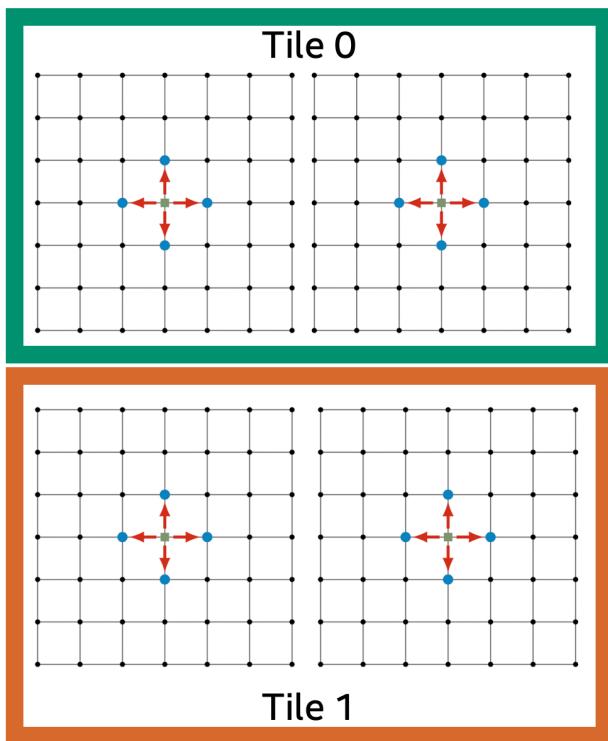
## 12.2.4 Programming Principles

To achieve good performance with implicit scaling, cross-tile memory accesses must be minimized but it is not required to eliminate all cross-tile accesses. A certain amount of cross-tile traffic can be handled by tile-to-tile interconnect if performed concurrently with local memory accesses. For memory bandwidth bound workload the amount of acceptable cross-tile accesses is determined by ratio of local memory bandwidth and cross-tile bandwidth (see [Cross-Tile Traffic](#)).

The following principles should be embraced by workloads that use implicit scaling:

- Kernel must have enough work-items to utilize both tiles.
  - The minimal number of work-items needed to utilize both tiles is: <number of EUs> \* <hardware-threads per EU> \* <SIMD width>.
  - 2T ATS with 960 EU and SIMD8 requires at least 61440 work-items.
- Device time must dominate runtime to observe whole application scaling.
- Minimize cross-tile memory accesses by exploiting locality in algorithm.
- Slowest moving dimension should be large to avoid tile load imbalance.
- Cross-tile memory accesses and local memory accesses should be interleaved.
- Avoid stride-1 memory access in slowest moving dimension for 2D and 3D kernel launches.
- If memory access pattern changes dynamically over time, a sorting step every nth iteration should be performed to minimize cross-tile memory accesses.
- Don't use a memory pool based on a single allocation (see [Memory Coloring](#)).

Many applications naturally have a concept of locality. These applications are expected to be a good fit for using implicit scaling due to low cross-tile traffic. To illustrate this concept, let's use a stencil kernel as an example. A stencil operates on a grid which can be divided into blocks where majority of stencil computations within a block use tile local data. Only stencil operations that are at border of the block require data from another block, i.e. on another tile. The amount of these cross-tile/cross-border accesses are suppressed by halo to local volume ratio. This concept is illustrated below:



## Cross-Tile Traffic

As mentioned in last section, it is crucial to minimize cross-tile traffic. To guide how much traffic can be tolerated without significantly impacting application performance we can benchmark STREAM kernel with varying amount of cross-tile traffic and compare to tile-local STREAM performance. The worst case is 100% cross-tile traffic. This is generated by reversing loop order in STREAM triad kernel (see [STREAM](#)):

**Listing 140:** /examples/implicit-scaling/06\_cross\_tile/cross\_tile\_01.cpp

```
#pragma omp target teams distribute parallel for simd
for (int i = N - 1; i >= 0; --i)
{
    c[i] = a[i] + b[i];
}
```

Here, each tile has 100% cross-tile memory traffic as work-groups on tile-0 access array elements N-1 to N/2 which are located in tile-1 memory. This kernel essentially benchmarks tile-to-tile bi-directional bandwidth. This approach can be generalized to interpolate between 0% cross-tile accesses and 100% cross-tile accesses by modified STREAM triad below:

**Listing 141:** /examples/implicit-scaling/05\_stream\_cross\_tile/stream\_cross\_tile.cpp

```
1 // clang-format off
2 // Code for cross tile stream
3 #include <iostream>
```

(continues on next page)

(continued from previous page)

```
4 #include <omp.h>
5
6 // compile via:
7 // icpx -O2 -fopenmp -fopenmp-targets=spir64 ./stream_cross_tile.cpp
8 // run via:
9 // EnableWalkerPartition=1 ZE_AFFINITY_MASK=0 ./a.out
10
11 int main()
12 {
13     constexpr int64_t N = 256 * 1e6;
14     constexpr int64_t bytes = N * sizeof(int);
15
16     // vary n_th from 1 to 8 to change cross-tile traffice
17     constexpr int n_th = 4;
18
19     std::cout << "array size = " << bytes * 1e-9 << " GB" << std::endl;
20
21     int *a = static_cast<int *>(malloc(bytes));
22     int *b = static_cast<int *>(malloc(bytes));
23     int *c = static_cast<int *>(malloc(bytes));
24
25     #pragma omp target enter data map(alloc:a[0:N])
26     #pragma omp target enter data map(alloc:b[0:N])
27     #pragma omp target enter data map(alloc:c[0:N])
28
29     for (int i = 0; i < N; ++i)
30     {
31         a[i] = i + 1;
32         b[i] = i - 1;
33     }
34
35     #pragma omp target update to(a[0:N])
36     #pragma omp target update to(b[0:N])
37
38     const int no_max_rep = 100;
39     double time;
40
41     for (int irep = 0; irep < no_max_rep + 10; ++irep)
42     {
43         if (irep == 10) time = omp_get_wtime();
44
45         #pragma omp target teams distribute parallel for \
46             simd simdlen(32) thread_limit(256)
47         for (int j = 0; j < N; ++j)
48         {
49             const int cache_line_id = j / 16;
50             int i;
51             if ((cache_line_id % n_th) == 0)
52             {
53
54                 i = (j + N / 2) % N;
```

(continues on next page)

(continued from previous page)

```

55
56     }
57     else
58     {
59         i = j;
60     }
61
62     c[i] = a[i] + b[i];
63 }
64 time = omp_get_wtime() - time;
65 time = time / no_max_rep;
66
67 #pragma omp target update from(c[0:N])
68
69 for (int i = 0; i < N; ++i)
70 {
71     if (c[i] != 2 * i)
72     {
73         std::cout << "wrong results at i " << i << std::endl;
74         exit(1);
75     }
76 }
77
78 const int64_t streamed_bytes = 3 * N * sizeof(int);
79
80 std::cout << "bandwidth = " << (streamed_bytes / time) * 1E-9 << " GB/s"
81     << std::endl;
82 std::cout << "cross-tile traffic = " << (1 / (double)n_th) * 100 << "%"
83     << std::endl;
84 }
85 }
```

The kernel on line 47-63 accesses every nth cache line cross-tile by offsetting array access with  $(j+N/2)\%N$ . For  $n\_th=1$ , we generate 100% cross-tile memory accesses. By doubling  $n\_th$  we decrease cross-tile traffic by a factor of 2. Note that this kernel is written such that cross-tile and local memory accesses are interleaved within work-groups to maximize hardware utilization. Measured performance on 2-tile ATS with 1 GB array size can be seen below:

**Table 19:** Measured bandwidth with cross-tile accesses

Partial cross-tile STREAM bandwidth [GB/s]	<b><i>n_th</i></b>	% of cross-tile accesses	% of max local 2-tile STREAM bandwidth
368	1	100%	54%
587	2	50%	87%
622	4	25%	92%
636	8	12.5%	94%

The impact of 50% cross-tile accesses is low already. Compared to a local STREAM benchmark, achieved bandwidth drops by 13%. For decreasing cross-tile traffic the performance impact becomes less, e.g. 12.5%

cross-tile traffic reduces achieved bandwidth by only 6%. These numbers can be used to estimate impact of cross-tile memory accesses on application kernel execution time.

### 12.2.5 Performance Profiling

Profiling should be focused on analyzing kernel execution time using tracing tools (**ze\_tracer**, **onetrace**) or VTune. As mentioned above, implicit scaling can only speed-up kernel execution time. Full application time scaling might be suppressed due to scalar bottlenecks. Scaling must be analyzed on a per kernel basis.

To study cross-tile memory access behavior, memory bandwidth utilization should be tracked on a per kernel basis using VTune. If application is memory bandwidth bound, high memory bandwidth utilization must be observed on both tiles for entire duration of the kernel. ATS does not allow to measure cross-tile traffic using performance counters. This will be fixed on PVC.

### Arctic Sound Specific Optimizations

Implicit scaling on ATS has limitations due to non-coherent L3 cache. Each tile has its own L3 cache which can cache memory from both tiles. There is no hardware mechanism to guarantee coherency, thus critical operations are bypassing L3 cache and use HBM instead to achieve coherency and application correctness. The root-device driver ensures coherency for following operations through HBM:

- partial writes, i.e. all memory writes that access less than a full cache line
- atomics

The impact of these operations bypassing L3 cache is severe if application has heavy usage of partial writes or atomics. This limitation applies to ATS only and will be fixed with PVC. Driver environmental variables are exposed to user for controlling partial write and atomic behavior for performance debugging purpose only. They are described in next two sections.

### Partial Writes

Special handling of partial writes, i.e. bypassing of L3 cache, can be disabled via:

```
export ForceMultiGpuPartialWrites=1
```

The default for implicit scaling is ForceMultiGpuPartialWrites=0 to ensure coherency. Note that disabling special handling of partial writes will cause wrong results in applications. Use this control only to find out if performance is limited by partial writes. If you measure better performance compared to default behavior, your application has a significant number of partial writes. An application with partial writes must be optimized to remove partial writes as this also improves bandwidth efficiency for single tile runs. A common optimization to remove partial writes is a data layout conversion from Array of Structs (AoS) to Struct of Arrays (SoA). In a 2-tile ATS VTune profile, an application will show a significant higher read-to-write ratio from HBM compared single tile profile if partial writes are present. PVC will not have this limitation.

## Atoms

Special handling of atomics, i.e. bypassing of L3 cache, can be disabled via:

```
export ForceMultiGpuAtomics=1
```

The default for implicit scaling is ForceMultiGpuPartialWrites=0. Disabling special handling of atomics will cause wrong results. Use this control only to find out if performance is limited by special handling of atomics. PVC will not have this limitation. To improve performance on ATS, you can reduce usage of atomics, if possible.

## 12.3 Explicit Scaling on Multi-GPU, Multi-Tile and Multi-C-Slice

Under explicit scaling, programmer take direct control over work group distribution and memory placement. In this chapter, we will cover

### 12.3.1 Explicit Scaling on Multi-GPU, Multi-Tile, Multi-C-Slice in DPC++

The section we describe the DPC++/SYCL explicit scaling language API and provides usage examples on ATS-based platform for multi-GPU and multi-Tiles execution.

#### Device discovery

Before you run an application, it is recommended to run a simple “`sycl-ls`” command to find out devices are available on this platform, especially, when the run is for a performance measure, so you can ensure the run is not taking a fallback path.

#### Root-device

Intel GPUs are represented as DPC++/SYCL GPU devices, root-devices. The discovery of root-devices is best with “`sycl-ls`” tool, for example:

```
$ sycl-ls
```

```
[opencl:0] GPU : Intel(R) OpenCL HD Graphics 3.0 [21.35.020776]
[level_zero:0] GPU : Intel(R) Level-Zero 1.1 [1.1.20776]
[host:0] HOST: SYCL host platform 1.2 [1.2]
```

Note that “`sycl-ls`” shows devices from all platforms of all SYCL backends that are seen by SYCL runtime. Thus in the example above there are GPUs corresponding to the single physical GPU (managed by either OpenCL or Level-Zero backend). There are few ways to filter observable root-devices.

One is using environment variable **SYCL\_DEVICE\_FILTER** described in [EnvironmentVariables.md](#)

```
$ SYCL_DEVICE_FILTER = LEVEL_ZERO sycl-ls
```

```
[level_zero:0] GPU : Intel(R) Level-Zero 1.1 [1.1.20776]
```

Another way is to use similar SYCL API described in [FilterSelector.md](#) E.g. filter\_selector (“level\_zero”) will only see Level-Zero operated devices.

If there are multiple GPUs in a system then they will be seen as multiple different root-devices. On Linux these would be multiple SYCL root-devices of the same SYCL platform (representing Level-Zero driver). On Windows these would appear as root-devices of multiple different SYCL platforms (Level-Zero drivers).

**CreateMultipleRootDevices=N NEOReadDebugKeys=1** environment variables can be used to emulate multiple GPU cards, e.g.

```
$ CreateMultipleRootDevices=2 NEOReadDebugKeys=1 \
SYCL_DEVICE_FILTER=level_zero sycl-ls
```

```
[level_zero:0] GPU : Intel(R) Level-Zero 1.1 [1.1.20776]
[level_zero:1] GPU : Intel(R) Level-Zero 1.1 [1.1.20776]
```

## Sub-device

Some Intel GPU HW is composed of multiple tiles, e.g. 2 tile ATS. The root-device in such cases can be partitioned to sub-devices, each subdevice corresponding to a physical tile.

**Listing 142:** /examples/explicit-scaling/02\_create\_subdevices/create\_subdevices.cpp

```
1 try {
2     vector<device> SubDevices = RootDevice.create_sub_devices<
3         cl::sycl::info::partition_property::partition_by_affinity_domain>(
4             cl::sycl::info::partition_affinity_domain::numa);
5 }
```

Each call to create\_sub\_devices will return exactly the same sub-devices and in the persistent order. To control what sub-devices are exposed by Level-Zero UMD one can use **ZE\_AFFINITY\_MASK** environment variable. Note that the partition\_by\_affinity\_domain is the only partitioning supported for Intel GPU.

Similar next\_partitionable and numa are the only partitioning properties supported (both doing the same thing). **CreateMultipleRootDevices=N NEOReadDebugKeys=1** environment variables can be used to emulate multiple tiles of a GPU.

## Sub-sub-device

Each sub-device (Tile) can be further decompsoed to a set of sub-sub-device (Compute Slice). One can create a context associating with a sub-sub-device. In this scheme, the execution resource will be limited to the sub-sub-device, giving the program fine-grained control at compute slice level. The following code finds all sub-devices and sub-sub-devices of a device:

**Listing 143:** /examples/explicit-scaling/14\_explicit\_subsubdevice/ccs.cpp

```
#include <CL/sycl.hpp>
#include <iostream>
namespace sycl;
int main() {
    sycl::device d(sycl::gpu_selector{});
    std::vector<sycl::device> *subdevices = new std::vector<sycl::device>();
    std::vector<sycl::device> *CCS = new std::vector<sycl::device>();
    auto part_prop = d.get_info<sycl::info::device::partition_properties>();
    size_t num_of_tiles;
    size_t num_of_ccs;
    if (part_prop.empty()) {
        num_of_tiles = 1;
    } else {
        for (int i = 0; i < part_prop.size(); i++) {
            if (part_prop[i] ==
                sycl::info::partition_property::partition_by_affinity_domain) {
                auto sub_devices = d.create_sub_devices<
                    sycl::info::partition_property::partition_by_affinity_domain>(
                    sycl::info::partition_affinity_domain::numa);
                num_of_tiles = sub_devices.size();
                for (int j = 0; j < num_of_tiles; j++)
                    subdevices->push_back(sub_devices[j]);
                break;
            } else {
                num_of_tiles = 1;
            }
        }
    }
    std::cout << "List of Tiles:\n";
    for (int i = 0; i < num_of_tiles; i++) {
        std::cout << i << ") Device name: "
            << (*subdevices)[i].get_info<sycl::info::device::name>() << "\n";
        std::cout
            << "  Max Compute Units: "
            << (*subdevices)[i].get_info<sycl::info::device::max_compute_units>()
            << "\n";
    }
    for (int j = 0; j < num_of_tiles; j++) {
        auto part_prop1 =
            (*subdevices)[j].get_info<sycl::info::device::partition_properties>();
        if (part_prop1.empty()) {
            std::cout << "No partition support\n";
        } else {
            for (int i = 0; i < part_prop1.size(); i++) {
                if (part_prop1[i] ==
                    sycl::info::partition_property::partition_by_affinity_domain) {
                    auto ccses =
                        (*subdevices)[j]
                            .create_sub_devices<sycl::info::partition_property::
                                partition_by_affinity_domain>(

```

(continues on next page)

(continued from previous page)

```

        sycl::info::partition_affinity_domain::numa);
    num_of_ccs = ccses.size();
    for (int k = 0; k < num_of_ccs; k++)
        CCS->push_back(ccses[k]);
    break;
} else {
    num_of_ccs = 1;
}
}
}
}

std::cout << "List of Compute Command Streamers:\n";
for (int i = 0; i < CCS->size(); i++) {
    std::cout << i << ") Device name: "
        << (*CCS)[i].get_info<sycl::info::device::name>() << "\n";
    std::cout << " Max Compute Units: "
        << (*CCS)[i].get_info<sycl::info::device::max_compute_units>()
        << "\n";
}
return 0;
}

```

## Context

Contexts are used for resources isolation and sharing. A SYCL context may consist of one or multiple devices. Both root-devices and sub-devices can be within single context, but they all should be of the same SYCL platform. A SYCL program (kernel\_bundle) created against a context with multiple devices will be built to each of the root-devices in the context. For context that consists of multiple sub-devices of the same root-device only single build (to that root-device) is needed.

## Unified shared memory

Memory allocated against a root-device is accessible by all of its sub-devices (tiles). So if operating on a context with multiple sub-devices of the same root-device then you can use malloc\_device on that root-device instead of using the slower malloc\_host. Remember that if using malloc\_device you'd need an explicit copy out to the host if it necessary to see data there. Please refer to section [Unified Shared Memory Allocations](#) for the details on the 3 types of USM allocations.

## Buffer

SYCL buffers are also created against a context and under the hood are mapped to the Level-Zero USM allocation discussed above. The current mapping is following:

- For integrated device the allocations are made on host, and are accessible by the host and the device without any copying.
- Memory buffers for context with sub-devices of the same root-device (possibly including the root-device itself) are allocated on that root-device. Thus they are readily accessible by all the devices in such context. The synchronization with the host is performed by SYCL RT with map/unmap doing implicit copies when necessary.

- Memory buffers for context with devices from different root-devices in it are allocated on host (thus made accessible to all devices).

## Queue

SYCL queue is always attached to a single device in a possibly multi-device context. Some typical scenarios are the following (from most performant to least performant):

### Context associated with single sub-device

Creating a context with a single sub-device in it and the queue is attached to that sub-device (tile), in this scheme, the execution/visibility is limited to the single sub-device only, and expected to offer the best performance per tile. See a code example:

**Listing 144:** /examples/explicit-scaling/03\_single\_tile\_subdevices/single\_tile\_subdevices.cpp

```
1 try {
2     vector<device> SubDevices = ...;
3     for (auto &D : SubDevices) {
4         // Each queue is in its own context, no data sharing across them.
5         auto Q = queue(D);
6         Q.submit([&](handler &cgh) { ... });
7     }
8 }
```

### Context associated with multiple sub-devices

Creating a context with multiple sub-devices (multiple tiles) of the same root-device, in this scheme, queues are to be attached to the sub-devices effectively implementing “explicit scaling”. In this scheme, the root-device should not be passed to such context for better performance. See a code example below:

**Listing 145:** /examples/explicit-scaling/04\_multi\_tile\_subdevices/multi\_tile\_subdevices.cpp

```
1 try {
2     vector<device> SubDevices = ...;
3     auto C = context(SubDevices);
4     for (auto &D : SubDevices) {
5         // All queues share the same context, data can be shared across
6         // queues.
7         auto Q = queue(C, D);
8         Q.submit([&](handler &cgh) { ... });
9     }
10 }
```

## Context associated with root device

Creating a context with a single root-device in it and the queue is attached to that root-device, in this scheme, the work will be automatically distributed across all sub-devices/tiles via “implicit scaling” by the GPU driver, which is the most simple way to enable multi-tile HW but doesn’t offer possibility to target specific tiles. See a code example below:

**Listing 146:** /examples/explicit-scaling/05\_multi\_tile\_root/multi\_tile\_root.cpp

```

1 try {
2     // The queue is attached to the root-device, driver distributes to
3     // sub - devices, if any.
4     auto D = device(gpu_selector{});
5     auto Q = queue(D);
6     Q.submit([&](handler &cgh) { ... });
7 }
```

## Context associated with multiple root devices

Creating Contexts with multiple root-devices (multi-card), in this scheme, the most unrestrictive context with queues attached to different root-devices, which offers most sharing possibilities at the cost of slow access through host memory or explicit copies needed. See a code example:

**Listing 147:** /examples/explicit-scaling/06\_multi\_roots/multi\_roots.cpp

```

1 try {
2     auto P = platform(gpu_selector{});
3     auto RootDevices = P.get_devices();
4     auto C = context(RootDevices);
5     for (auto &D : RootDevices) {
6         // Context has multiple root-devices, data can be shared across
7         // multi - card(requires explicit copying)
8         auto Q = queue(C, D);
9         Q.submit([&](handler &cgh) { ... });
10    }
11 }
```

Depending on the chosen explicit sub-devices usage described and algorithm used, make sure to do proper memory allocation/synchronization. The following program is a full example using explicit sub-devices - one queue per tile:

**Listing 148:** /examples/explicit-scaling/07\_explicit\_subdevice/explicit\_subdevice.cpp

```

1 // clang-format off
2 // Copyright (C) 2020 Intel Corporation
3 // SPDX-License-Identifier: MIT
4
5 #include <CL/sycl.hpp>
```

(continues on next page)

(continued from previous page)

```
6 #include <algorithm>
7 #include <cassert>
8 #include <cfloat>
9 #include <iostream>
10 #include <string>
11 namespace sycl;
12
13 constexpr int num_runs = 10;
14 constexpr size_t scalar = 3;
15
16 cl_ulong triad(size_t array_size) {
17
18     cl_ulong min_time_ns0 = DBL_MAX;
19     cl_ulong min_time_ns1 = DBL_MAX;
20
21     device dev = device(gpu_selector());
22
23     std::vector<device> subdev = {};
24     subdev = dev.create_sub_devices<sycl::info::partition_property::partition_by_affinity_domain>(sycl::info::partition_affinity_domain::numa);
25
26     queue q[2] = {queue(subdev[0], property::queue::enable_profiling{}),
27                   queue(subdev[1], property::queue::enable_profiling{})};
28
29     std::cout << "Running on device: " <<
30         q[0].get_device().get_info<info::device::name>() << "\n";
31     std::cout << "Running on device: " <<
32         q[1].get_device().get_info<info::device::name>() << "\n";
33
34     double *A0 = malloc_shared<double>(array_size/2 * sizeof(double), q[0]);
35     double *B0 = malloc_shared<double>(array_size/2 * sizeof(double), q[0]);
36     double *C0 = malloc_shared<double>(array_size/2 * sizeof(double), q[0]);
37
38     double *A1 = malloc_shared<double>(array_size/2 * sizeof(double), q[1]);
39     double *B1 = malloc_shared<double>(array_size/2 * sizeof(double), q[1]);
40     double *C1 = malloc_shared<double>(array_size/2 * sizeof(double), q[1]);
41
42     for (int i = 0; i < array_size/2; i++) {
43         A0[i]= 1.0; B0[i]= 2.0; C0[i]= 0.0;
44         A1[i]= 1.0; B1[i]= 2.0; C1[i]= 0.0;
45     }
46
47     for (int i = 0; i < num_runs; i++) {
48         auto q0_event = q[0].submit([&](handler& h) {
49             h.parallel_for(array_size/2, [=](id<1> idx) {
50                 C0[idx] = A0[idx] + B0[idx] * scalar;
51             });
52         });
53
54         auto q1_event = q[1].submit([&](handler& h) {
55             h.parallel_for(array_size/2, [=](id<1> idx) {
```

(continues on next page)

(continued from previous page)

```

57         C1[idx] = A1[idx] + B1[idx] * scalar;
58     });
59 }
60
61 q[0].wait();
62 q[1].wait();
63
64 cl_ulong exec_time_ns0 =
65     q0_event.get_profiling_info<info::event_profiling::command_end>() -
66     q0_event.get_profiling_info<info::event_profiling::command_start>();
67
68 std::cout << "Tile-0 Execution time (iteration " << i << ") [sec]: "
69     << (double)exec_time_ns0 * 1.0E-9 << "\n";
70 min_time_ns0 = std::min(min_time_ns0, exec_time_ns0);
71
72 cl_ulong exec_time_ns1 =
73     q1_event.get_profiling_info<info::event_profiling::command_end>() -
74     q1_event.get_profiling_info<info::event_profiling::command_start>();
75
76 std::cout << "Tile-1 Execution time (iteration " << i << ") [sec]: "
77     << (double)exec_time_ns1 * 1.0E-9 << "\n";
78 min_time_ns1 = std::min(min_time_ns1, exec_time_ns1);
79 }
80
81 // Check correctness
82 bool error = false;
83 for ( int i = 0; i < array_size/2; i++) {
84     if ((C0[i] != A0[i] + scalar * B0[i]) || (C1[i] != A1[i] + scalar * B1[i])) {
85         std::cout << "\nResult incorrect (element " << i << " is " << C0[i] << ")\!\n";
86         error = true;
87     }
88 }
89
90 sycl::free(A0, q[0]);
91 sycl::free(B0, q[0]);
92 sycl::free(C0, q[0]);
93
94 sycl::free(A1, q[1]);
95 sycl::free(B1, q[1]);
96 sycl::free(C1, q[1]);
97
98 if (error) return -1;
99
100 std::cout << "Results are correct!\n\n";
101 return std::max(min_time_ns0, min_time_ns1);
102 }
103
104 int main(int argc, char *argv[]) {
105
106     size_t array_size;
107     if (argc > 1 ) {

```

(continues on next page)

(continued from previous page)

```

108     array_size = std::stoi(argv[1]);
109 }
110 else {
111     std::cout << "Run as ./<prognome> <arraysize in elements>\n";
112     return 1;
113 }
114 std::cout << "Running with stream size of " << array_size
115     << " elements (" << (array_size * sizeof(double))/(double)1024/1024 << "MB)\n";
116
117 cl_ulong min_time = triad(array_size);
118
119 if (min_time == -1) return 1;
120 size_t triad_bytes = 3 * sizeof(double) * array_size;
121 std::cout << "Triad Bytes: " << triad_bytes << "\n";
122 std::cout << "Time in sec (fastest run): " << min_time * 1.0E-9 << "\n";
123 double triad_bandwidth = 1.0E-09 * triad_bytes/(min_time*1.0E-9);
124 std::cout << "Bandwidth of fastest run in GB/s: " << triad_bandwidth << "\n";
125 return 0;
126 }
```

The explicit scaling of this example achieves a linear scaling ~2x on an ATS-P BO (2-tile, 480EUs/tile, 1.4GHz) system using Ahead-Of-Time (AOT) compilation. The build command is

```
dpcpp -fsycl-targets=spir64_gen -O2 -ffast-math -Xs "-device xehp" explicit-subdevice.cpp -o run.exe
```

## 12.3.2 Explicit Scaling on Multi-GPU, Multi-Tile and Multi-C-Slice in OpenMP

### Devices discovery

Before you run an application, it is recommended to run a simple “sycl-ls” command as well to find out devices are available on this platform, especially, when the run is for a performance measure, so you can ensure the run is not taking a fallback path.

### Context

OpenMP Context is implicit. The runtime creates and management context. But, we added one Intel extension to query OpenMP context (omp\_target\_get\_context(D)) for interop support.

### Unified shared memory

We added 3 new OpenMP API as Intel extensions for USM memory allocations. We also provide omp\_target\_alloc API support for USM. Please refer to [OpenMP USM Allocation API](#) section for details.

## Context associated with sub-sub-devices

Creating a context with a single sub-sub-device in it and the queue is attached to that sub-sub-device (compute slice), in this scheme, the execution/visibility is limited to the sub-sub-device only, and expected to offer the fine-grained partitioning and control at compute slice (C-slice) at level. See a code example:

**Listing 149:** /examples/explicit-scaling/09\_openmp\_subsubdevice/openmp\_subsubdevice.cpp

```

1 #define DEVKIND 1 // C-Slice
2
3 int root_id = omp_get_default_device();
4
5 #pragma omp parallel for
6 for (int id = 0; id < NUM_SUBSUBDEVICES; ++id) {
7 #pragma omp target teams distribute parallel for device(root_id)
8     subdevice(DEVKIND, id) map(...)
9     for (int i = lb(id), i < ub(id); i++) {
10         ...
11     }
12 }
```

The recommendation of using sub-sub-device scheme is for running multiple small kernels concurrently on a tile at compute-slice level, and these kernels does not have enough parallelism to utilize all compute-slices of a tile. On ATS-P BO system, each tile has 4 compute command streamers for dispatching kernels to their associated compute-slices. For each tile, we can running 4 smaller kernels concurrently. It means we can run 8 smaller kernels concurrently on a 2-tile ATS system.

## Context associated with sub-devices

Creating a context with multiple sub-devices (multiple tiles) of the same root-device, in this scheme, queues are to be attached to the sub-devices effectively implementing “explicit scaling”. In this scheme, the code will run on multiple sub-devices. See a code example below:

**Listing 150:** /examples/explicit-scaling/10\_openmp\_subdevice/openmp\_subdevice.cpp

```

1 #define DEVKIND 0 // TILE
2
3 int root_id = omp_get_default_device();
4
5 #pragma omp parallel for
6 for (int id = 0; id < NUM_SUBDEVICES; ++id) {
7 #pragma omp target teams distribute parallel for device(root_id)
8     subdevice(DEVKIND, id) map(...)
9     for (int i = lb(id), i < ub(id); i++) {
10         ...
11     }
12 }
```

## Context associated with root-device

Creating a context with a single root-device and queue is attached to that, the work will be automatically distributed across all sub-devices/tiles via “implicit scaling” by the GPU driver, which is the most simple way to enable multi-tile utilization, which doesn’t offer possibility to target specific tiles. See a code example below:

**Listing 151:** /examples/explicit-scaling/11\_openmp\_root\_device/openmp\_root\_device.cpp

```
1 int root_id = omp_get_default_device();
2
3 #pragma omp target teams distribute parallel for device(root_id) map(...)
4 for (int i = 0, i < N; i++) {
5     ...
6 }
```

## Context associated with multiple root-devices

Launching offload region to multiple root-devices (multi-card), in this scheme, the most unrestrictive context with queues attached to different root-devices, which offers most sharing possibilities at the cost of slow access through host memory or explicit copies needed. See a code example:

**Listing 152:** /examples/explicit-scaling/12\_openmp\_multi\_roots/openmp\_multi\_roots.cpp

```
1 Int num_devices = omp_get_num_devices();
2
3 #pragma omp parallel for
4 for (int root_id = 0; root_id < num_devices; root_id++) {
5 #pragma omp target teams distribute parallel for device(root_id) map(...)
6     for (int i = lb(root_id); I < ub(root_id); i++) {
7         ...
8     }
9 }
```

Depending on the chosen explicit sub-devices usage described and algorithm used, make sure to do proper memory allocation/synchronization. The following program is a full example using explicit sub-devices.

**Listing 153:** /examples/explicit-scaling/13\_openmp\_explicit\_subdevice/openmp\_explicit\_subdevice.cpp

```
1 #include <assert.h>
2 #include <iostream>
3 #include <omp.h>
4 #include <stdint.h>
5 #ifndef NUM_SUBDEVICES
6 #define NUM_SUBDEVICES 1
7 #endif
8
9 #ifndef DEVKIND
10 #define DEVKIND 0
```

(continues on next page)

(continued from previous page)

```

11 #endif
12
13 template <int num_subdevices> struct mptr { float *p[num_subdevices]; };
14
15 int main(int argc, char **argv) {
16     constexpr int SIZE = 8e6;
17     constexpr int SIMD_SIZE = 32;
18     constexpr std::size_t TOTAL_SIZE = SIZE * SIMD_SIZE;
19     constexpr int num_subdevices = NUM_SUBDEVICES;
20
21     mptr<num_subdevices> device_ptr_a;
22     mptr<num_subdevices> device_ptr_b;
23     mptr<num_subdevices> device_ptr_c;
24
25     const int device_id = omp_get_default_device();
26     std::cout << "device_id = " << device_id << std::endl;
27
28     for (int sdev = 0; sdev < num_subdevices; ++sdev) {
29         device_ptr_a.p[sdev] =
30             static_cast<float *> malloc(TOTAL_SIZE * sizeof(float));
31         device_ptr_b.p[sdev] =
32             static_cast<float *> malloc(TOTAL_SIZE * sizeof(float));
33         device_ptr_c.p[sdev] =
34             static_cast<float *> malloc(TOTAL_SIZE * sizeof(float));
35
36 #pragma omp target enter data map(alloc
37                         : device_ptr_a.p[sdev] [0:TOTAL_SIZE]) \
38     device(device_id) subdevice(DEVKIND, sdev)
39 #pragma omp target enter data map(alloc
40                         : device_ptr_b.p[sdev] [0:TOTAL_SIZE]) \
41     device(device_id) subdevice(DEVKIND, sdev)
42 #pragma omp target enter data map(alloc
43                         : device_ptr_c.p[sdev] [0:TOTAL_SIZE]) \
44     device(device_id) subdevice(DEVKIND, sdev)
45 }
46 std::cout << "memory footprint per GPU = "
47             << 3 * (std::size_t)(TOTAL_SIZE) * sizeof(float) * 1E-9 << " GB"
48             << std::endl;
49
50 #pragma omp parallel for
51     for (int sdev = 0; sdev < num_subdevices; ++sdev) {
52         float *a = device_ptr_a.p[sdev];
53         float *b = device_ptr_b.p[sdev];
54
55 #pragma omp target teams distribute parallel for device(device_id)
56             subdevice(LEVEL, sdev)
57             for (int i = 0; i < TOTAL_SIZE; ++i) {
58                 a[i] = i + 0.5;
59                 b[i] = i - 0.5;
60             }
61     }

```

(continues on next page)

(continued from previous page)

```

62
63     const int no_max_rep = 200;
64     double time = 0.0;
65     for (int irep = 0; irep < no_max_rep + 1; ++irep) {
66         if (irep == 1)
67             time = omp_get_wtime();
68
69 #pragma omp parallel for num_threads(num_subdevices)
70     for (int sdev = 0; sdev < num_subdevices; ++sdev) {
71         float *a = device_ptr_a.p[sdev];
72         float *b = device_ptr_b.p[sdev];
73         float *c = device_ptr_c.p[sdev];
74
75 #pragma omp target teams distribute parallel for device(device_id) \
76     subdevice(LEVEL, sdev)
77         for (int i = 0; i < TOTAL_SIZE; ++i) {
78             c[i] = a[i] + b[i];
79         }
80     }
81
82
83     time = omp_get_wtime() - time;
84     time = time / no_max_rep;
85
86     const std::size_t streamed_bytes =
87         3 * (std::size_t)(TOTAL_SIZE)*num_subdevices * sizeof(float);
88     std::cout << "bandwidth = " << (streamed_bytes / time) * 1E-9 << " GB/s"
89         << std::endl;
90     std::cout << "time = " << time << " s" << std::endl;
91     std::cout.precision(10);
92
93     for (int sdev = 0; sdev < num_subdevices; ++sdev) {
# pragma omp target update from(device_ptr_c.p[sdev][:TOTAL_SIZE]) \
94         device(device_id) subdevice(LEVEL, sdev)
95         std::cout << "-GPU: device id = : " << sdev << std::endl;
96         std::cout << "target result:" << std::endl;
97         std::cout << "c[" << 0 << "] = " << device_ptr_c.p[sdev][0] << std::endl;
98         std::cout << "c[" << SIMD_SIZE - 1
99             << "] = " << device_ptr_c.p[sdev][SIMD_SIZE - 1] << std::endl;
100        std::cout << "c[" << TOTAL_SIZE / 2
101            << "] = " << device_ptr_c.p[sdev][TOTAL_SIZE / 2] << std::endl;
102        std::cout << "c[" << TOTAL_SIZE - 1
103            << "] = " << device_ptr_c.p[sdev][TOTAL_SIZE - 1] << std::endl;
104    }
105
106
107    for (int sdev = 0; sdev < num_subdevices; ++sdev) {
108        for (int i = 0; i < TOTAL_SIZE; ++i) {
109            assert((int)(device_ptr_c.p[sdev][i]) ==
110                  (int)(device_ptr_c.p[sdev][i] +
111                      device_ptr_a.p[sdev][i] * device_ptr_b.p[sdev][i]));
112    }

```

(continues on next page)

(continued from previous page)

```

113 }
114
115     for (int sdev = 0; sdev < num_subdevices; ++sdev) {
116 #pragma omp target exit data map(release : device_ptr_a.p[sdev][:TOTAL_SIZE])
117 #pragma omp target exit data map(release : device_ptr_b.p[sdev][:TOTAL_SIZE])
118 #pragma omp target exit data map(release : device_ptr_a.p[sdev][:TOTAL_SIZE])
119     }
120 }
```

The explicit scaling of this OpenMP workload achieves a linear scaling ~2x on an ATS-P BO (2-tile, 480EUs/tile, 1.4GHz) system using Ahead-Of-Time (AOT) compilation. The build command is:

```
icpx -fp-contract=fast -O2 -ffast-math -DNUM_SUBDEVICES=2 -fopenmp \
-fopenmp-targets=spir64_gen -Xopenmp-target-backend "-device xehp" \
openmp_explicit_subdevice.cpp -o run.x
```

Besides “subdevice” clause language extension, an environment variable is provided by the OpenMP offloading compiler and runtime to map an OpenMP “device” to ATS gpu card (device), tile (sub-device) and compute-slice (sub-sub-device). The environment variable is LIBOMPTARGET\_DEVICES=[device | subdevice | sub-subdevice]

### 12.3.3 Explicit Scaling Summary

Performance tuning for multi-tile dGPU imposes a tedious process given the parallelism granularity is at a finer-level, however, the fundamentals are similar to CPU performance tuning. Thus, to understand performance scaling dominator, pay attention to:

- EU utilization efficiency - how kernels utilize the execution resources of different tiles
- Data placement - how allocations are spread across the HBM of different tiles
- Thread-data affinity: where data “located” and how they are accessed in the system

In addition, there are several critical programming model concepts for application developers to keep in mind in order to select their favorite scaling scheme for productivity, portability and performance.

- Sub-devices (numa\_domains) and Sub-sub-devices (subnuma\_domains)
- Implicit and explicit scaling
- Contexts and queues
- Environment variables and program language APIs or constructs

## 12.4 OpenMP Offloading Tuning Guide

Intel® LLVM-based C/C++ and Fortran compilers, `icx`, `icpx`, and `ifx`, support OpenMP offloading onto GPUs. When using OpenMP, the programmer inserts device directives in the code to direct the compiler to offload certain parts of the application onto the GPU. Offloading compute-intensive code can yield better performance.

In this section, we cover the following topics related to OpenMP offloading, and how to improve the performance of offloaded code.

### 12.4.1 OpenMP Directives

Intel® compilers, `icx`, `icpx`, and `ifx` support various OpenMP device directives that control the offloading of computations and mapping of data onto a device. These include:

- `target`
- `teams`
- `distribute`
- `target data`
- `target enter data`
- `target exit data`
- `target update`
- `declare target`
- `dispatch`

The `target` construct specifies that a specific part of the code is to be executed on the device and how data is to be mapped to the device.

The `teams` construct creates a number of thread teams, where each team is composed of a master thread and a number of worker threads. If `teams` is specified without the `num_teams` clause, then the number of teams is implementation defined.

The `distribute` construct distributes iterations of a loop to the master thread of each team.

The `target data` construct maps variables to a device data environment. Variables are mapped for the extent of the `target data` region, according to any `map` clauses.

The `target enter data` directive specifies that variables are mapped to a device. With this directive, the map-type specified in `map` clauses must be either `to` or `alloc`.

The `target exit data` directive specifies that variables are unmapped from the device. With this directive, the map-type specified in `map` clauses must be `from`, `release`, or `delete`.

The `target update` directive makes the values of variables on the device consistent with their original host variables, according to the specified motion clauses.

The `declare target` directive specifies that variables, functions (C, C++ and Fortran), and subroutines (Fortran) are mapped to a device.

The `declare variant` directive declares a specialized variant of a base function and specifies the context in which that specialized variant is used.

The `dispatch` construct controls whether variant substitution occurs for a given function call. The `declare variant` and `dispatch` directives are useful when calling Intel® oneAPI Math Kernel Library (oneMKL) routines from within a target construct.

The `map` clause determines how an original host variable is mapped to a corresponding variable on the device. Map-types include:

- `to`: The value of the original host variable is copied to the device on entry to the target region.
- `from`: The value of the variable on the device is copied from the device to the original host variable on exit from the target region.
- `to from`: The value of the original host variable is copied to the device on entry to the target region, and copied back to the host on exit from the target region.
- `alloc`: Allocate an uninitialised copy of the original host variable on the device (values are not copied from the host to the device).

Directives can be combined. For example, the following combined directives may be used:

- `target teams`
- `target teams distribute`
- `target teams distribute parallel for`
- `target teams distribute parallel for simd`

It is recommended that combined directives be used where possible. Combined directives allow the compiler and runtime to decide how to best partition the iterations of an offloaded loop for execution on the GPU.

## 12.4.2 OpenMP Execution Model

In the OpenMP execution model, there is one host device and multiple target devices. A device is a logical execution engine with its own local storage and data environment.

When executing on ATS or PVC, the entire GPU (which is composed of 2-tiles) can be considered as a device, or each tile can be considered as a device.

OpenMP starts executing on the host. When a host thread encounters a target construct, data is transferred from the host to the device (if specified by map clauses, for example), and code in the construct is offloaded onto the device. At the end of the target region, data is transferred from the device to the host (if so specified).

By default, the host thread waits for the target region to finish before proceeding further. `nowait` on a target construct specifies that the host thread does not need to wait for the target region to finish. In other words, the `nowait` clause allows the asynchronous execution of the target region.

Synchronizations between regions of the code executing asynchronously can be achieved via the `taskwait` directive, `depend` clauses, (implicit and explicit) barriers, or other synchronization mechanisms.

### 12.4.3 Terminology

In this chapter, we use OpenMP and DPC++ terminology interchangeably to describe the partitioning of iterations of an offloaded parallel loop.

As described in the “DPC++ Thread Hierarchy and Mapping” chapter, the iterations of a parallel loop (execution range) offloaded onto the GPU are divided into work-groups, sub-groups, and work-items. The ND-range represents the total execution range, which is divided into work-groups of equal size. A work-group is a 1-, 2-, or 3-dimensional set of work-items. Each work-group can be divided into sub-groups. A sub-group represents a short range of consecutive work-items that are processed together as a SIMD vector.

The following table shows how DPC++ concepts map to OpenMP and CUDA concepts.

DPC++	OpenMP	CUDA
Work-item	OpenMP thread or SIMD lane	CUDA thread
Work-group	Team	Thread block
Work-group size	Team size	Thread block size
Number of work-groups	Number of teams	Number of thread blocks
Sub-group	SIMD chunk (simdlen = 8, 16, 32)	Warp (size = 32)
Maximum number of work-items per work-group	Thread limit	Maximum number of CUDA threads per thread block

### 12.4.4 Compiling and Running an OpenMP Application

Use the following compiler options to enable offloading onto Intel GPUs.

- fopenmp -fopenmp-targets=spir64

Add the following compiler options to enable the output of the compiler optimization report.

- qopt-report=3 -O3

By default, compilation is Just-In-Time (JIT). So the following compilation command will result in JIT compilation.

- icx -fiopenmp -fopenmp-targets=spir64\_gen <source\_file>

For Ahead-Of-Time (AOT) compilation, use:

ATS-P AO stepping:

- icx -fiopenmp -fopenmp-targets=spir64\_gen -Xopenmp-target-backend "-device xehp" -Xopenmp-target-backend "-revision\_id 1" <source\_file>

ATS-P BO stepping:

- icx -fiopenmp -fopenmp-targets=spir64\_gen -Xopenmp-target-backend "-device xehp" -Xopenmp-target-backend "-revision\_id 4" <source\_file>

The following are some device-related environment variables:

- `ZE_AFFINITY_MASK=0 . 0` (for 1-tile execution)
- `EnableWalkerPartition=1` (for 2-tile execution with implicit scaling)
- `OMP_TARGET_OFFLOAD=MANDATORY` (specifies that program execution is terminated if a device construct or device memory routine is encountered and the device is not available or is not supported by the implementation.)
- `LIBOMPTARGET_DEBUG=1` (for emitting debug information from `libomptarget.so`)

The following are some device-related runtime routines:

- `omp_target_alloc`
- `omp_target_free`
- `omp_target_memcpy`

The following runtime routines are supported by the Intel® compilers as Intel® extensions:

- `omp_target_alloc_host`
- `omp_target_alloc_device`
- `omp_target_alloc_shared`

`omp_target_free` can be called to free up the memory allocated using the above Intel® extensions.

For a listing of OpenMP features supported in the icx, icpx, and ifx compilers, see:

- [OpenMP Features and Extensions Supported in Intel® oneAPI DPC++/C++ Compiler](#)
- [Fortran Language and OpenMP Features Implemented in Intel® Fortran Compiler \(Beta\)](#)

## 12.4.5 OpenMP Offload Best Practices

In this chapter, we present best practices for improving the performance of applications that offload onto the GPU. We organize the best practices into the following categories, which are described in the sections that follow:

### Using More GPU Resources

We can improve the performance of offloaded code by using a larger number of work-items that can run in parallel, thus utilizing more GPU resources (filling up the GPU).

### Use collapse Clause

One way to increase parallelism in a loop nest is to use the `collapse` clause to collapse two or more loops in the loop nest. Collapsing results in a larger number of iterations that can run in parallel, thus using more work-items on the GPU.

In the following example, a loop nest composed of four perfectly nested loops is offloaded onto the GPU. The `parallel for` directive indicates that the outermost loop (on line 47) is parallel. The number of iterations in the loop is `BLOCKS`, which is equal to 8.

**Listing 154:** /examples/OpenMP/01\_collapse/test\_no\_collapse.cpp

```
1 // clang-format off
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <time.h>
5
6 #include <math.h>
7 #include <omp.h>
8
9 #define P 16
10 #define BLOCKS 8
11 #define SIZE (BLOCKS * P * P * P)
12
13 #define MAX 100
14 #define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
15
16 #define IDX2(i, j) (i * P + j)
17 #define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)
18
19 int main(int argc, char *argv[]) {
20     double w[SIZE];           /* output */
21     double u[SIZE], dx[P * P]; /* input */
22     int b, i, j, k, l;        /* loop counters */
23     double start, end;       /* timers */
24
25     omp_set_default_device(0);
26
27     /* dummy target region, so as not to measure startup time. */
28     #pragma omp target
29     { ; }
30
31     /* initialize input with random values */
32     srand(0);
33     for (int i = 0; i < SIZE; i++)
34         u[i] = scaled_rand();
35
36     for (int i = 0; i < P * P; i++)
37         dx[i] = scaled_rand();
38
39     /* map data to device */
40     #pragma omp target enter data map(to: u[0:SIZE], dx[0:P * P])
41
42     start = omp_get_wtime();
43
44     /* offload the kernel with no collapse clause */
45     #pragma omp target teams distribute parallel for \
46         private(b, i, j, k, l)
47     for (b = 0; b < BLOCKS; b++) {
48         for (i = 0; i < P; i++) {
49             for (j = 0; j < P; j++) {
```

(continues on next page)

(continued from previous page)

```

50     for (k = 0; k < P; k++) {
51         double ur = 0.;
52         double us = 0.;
53         double ut = 0.;
54
55         for (l = 0; l < P; l++) {
56             ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
57             us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
58             ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
59         }
60
61         w[IDX4(b, i, j, k)] = ur * us * ut;
62     }
63 }
64 }
65 }
66
67 end = omp_get_wtime();
68
69 #pragma omp target exit data map(from: w[0:SIZE])
70
71 /* print result */
72 printf("no-collapse-clause: w[0]=%lf time=%lf\n", w[0], end - start);
73
74 return 0;
75 }
```

libomp target.so debug information (emitted at runtime when the environment variable LIBOMPTARGET\_DEBUG=1) shows the ND-range partitioning of loop iterations and how parallelism is increased by using the collapse clause. In the output, Lb and Ub refer to the parallel loop lower bound and upper bound, respectively, in each dimension of the partitioning.

Without the collapse clause, LIBOMPTARGET\_DEBUG=1 output shows the following information about the target region on line 45.

**Listing 155:** /examples/OpenMP/01\_collapse/test\_no\_collapse.debug

```

Libomptarget --> Launching target execution __omp_offloading_802_b85fb2__Z4main_l45 with
↳ pointer 0x000000000ff1b48 (index=1).
Libomptarget --> Manifesting used target pointers:
Target LEVEL0 RTL --> Executing a kernel 0x000000000ff1b48...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 7, Stride = 1
Target LEVEL0 RTL --> Group sizes = {8, 1, 1}
Target LEVEL0 RTL --> Group counts = {1, 1, 1}
```

Note that without the collapse clause, the number of parallel loop iterations = 8, since the upper bound of the outermost loop (BLOCKS) = 8. In this case, we end up with one work-group that has 8 work-items (total work-group count = 1x1x1 = 1, and each work-group size = 8x1x1 = 8 work-items). The kernel is vectorized using SIMD

32, which means every 32 work-items are combined into one sub-group. Since we have only 8 work-items, it follows that we have only one sub-group where not all SIMD lanes are active.

We can increase parallelism and hence the number of work-items used on the GPU by adding a `collapse` clause on the `parallel for` directive. We start by adding the `collapse(2)` clause, as shown in the following modified example.

**Listing 156:** /examples/OpenMP/01\_collapse/test\_collapse\_2levels.cpp

```

44  /* offload the kernel with collapse clause */
45  #pragma omp target teams distribute parallel for collapse(2) \
46      private(b, i, j, k, l)
47  for (b = 0; b < BLOCKS; b++) {
48      for (i = 0; i < P; i++) {
49          for (j = 0; j < P; j++) {
50              for (k = 0; k < P; k++) {
51                  double ur = 0.;
52                  double us = 0.;
53                  double ut = 0.;

54                  for (l = 0; l < P; l++) {
55                      ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
56                      us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
57                      ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
58                  }
59
60                  w[IDX4(b, i, j, k)] = ur * us * ut;
61              }
62          }
63      }
64  }
65 }
```

LIBOMPTARGET\_DEBUG=1 output shows the following partitioning when `collapse(2)` is used.

**Listing 157:** /examples/OpenMP/01\_collapse/test\_collapse\_2levels.debug

```

Libomptarget --> Launching target execution __omp_offloading_802_b85fb3_Z4main_l45 with
→ pointer 0x0000000001dfffc98 (index=1).
Libomptarget --> Manifesting used target pointers:
Target LEVEL0 RTL --> Executing a kernel 0x0000000001dfffc98...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 16
Target LEVEL0 RTL --> Preferred group size is multiple of 32
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 15, Stride = 1
Target LEVEL0 RTL --> Level 1: Lb = 0, Ub = 7, Stride = 1
Target LEVEL0 RTL --> Group sizes = {16, 1, 1}
Target LEVEL0 RTL --> Group counts = {1, 8, 1}
```

Note that with `collapse(2)`, the number of parallel loop iterations =  $BLOCKS \times P = 8 \times 16 = 128$ . In this case, we end up with 8 work-groups, and each work-group has 16 work-items (total work-group count =  $1 \times 8 \times 1 = 8$ , and each work-group size =  $16 \times 1 \times 1 = 16$  work-items). The kernel is vectorized using SIMD 16, which means every 16 work-items are combined into one sub-group. It follows that each work-group has one sub-group.

On the other hand, if we use the collapse(3) clause, LIBOMPTARGET\_DEBUG=1 output shows the following partitioning.

**Listing 158:** /examples/OpenMP/01\_collapse/test\_collapse\_3levels.debug

```
Libomptarget --> Launching target execution __omp_offloading_802_b85fb4_Z4main_l45 with_u
↳ pointer 0x0000000000a2b9b8 (index=1).
Libomptarget --> Manifesting used target pointers:
Target LEVEL0 RTL --> Executing a kernel 0x0000000000a2b9b8...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 16
Target LEVEL0 RTL --> Preferred group size is multiple of 32
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 15, Stride = 1
Target LEVEL0 RTL --> Level 1: Lb = 0, Ub = 15, Stride = 1
Target LEVEL0 RTL --> Level 2: Lb = 0, Ub = 7, Stride = 1
Target LEVEL0 RTL --> Group sizes = {16, 1, 1}
Target LEVEL0 RTL --> Group counts = {1, 16, 8}
```

With collapse(3), the number of resulting parallel loop iterations = BLOCKS x P x P = 8 x 16 x 16 = 2048. In this case, we end up with 128 work-groups, and each work-group has 16 work-items (total work-group count = 1 x 16 x 8 = 128, and each work-group size = 16 x 1 x 1 = 16 work-items). The kernel is vectorized using SIMD 16, which means every 16 work-items are combined into one sub-group. It follows that each work-group has one sub-group.

If we were to use the collapse(4) clause, instead of collapse(3), LIBOMPTARGET\_DEBUG=1 output shows the following.

**Listing 159:** /examples/OpenMP/01\_collapse/test\_collapse\_4levels.debug

```
Libomptarget --> Launching target execution __omp_offloading_802_b85fb5_Z4main_l45 with_u
↳ pointer 0x0000000000aeec98 (index=1).
Libomptarget --> Manifesting used target pointers:
Target LEVEL0 RTL --> Executing a kernel 0x0000000000aeec98...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 16
Target LEVEL0 RTL --> Preferred group size is multiple of 32
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {32, 1, 1}
Target LEVEL0 RTL --> Group counts = {1024, 1, 1}
```

With collapse(4), the number of resulting parallel loop iterations = BLOCKS x P x P x P = 8 x 16 x 16 x 16 = 32768. In this case, the compiler and runtime decide on one-dimensional partitioning, where we have 1024 work-groups, and each work-group has 32 work-items (total work-group count = 1024 x 1 x 1 = 1024, and each work-group size = 32 x 1 x 1 = 32 work-items). The kernel is vectorized using SIMD 16, which means every 16 work-items are combined into one sub-group. It follows that each work-group has two sub-groups.

Using the collapse clause significantly reduces the runtime of the loop nest. The performance of the various versions when running on the particular ATS GPU used (1-tile only) was as follows:

```
no collapse version : 0.028665 seconds
collapse(2) version : 0.003309 seconds
collapse(3) version : 0.002016 seconds
collapse(4) version : 0.002016 seconds
```

The above timings show that adding the collapse(3) or collapse(4) clause gives a performance boost of about 14x (0.002016 seconds versus 0.028665 seconds).

Note that, on the GPU, the collapse clause may not result in any actual loop collapsing at all, but the clause conveys to the compiler and runtime the degree of parallelism in the loop nest and is used in determine the ND-range partitioning.

To take advantage of vector loads and stores, it is recommended that the innermost loop in a loop nest not be included in the collapsing so it can be vectorized. Best performance is achieved when the innermost loop has unit stride and its number of iterations is at least as large as the SIMD width.

## Minimizing Data Transfers and Memory Allocations

When offloading computations onto the GPU, it is important to minimize data transfers between the host and the device, and reduce memory allocations on the device. There are various ways to achieve this, as described below.

### Use target enter data and target exit data Directives

When variables are used by multiple target constructs, the target enter data and target exit data pair of directives can be used to minimize data transfers between the host and the device.

Place the target enter data directive before the first target construct to transfer data from the host to the device, and place the target exit data directive after the last target construct to transfer data from the device to the host.

Consider the following example where we have two target constructs (on lines 42 and 66), and each target construct reads arrays dx and u, and reads and writes to array w.

**Listing 160:** /examples/OpenMP/03\_target\_enter\_exit\_data/test\_no\_target\_enter\_exit\_data.cpp

```
1 // clang-format off
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <time.h>
5
6 #include <math.h>
7 #include <omp.h>
8
9 #define P 16
10 #define BLOCKS 8
11 #define SIZE (BLOCKS * P * P * P)
12
13 #define MAX 100
14 #define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
15
16 #define IDX2(i, j) (i * P + j)
17 #define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)
```

(continues on next page)

(continued from previous page)

```

19 int main(int argc, char *argv[]) {
20     double w[SIZE];           /* output */
21     double u[SIZE], dx[P * P]; /* input */
22     int b, i, j, k, l;        /* loop counters */
23     double start, end;       /* timers */
24
25     omp_set_default_device(0);
26
27     /* dummy target region, so as not to measure startup time. */
28     #pragma omp target
29     { ; }
30
31     /* initialize input with random values */
32     srand(0);
33     for (int i = 0; i < SIZE; i++)
34         u[i] = scaled_rand();
35
36     for (int i = 0; i < P * P; i++)
37         dx[i] = scaled_rand();
38
39     start = omp_get_wtime();
40
41     /* offload kernel #1 */
42     #pragma omp target teams distribute parallel for collapse(4) \
43         map(to: u[0:SIZE], dx[0:P * P]) map(from: w[0:SIZE]) \
44         private(b, i, j, k, l)
45     for (b = 0; b < BLOCKS; b++) {
46         for (i = 0; i < P; i++) {
47             for (j = 0; j < P; j++) {
48                 for (k = 0; k < P; k++) {
49                     double ur = 0.;
50                     double us = 0.;
51                     double ut = 0.;

52                     for (l = 0; l < P; l++) {
53                         ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
54                         us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
55                         ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
56                     }
57                 }

58                 w[IDX4(b, i, j, k)] = ur * us * ut;
59             }
60         }
61     }
62 }
63 }

64 /* offload kernel #2 */
65 #pragma omp target teams distribute parallel for collapse(4) \
66     map(to: u[0:SIZE], dx[0:P * P]) map(tofrom: w[0:SIZE]) \
67     private(b, i, j, k, l)
68     for (b = 0; b < BLOCKS; b++) {

```

(continues on next page)

(continued from previous page)

```

70   for (i = 0; i < P; i++) {
71     for (j = 0; j < P; j++) {
72       for (k = 0; k < P; k++) {
73         double ur = b + i + j - k;
74         double us = b + i + j - k;
75         double ut = b + i + j - k;
76
77         for (l = 0; l < P; l++) {
78           ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
79           us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
80           ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
81         }
82
83         w[IDX4(b, i, j, k)] += ur * us * ut;
84       }
85     }
86   }
87
88   end = omp_get_wtime();
89
90   /* print result */
91   printf("target region: w[0]=%lf time=%lf\n", w[0], end - start);
92
93   return 0;
94 }

```

When the first target construct (on line 42) is encountered:

- Since arrays dx and u appear in a map clause with the to map-type, storage is allocated for arrays dx and u on the device, and the values of dx and u on the host are copied to the corresponding arrays on the device.
- Since array w appears in a map clause with the from map-type, uninitialized storage is allocated for array w on the device.

At the end of the first target region:

- Since array w appears in a map clause with the from map-type, the values of array w on the device are copied to the original array w on the host.

When the second target construct (on line 66) is encountered:

- Since arrays dx, u, and w appear in a map clause with the to map-type, storage is allocated for arrays dx, u, and w on the device and the values of arrays dx, u, and w on the host are copied to the corresponding arrays on the device.

At the end of the second target region:

- Since array w appears in a map clause with the from map-type, the values of array w on the device are copied to the original array w on the host.

LIBOMPTARGET\_DEBUG=1 output shows that both target regions (on lines 42 and 66) have the data partitioning.

**Listing 161:** /examples/OpenMP/03\_target\_enter\_exit\_data/test\_no\_target\_enter\_exit\_data.debug

```
Libomptarget --> Launching target execution __omp_offloading_802_b85fc9__Z4main_l42 with
→ pointer 0x0000000000cf3d28 (index=1).
Libomptarget --> Manifesting used target pointers:
Target LEVEL0 RTL --> Executing a kernel 0x0000000000cf3d28...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 16
Target LEVEL0 RTL --> Preferred group size is multiple of 32
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {32, 1, 1}
Target LEVEL0 RTL --> Group counts = {1024, 1, 1}
```

**Listing 162:** /examples/OpenMP/03\_target\_enter\_exit\_data/test\_no\_target\_enter\_exit\_data.debug

```
Libomptarget --> Launching target execution __omp_offloading_802_b85fc9__Z4main_l66 with
→ pointer 0x0000000000cf3d30 (index=2).
Libomptarget --> Manifesting used target pointers:
Target LEVEL0 RTL --> Executing a kernel 0x0000000000cf3d30...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 16
Target LEVEL0 RTL --> Preferred group size is multiple of 32
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {32, 1, 1}
Target LEVEL0 RTL --> Group counts = {1024, 1, 1}
```

The amount of data transferred (for both target regions) can be seen in LIBOMPTARGET\_DEBUG=1 output by grepping for "Libomptarget --> Moving":

```
$ grep "Libomptarget --> Moving" test_no_target_enter_exit_data.debug
Libomptarget --> Moving 2048 bytes (hst:0x00007ffee944a8f0) -> (tgt:0xfffffd556aa780000)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffee93ca8f0) -> (tgt:0xfffffd556aa680000)
Libomptarget --> Moving 262144 bytes (tgt:0xfffffd556aa6c0000) -> (hst:0x00007ffee940a8f0)
Libomptarget --> Moving 2048 bytes (hst:0x00007ffee944a8f0) -> (tgt:0xfffffd556aa780000)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffee93ca8f0) -> (tgt:0xfffffd556aa680000)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffee940a8f0) -> (tgt:0xfffffd556aa6c0000)
Libomptarget --> Moving 262144 bytes (tgt:0xfffffd556aa6c0000) -> (hst:0x00007ffee940a8f0)
```

We can reduce the copying of data from the host to the device and vice versa by using the target enter data and target exit data directives as shown in the modified example below.

**Listing 163:** /examples/OpenMP/03\_target\_enter\_exit\_data/test\_target\_enter\_exit\_data.cpp

```
1 // clang-format off
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <time.h>
5
6 #include <math.h>
7 #include <omp.h>
```

(continues on next page)

(continued from previous page)

```
8  
9 #define P 16  
10#define BLOCKS 8  
11#define SIZE (BLOCKS * P * P * P)  
12  
13#define MAX 100  
14#define scaled_rand() ((rand() % MAX) / (1.0 * MAX))  
15  
16#define IDX2(i, j) (i * P + j)  
17#define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)  
18  
19 int main(int argc, char *argv[]) {  
20     double w[SIZE];           /* output */  
21     double u[SIZE], dx[P * P]; /* input */  
22     int b, i, j, k, l;        /* loop counters */  
23     double start, end;       /* timers */  
24  
25     omp_set_default_device(0);  
26  
27     /* dummy target region, so as not to measure startup time. */  
28     #pragma omp target  
29     { ; }  
30  
31     /* initialize input with random values */  
32     srand(0);  
33     for (int i = 0; i < SIZE; i++)  
34         u[i] = scaled_rand();  
35  
36     for (int i = 0; i < P * P; i++)  
37         dx[i] = scaled_rand();  
38  
39     start = omp_get_wtime();  
40  
41     /* map data to device. alloc for w avoids map(tofrom: w[0:SIZE])  
42     on target by default. */  
43     #pragma omp target enter data map(to: u[0:SIZE], dx[0:P * P]) \  
44     map(alloc: w[0:SIZE])  
45  
46     /* offload kernel #1 */  
47     #pragma omp target teams distribute parallel for collapse(4) \  
48     private(b, i, j, k, l)  
49     for (b = 0; b < BLOCKS; b++) {  
50         for (i = 0; i < P; i++) {  
51             for (j = 0; j < P; j++) {  
52                 for (k = 0; k < P; k++) {  
53                     double ur = 0.;  
54                     double us = 0.;  
55                     double ut = 0.;  
56  
57                     for (l = 0; l < P; l++) {  
58                         ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
```

(continues on next page)

(continued from previous page)

```

59         us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
60         ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
61     }
62
63     w[IDX4(b, i, j, k)] = ur * us * ut;
64 }
65 }
66 }
67 }
68
69 /* offload kernel #2 */
70 #pragma omp target teams distribute parallel for collapse(4) \
71 private(b, i, j, k, l)
72 for (b = 0; b < BLOCKS; b++) {
73     for (i = 0; i < P; i++) {
74         for (j = 0; j < P; j++) {
75             for (k = 0; k < P; k++) {
76                 double ur = b + i + j - k;
77                 double us = b + i + j - k;
78                 double ut = b + i + j - k;
79
80                 for (l = 0; l < P; l++) {
81                     ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
82                     us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
83                     ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
84                 }
85
86                 w[IDX4(b, i, j, k)] += ur * us * ut;
87             }
88         }
89     }
90 }
91
92 #pragma omp target exit data map(from: w[0:SIZE])
93
94 end = omp_get_wtime();
95
96 /* print result */
97 printf("target region: w[0]=%lf time=%lf\n", w[0], end - start);
98
99 return 0;
100 }
```

In the above modified example, when the `target enter data` directive (on line 43) is encountered:

- Since arrays `dx` and `u` appear in a `map` clause with the `to` map-type, storage is allocated for arrays `dx` and `u` on the device, and the values of arrays `dx` and `u` on the host are copied to the corresponding arrays on the device.
- Since array `w` appears in a `map` clause with the `alloc` map-type, uninitialized storage is allocated for array `w` on the device.

When the first `target` construct (on line 47) is encountered:

- The runtime checks whether storage corresponding to arrays dx, u, and w already exists on the device. Since it does, no data transfer occurs.

At the end of the first target region:

- The runtime will recognize that the storage for arrays dx, u, and w should remain on the device, and no copy back from the device to the host occurs.

When the second target construct (on line 70) is encountered:

- Again no data transfer from the host to the device occurs.

At the end of the second target region:

- The runtime will recognize that the storage for the arrays dx, u, and w should remain on the device, and no copy back from the device to the host will occur.

When the target exit data directive (on line 92) is encountered:

- Since array w appears in a map clause with the from map-type, the values of array w on the device are copied to the original array w on the host.

Using the target enter data and target exit data pair of directives reduced the runtime on the particular ATS GPU used (1-tile only):

```
No target enter/exit data version : 0.002503 seconds
target enter/exit data version      : 0.001560 seconds
```

LIBOMPTARGET\_DEBUG=1 output shows that data partitioning is the same in both examples (with and without target enter data and target exit data).

**Listing 164:** /examples/OpenMP/03\_target\_enter\_exit\_data/test\_target\_enter\_exit\_data.debug

```
Libomptarget --> Launching target execution __omp_offloading_802_b85fc8__Z4main_l47 with
→ pointer 0x0000000001d7f208 (index=1).
Libomptarget --> Manifesting used target pointers:
Target LEVEL0 RTL --> Executing a kernel 0x00000000001d7f208...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 16
Target LEVEL0 RTL --> Preferred group size is multiple of 32
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {32, 1, 1}
Target LEVEL0 RTL --> Group counts = {1024, 1, 1}
```

**Listing 165:** /examples/OpenMP/03\_target\_enter\_exit\_data/test\_target\_enter\_exit\_data.debug

```
Libomptarget --> Launching target execution __omp_offloading_802_b85fc8__Z4main_l70 with
→ pointer 0x0000000001d7f210 (index=2).
Libomptarget --> Manifesting used target pointers:
Target LEVEL0 RTL --> Executing a kernel 0x00000000001d7f210...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 16
Target LEVEL0 RTL --> Preferred group size is multiple of 32
```

(continues on next page)

(continued from previous page)

```
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {32, 1, 1}
Target LEVEL0 RTL --> Group counts = {1024, 1, 1}
```

The improvement in performance when using `target enter data` and `target exit data` came from the reduction of data transfers, where we now have the following three data transfers:

```
$ grep "Libomptarget --> Moving" test_target_enter_exit_data.debug
Libomptarget --> Moving 262144 bytes (hst:0x00007fff0188f5d0) -> (tgt:0xfffffd556aa690000)
Libomptarget --> Moving 2048 bytes (hst:0x00007fff0190f5d0) -> (tgt:0xfffffd556aa680000)
Libomptarget --> Moving 262144 bytes (tgt:0xfffffd556aa6d0000) -> (hst:0x00007fff018cf5d0)
```

## Choose map-type Appropriately

For improved performance, it is important that the map-type for a mapped variable matches how the variable is used in the target construct.

In the following example, arrays `u` and `dx` are read only in the target construct, and array `w` is written to in the target construct. However, the map-types for all these variables is (inefficiently) specified to be `tofrom`.

**Listing 166:** /examples/OpenMP/10\_map/test\_map\_tofrom.cpp

```
1 // clang-format off
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <time.h>
5
6 #include <math.h>
7 #include <omp.h>
8
9 #define P 16
10 #define BLOCKS 8
11 #define SIZE (BLOCKS * P * P * P)
12
13 #define MAX 100
14 #define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
15
16 #define IDX2(i, j) (i * P + j)
17 #define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)
18
19 int main(int argc, char *argv[]) {
20     double w[SIZE];           /* output */
21     double u[SIZE], dx[P * P]; /* input */
22     double ur, us, ut;        /* scalars */
23     int b, i, j, k, l;         /* loop counters */
24     double start, end;        /* timers */
25
26     omp_set_default_device(0);
```

(continues on next page)

(continued from previous page)

```
27  /* dummy target region, so as not to measure startup time. */
28  #pragma omp target
29  { ; }
30
31
32  /* initialize input with random values */
33  srand(0);
34  for (int i = 0; i < SIZE; i++)
35      u[i] = scaled_rand();
36
37  for (int i = 0; i < P * P; i++)
38      dx[i] = scaled_rand();
39
40  start = omp_get_wtime();
41
42  #pragma omp target teams distribute parallel for \
43  private(b, i, j, k, l) \
44  map(tofrom: u[0:SIZE], dx[0:P * P]) \
45  map(tofrom: w [0:SIZE])
46  for (int n = 0; n < SIZE; n++) {
47      k = n - (n / P) * P;
48      j = (n - k) / P;
49      i = (n - (j * P + k)) / (P * P);
50      b = n / (P * P * P);
51
52      double ur = 0.;
53      double us = 0.;
54      double ut = 0.;
55
56      for (l = 0; l < P; l++) {
57          ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
58          us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
59          ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
60      }
61
62      w[IDX4(b, i, j, k)] = ur * us * ut;
63  }
64
65  end = omp_get_wtime();
66
67  printf("offload: w[0]=%lf time=%lf\n", w[0], end - start);
68
69  return 0;
70 }
```

For better performance, the map-type for u and dx should be `to`, and the map-type for w should be `from`, as shown in the following modified example.

**Listing 167:** /examples/OpenMP/10\_map/test\_map\_to\_or\_from.cpp

```

42 #pragma omp target teams distribute parallel for \
43     private(b, i, j, k, l) \
44     map(to: u[0:SIZE], dx[0:P * P]) \
45     map(from: w [0:SIZE])
46 for (int n = 0; n < SIZE; n++) {
47     k = n - (n / P) * P;
48     j = (n - k) / P;
49     i = (n - (j * P + k)) / (P * P);
50     b = n / (P * P * P);

51
52     double ur = 0.;
53     double us = 0.;
54     double ut = 0.;

55
56     for (l = 0; l < P; l++) {
57         ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
58         us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
59         ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
60     }
}

```

Using more specific map-types (to or from, instead of tofrom), reduced the runtime on the particular ATS GPU used (1-tile only):

```

tfrom map-types version      : 0.002503 seconds
to or from map-types version : 0.001115 seconds

```

LIBOMPTARGET\_DEBUG=1 output shows that there are unnecessary data transfers between the host and the device when the tofrom map-type is used for u, dx, and w. With tofrom, there are six transfers to copy the values of u, dx, and w from the host to the device and vice-versa:

```

$ grep "Libomptarget --> Moving" test_map_tofrom.debug
Libomptarget --> Moving 2048 bytes (hst:0x00007ffe24481360) -> (tgt:0xfffffd556aa760000)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffe24401360) -> (tgt:0xfffffd556aa660000)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffe24441360) -> (tgt:0xfffffd556aa6a0000)
Libomptarget --> Moving 262144 bytes (tgt:0xfffffd556aa6a0000) -> (hst:0x00007ffe24441360)
Libomptarget --> Moving 262144 bytes (tgt:0xfffffd556aa660000) -> (hst:0x00007ffe24401360)
Libomptarget --> Moving 2048 bytes (tgt:0xfffffd556aa760000) -> (hst:0x00007ffe24481360)

```

With the more specific map-types (to or from), we see only three data transfers: two transfers to copy the values of u and dx from the host to the device, and one transfer to copy the values of w from the device to host:

```

$ grep "Libomptarget --> Moving" test_map_to_or_from.debug
Libomptarget --> Moving 2048 bytes (hst:0x00007ffe4e66b100) -> (tgt:0xfffffd556aa760000)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffe4e5eb100) -> (tgt:0xfffffd556aa660000)
Libomptarget --> Moving 262144 bytes (tgt:0xfffffd556aa6a0000) -> (hst:0x00

```

## Do Not Map Read-Only Scalar Variables

The compiler will produce more efficient code if read-only scalar variables in a target construct are not mapped, but are listed in a `firstprivate` clause on the target construct or not listed in any clause at all. (Note that when a scalar variable is not listed in any clause on the target construct, it will be `firstprivate` by default.)

Listing a read-only scalar variable on a `map(to: )` clause causes unnecessary memory allocation on the device and copying of data from the host to the device. On the other hand, when a read-only scalar is specified to be `firstprivate` on the target construct, the variable is passed as argument when launching the kernel, and no memory allocation or copying for the variable is required.

In the following example, a loop nest is offloaded onto the GPU. In the target construct, the three scalar variables, `s1`, `s2`, and `s3`, are read-only and are listed in a `map(to: )` clause.

**Listing 168:** /examples/OpenMP/05\_scalars\_fp/test\_scalars\_map.cpp

```
1 // clang-format off
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <time.h>
5
6 #include <math.h>
7 #include <omp.h>
8
9 #define P 16
10 #define BLOCKS 8
11 #define SIZE (BLOCKS * P * P * P)
12
13 #define MAX 100
14 #define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
15
16 #define IDX2(i, j) (i * P + j)
17 #define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)
18
19 int main(int argc, char *argv[]) {
20     double w[SIZE];           /* output */
21     double u[SIZE], dx[P * P]; /* input */
22     double s1, s2, s3;        /* scalars */
23     int b, i, j, k, l;        /* loop counters */
24     double start, end;        /* timers */
25
26     omp_set_default_device(0);
27
28     /* dummy target region, so as not to measure startup time. */
29     #pragma omp target
30     { ; }
31
32     /* initialize input with random values */
33     srand(0);
34     for (int i = 0; i < SIZE; i++)
35         u[i] = scaled_rand();
```

(continues on next page)

(continued from previous page)

```

36
37     for (int i = 0; i < P * P; i++)
38         dx[i] = scaled_rand();
39
40     /* initialize scalars */
41     s1 = u[SIZE / 2];
42     s2 = scaled_rand();
43     s3 = 0.145;
44
45     /* map data to device */
46     #pragma omp target enter data map(to: u[0:SIZE], dx[0:P * P])
47
48     start = omp_get_wtime();
49
50     /* offload the kernel with collapse clause */
51     #pragma omp target teams distribute parallel for collapse(4) \
52         map(to: s1, s2, s3) private(b, i, j, k, l)
53     for (b = 0; b < BLOCKS; b++) {
54         for (i = 0; i < P; i++) {
55             for (j = 0; j < P; j++) {
56                 for (k = 0; k < P; k++) {
57                     double ur = 0.;
58                     double us = 0.;
59                     double ut = 0.;

60                     for (l = 0; l < P; l++) {
61                         ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)] + s1;
62                         us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)] - s2;
63                         ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)] * s3;
64                     }
65
66                     w[IDX4(b, i, j, k)] = ur * us * ut;
67                 }
68             }
69         }
70     }
71
72     end = omp_get_wtime();
73
74     #pragma omp target exit data map(from: w[0:SIZE])
75
76     /* print result */
77     printf("collapse-clause: w[0]=%lf time=%lf\n", w[0], end - start);
78
79     return 0;
80 }

```

It is more efficient to list `s1`, `s2`, and `s3` in a `firstprivate` clause on the `target` construct, as shown in the modified example below, or not list them in any clause at all.

**Listing 169:** /examples/OpenMP/05\_scalars\_fp/test\_scalars\_fp.cpp

```

51 #pragma omp target teams distribute parallel for collapse(4) \
52     firstprivate(s1, s2, s3) private(b, i, j, k, l)
53 for (b = 0; b < BLOCKS; b++) {
54     for (i = 0; i < P; i++) {
55         for (j = 0; j < P; j++) {
56             for (k = 0; k < P; k++) {
57                 double ur = 0.;
58                 double us = 0.;
59                 double ut = 0.;

60                 for (l = 0; l < P; l++) {
61                     ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)] + s1;
62                     us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)] - s2;
63                     ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)] * s3;
64                 }
65
66                 w[IDX4(b, i, j, k)] = ur * us * ut;
67             }
68         }
69     }
70 }
71 }
```

Using `firstprivate(s1, s2, s3)`, instead of `map(to:s1, s2, s3)`, reduced the runtime on the particular ATS GPU used (1-tile only):

```

map(to:s1,s2,s3) version      : 0.001324 seconds
firstprivate(s1,s2,s3) version : 0.000730 seconds
```

LIBOMPTARGET\_DEBUG=1 output shows that data partitioning is the same in both examples (with `map(to:s1, s2, s3)` and with `firstprivate(to:s1, s2, s3)`).

**Listing 170:** /examples/OpenMP/05\_scalars\_fp/test\_scalars\_map.debug

```

Libomptarget --> Launching target execution __omp_offloading_802_b85fc0__Z4main_l51 with_u
↳ pointer 0x00000000024dbc98 (index=1).
Libomptarget --> Manifesting used target pointers:
Target LEVEL0 RTL --> Executing a kernel 0x00000000024dbc98...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 16
Target LEVEL0 RTL --> Preferred group size is multiple of 32
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {32, 1, 1}
Target LEVEL0 RTL --> Group counts = {1024, 1, 1}
```

**Listing 171:** /examples/OpenMP/05\_scalars\_fp/test\_scalars\_fp.debug

```
Libomptarget --> Launching target execution __omp_offloading_802_b85fc4__Z4main_l51 with
→ pointer 0x0000000002289c98 (index=1).
Libomptarget --> Manifesting used target pointers:
Target LEVEL0 RTL --> Executing a kernel 0x0000000002289c98...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 16
Target LEVEL0 RTL --> Preferred group size is multiple of 32
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {32, 1, 1}
Target LEVEL0 RTL --> Group counts = {1024, 1, 1}
Target LEVEL0 RTL --> Kernel Pointer argument 0 (value: 0xfffffd556aa660000) was set
→ successfully.
```

However, more device memory allocations and host-to-device data transfers occur when the `map(to:s1, s2, s3)` clause is used.

LIBOMPTARGET\_DEBUG=1 output shows the following data about memory allocations on the device when `map(to:s1, s2, s3)` clause is used.

**Listing 172:** /examples/OpenMP/05\_scalars\_fp/test\_scalars\_map.debug

```
Target LEVEL0 RTL --> Memory usage for device memory, device 0:
Target LEVEL0 RTL --> -- Allocator: Native, Pool
Target LEVEL0 RTL --> -- Requested: 1179648, 526360
Target LEVEL0 RTL --> -- Allocated: 1179648, 526432
Target LEVEL0 RTL --> -- Freed : 1179648, 262240
Target LEVEL0 RTL --> -- InUse : 0, 264192
Target LEVEL0 RTL --> -- PeakUse : 1179648, 526432
Target LEVEL0 RTL --> -- NumAllocs: 3, 6
```

Note that the memory allocated is 1,179,649 bytes, and the number of allocations (from the pool) is 6 – for the three arrays (`dx`, `u`, and `w`) and the three scalars (`s1`, `s2`, and `s3`).

In contrast, LIBOMPTARGET\_DEBUG=1 output shows less memory allocations on the device when the `firstprivate(s1, s2, s3)` clause is used. The memory allocated is reduced from 1,179,648 to 1,114,112 bytes (a reduction of 64 kilobytes), and the number of allocations (from the pool) is reduced from 6 to 3, as shown below.

**Listing 173:** /examples/OpenMP/05\_scalars\_fp/test\_scalars\_fp.debug

```
Target LEVEL0 RTL --> Memory usage for device memory, device 0:
Target LEVEL0 RTL --> -- Allocator: Native, Pool
Target LEVEL0 RTL --> -- Requested: 1114112, 526336
Target LEVEL0 RTL --> -- Allocated: 1114112, 526336
Target LEVEL0 RTL --> -- Freed : 1114112, 262144
Target LEVEL0 RTL --> -- InUse : 0, 264192
Target LEVEL0 RTL --> -- PeakUse : 1114112, 526336
Target LEVEL0 RTL --> -- NumAllocs: 2, 3
```

In addition to more memory allocations, there are more data transfers from the host to the device when the `map(to: )` clause is used. This can be seen by grepping for "Libomptarget --> Moving" in the LIBOMP-TARGET\_DEBUG=1 output:

```
$ grep "Libomptarget --> Moving" test_scalars_map.debug
Libomptarget --> Moving 262144 bytes (hst:0x00007fff85a9b000) -> (tgt:0xfffffd556aa670000)
Libomptarget --> Moving 2048 bytes (hst:0x00007fff85b1b000) -> (tgt:0xfffffd556aa660000)
Libomptarget --> Moving 8 bytes (hst:0x00007fff85b1b950) -> (tgt:0xfffffd556aa650000)
Libomptarget --> Moving 8 bytes (hst:0x00007fff85b1b958) -> (tgt:0xfffffd556aa650020)
Libomptarget --> Moving 8 bytes (hst:0x00007fff85b1b960) -> (tgt:0xfffffd556aa650040)
Libomptarget --> Moving 262144 bytes (hst:0x00007fff85adb000) -> (tgt:0xfffffd556aa6b0000)
Libomptarget --> Moving 262144 bytes (tgt:0xfffffd556aa6b0000) -> (hst:0x00007fff85adb000)
```

In contrast, when the `firstprivate(to:s1, s2, s3)` clause is used, LIBOMP TARGET\_DEBUG=1 output shows:

```
$ grep "Libomptarget --> Moving" test_scalars_fp.debug
Libomptarget --> Moving 262144 bytes (hst:0x00007ffd9d207140) -> (tgt:0xfffffd556aa670000)
Libomptarget --> Moving 2048 bytes (hst:0x00007ffd9d287140) -> (tgt:0xfffffd556aa660000)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffd9d247140) -> (tgt:0xfffffd556aa6b0000)
Libomptarget --> Moving 262144 bytes (tgt:0xfffffd556aa6b0000) -> (hst:0x00007ffd9d247140)
```

Note that in the example with `map(to:s1, s2, s3)` we have three additional data transfers, each moving 8 bytes. These transfers are for copying the values of `s1`, `s2`, and `s3` from the host to the device.

## Do Not Map Loop Bounds to Get Better ND-Range Partitioning

As mentioned above, the compiler will produce more efficient code if read-only scalar variables in a target construct are not mapped, but are listed in a `firstprivate` clause on the target construct or not listed in any clause at all.

This is especially true when the scalars in question are parallel loop bounds in the target construct. If any of the loop bounds (lower bound, upper bound, or step) are mapped, then this will result in unnecessary memory allocation on the device and copying of data from the host to the device. In addition, the loop partitioning will be affected, and we may end up with non-optimal ND-range partitioning that negatively impacts performance.

Consider the following example where a `parallel for` loop is offloaded onto the GPU. The upper bound of the for loop is the scalar variable `upper`, which is mapped by the target construct (on line 49).

**Listing 174:** /examples/OpenMP/07\_loop\_bounds/test\_loop\_bounds\_map.cpp

```
1 // clang-format off
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <time.h>
5
6 #include <math.h>
7 #include <omp.h>
```

(continues on next page)

(continued from previous page)

```
9 #define P 16
10 #define BLOCKS 8
11 #define SIZE (BLOCKS * P * P * P)
12
13 #define MAX 100
14 #define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
15
16 #define IDX2(i, j) (i * P + j)
17 #define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)
18
19 int main(int argc, char *argv[]) {
20     double w[SIZE];           /* output */
21     double u[SIZE], dx[P * P]; /* input */
22     double ur, us, ut;        /* scalars */
23     int b, i, j, k, l;        /* loop counters */
24     int upper;
25     double start, end;       /* timers */
26
27     omp_set_default_device(0);
28
29     /* dummy target region, so as not to measure startup time. */
30     #pragma omp target
31     { ; }
32
33     /* initialize input with random values */
34     srand(0);
35     for (int i = 0; i < SIZE; i++)
36         u[i] = scaled_rand();
37
38     for (int i = 0; i < P * P; i++)
39         dx[i] = scaled_rand();
40
41     upper = (int)dx[0] + SIZE;
42
43     /* map data to device */
44     #pragma omp target enter data map(to: u[0:SIZE], dx[0:P * P])
45
46     start = omp_get_wtime();
47
48     /* offload kernel */
49     #pragma omp target teams distribute parallel for private(b, i, j, k, l) \
50         map(to: upper)
51     for (int n = 0; n < upper; n++) {
52         double ur = 0.;
53         double us = 0.;
54         double ut = 0.;

55         k = n - (n / P) * P;
56         j = (n - k) / P;
57         i = (n - (j * P + k)) / (P * P);
58         b = n / (P * P * P);
```

(continues on next page)

(continued from previous page)

```

60
61     for (l = 0; l < P; l++) {
62         ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
63         us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
64         ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
65     }
66
67     w[IDX4(b, i, j, k)] = ur * us * ut;
68 }
69
70 end = omp_get_wtime();
71
72 /* map data from device */
73 #pragma omp target exit data map(from: w[0:SIZE])
74
75 printf("offload: w[0]=%lf time=%lf\n", w[0], end - start);
76
77 return 0;
78 }
```

Since upper is mapped, the value of the variable upper on the host may be different from the value of the variable upper on the device. So when the target region is offloaded at runtime, the number of loop iterations in the offloaded loop is not known on the host. In this case, the runtime (libomptarget) will use device and kernel properties to choose ND-range partitioning that fills the whole GPU.

The compiler-generated code for the offloaded loop includes an additional innermost loop (per work-item) inside the offloaded loop. If the global size selected happens to be smaller than the actual number of loop iterations, then each work-item will process multiple iterations of the original loop. If the global size selected is larger than the actual number of loop iterations, then some of the work-items will not do any work. There will be a if-condition inside the loop generated by the compiler to check this and skip the rest of the loop body.

For the above example (where upper is mapped), LIBOMPTARGET\_DEBUG=1 shows the following ND-range partitioning.

**Listing 175:** /examples/OpenMP/07\_loop\_bounds/test\_loop\_bounds\_map.debug

```

Libomptarget --> Launching target execution __omp_offloading_802_b85fdf__Z4main_l49 with
→pointer 0x0000000001e66da8 (index=1).
Libomptarget --> Manifesting used target pointers:
Target LEVEL0 RTL --> Executing a kernel 0x0000000001e66da8...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 16
Target LEVEL0 RTL --> Preferred group size is multiple of 32
Target LEVEL0 RTL --> Group sizes = {1024, 1, 1}
Target LEVEL0 RTL --> Group counts = {240, 1, 1}
```

Note that in the above partitioning, the total number of work-items =  $240 \times 1024 = 245,760$ , which is larger than the actual number of loop iterations (32,767). So some of the work-items will not do any work.

Better ND-range partitioning is achieved if the number of loop iterations in the offloaded loop is known on the host. This allows the compiler and runtime to do an ND-range partitioning that matches the number of loop iterations.

To get this better partitioning, we use `firstprivate(upper)` instead of `map(to:upper)` on the target construct, as shown in the modified example below. This way, the compiler knows that the value of the variable `upper` on the host is the same as the value of the variable `upper` on the device.

**Listing 176:** /examples/OpenMP/07\_loop\_bounds/test\_loop\_bounds\_fp.cpp

```

49 #pragma omp target teams distribute parallel for private(b, i, j, k, l) \
50     firstprivate(upper)
51     for (int n = 0; n < upper; n++) {
52         double ur = 0.;
53         double us = 0.;
54         double ut = 0.;
55
56         k = n - (n / P) * P;
57         j = (n - k) / P;
58         i = (n - (j * P + k)) / (P * P);
59         b = n / (P * P * P);
60
61         for (l = 0; l < P; l++) {
62             ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
63             us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
64             ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
65         }
66
67         w[IDX4(b, i, j, k)] = ur * us * ut;
68     }

```

For the modified example (where `upper` is `firstprivate`), `LIBOMPTARGET_DEBUG=1` shows the following ND-range partitioning.

**Listing 177:** /examples/OpenMP/07\_loop\_bounds/test\_loop\_bounds\_fp.debug

```

Libomptarget --> Launching target execution __omp_offloading_802_b85fe0__Z4main_l49 with
↳ pointer 0x00000000016d97d8 (index=1).
Libomptarget --> Manifesting used target pointers:
Target LEVEL0 RTL --> Executing a kernel 0x00000000016d97d8...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 16
Target LEVEL0 RTL --> Preferred group size is multiple of 32
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {32, 1, 1}
Target LEVEL0 RTL --> Group counts = {1024, 1, 1}

```

Note that in the above partitioning, the total number of work-items =  $1024 \times 32 = 32,767$ , which exactly matches the actual number of loop iterations.

Using `firstprivate(upper)`, instead of `map(to:upper)`, reduced the runtime on the particular ATS GPU used (1-tile only):

<code>map(to:upper) version : 0.002432 seconds</code>
<code>firstprivate(upper) version : 0.001442 seconds</code>

## Allocating Memory on the Device

As mentioned above, the `map` clause determines how an original host variable is mapped to a corresponding variable on the device. However, the `map(to: )` clause may not be the most efficient way to allocate memory for a variable on the device.

In the following example, the variables `ur`, `us`, and `ut` are used as work (temporary) arrays in the computations on the device. The arrays are mapped to the device using `map(to: )` clauses (lines 46-48).

**Listing 178:** /examples/OpenMP/11\_device\_alloc/test\_map\_to.cpp

```
1 // clang-format off
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <time.h>
5
6 #include <math.h>
7 #include <omp.h>
8
9 #define P 16
10 #define BLOCKS 8
11 #define SIZE (BLOCKS * P * P * P)
12
13 #define MAX 100
14 #define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
15
16 #define IDX2(i, j) (i * P + j)
17 #define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)
18
19 int main(int argc, char *argv[]) {
20     double w[SIZE]; /* output */
21     double u[SIZE], dx[P * P]; /* input */
22     double ur[SIZE], us[SIZE], ut[SIZE]; /* work arrays */
23     int b, i, j, k, l; /* loop counters */
24     double start, end; /* timers */
25
26     omp_set_default_device(0);
27
28     /* dummy target region, so as not to measure startup time. */
29     #pragma omp target
30     { ; }
31
32     /* initialize input with random values */
33     srand(0);
34     for (int i = 0; i < SIZE; i++)
35         u[i] = scaled_rand();
36
37     for (int i = 0; i < P * P; i++)
38         dx[i] = scaled_rand();
39
40     start = omp_get_wtime();
```

(continues on next page)

(continued from previous page)

```

41  /* offload the kernel */
42  #pragma omp target teams distribute parallel for simd simdlen(16) collapse(4) \
43      map(to:u[0:SIZE],dx[0:P*P]) \
44      map(from:w[0:SIZE]) \
45      map(to:ur[0:SIZE]) \
46      map(to:us[0:SIZE]) \
47      map(to:ut[0:SIZE]) \
48      private(b,i,j,k,l)
49
50  for (b = 0; b < BLOCKS; b++) {
51      for (i = 0; i < P; i++) {
52          for (j = 0; j < P; j++) {
53              for (k = 0; k < P; k++) {
54                  w[IDX4(b, i, j, k)] = 0.;
55                  ur[IDX4(b, i, j, k)] = 0.;
56                  us[IDX4(b, i, j, k)] = 0.;
57                  ut[IDX4(b, i, j, k)] = 0.;

58                  for (l = 0; l < P; l++) {
59                      ur[IDX4(b, i, j, k)] += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
60                      us[IDX4(b, i, j, k)] += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
61                      ut[IDX4(b, i, j, k)] += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
62                  }
63
64                  w[IDX4(b, i, j, k)] = ur[IDX4(b, i, j, k)] * us[IDX4(b, i, j, k)] *
65                                  ut[IDX4(b, i, j, k)];
66              }
67          }
68      }
69  }
70
71  end = omp_get_wtime();
72
73  /* print result */
74  printf("collapse-clause: w[0]=%lf time=%lf\n", w[0], end - start);
75
76  return 0;
77}

```

The amount of data transferred between the host and the device can be seen in LIBOMPTARGET\_DEBUG=1 output by grepping for "Libomptarget --> Moving". The output shows that the map(to: ) clauses for the arrays ur, us, and ut cause the transfer of 262,144 bytes from the host to the device for each of the arrays:

```
$ grep "Libomptarget --> Moving" test_map_to.debug
Libomptarget --> Moving 262144 bytes (hst:0x00007ffefbe20e50) -> (tgt:0xfffffd556aa6b0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffefbe60e50) -> (tgt:0xfffffd556aa6f0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffefbea0e50) -> (tgt:0xfffffd556aa730000)
Libomptarget --> Moving 2048 bytes (hst:0x00007ffefbf60e50) -> (tgt:0xfffffd556aa660000)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffefbee0e50) -> (tgt:0xfffffd556aa560000)
Libomptarget --> Moving 262144 bytes (tgt:0xfffffd556aa670000) -> (hst:0x00007ffefbf20e50)
```

These data transfers are wasteful since the arrays ur, us, and ut are simply used as temporary work arrays on

the device. A better approach would be to place the declarations of the arrays between `declare target` and `end declare target` directives. This indicates that the arrays are mapped to the device data environment, but no data transfers for these arrays occur unless the `target update` directive is used to manage the consistency of the arrays between the host and the device. This approach is illustrated in the following modified example.

**Listing 179:** /examples/OpenMP/11\_device\_alloc/test\_declare\_target.cpp

```
1 // clang-format off
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <time.h>
5
6 #include <math.h>
7 #include <omp.h>
8
9 #define P 16
10 #define BLOCKS 8
11 #define SIZE (BLOCKS * P * P * P)
12
13 #define MAX 100
14 #define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
15
16 #define IDX2(i, j) (i * P + j)
17 #define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)
18
19 #pragma omp declare target
20 double ur[SIZE], us[SIZE], ut[SIZE]; /* work arrays */
21 #pragma omp end declare target
22
23 int main(int argc, char *argv[]) {
24     double w[SIZE]; /* output */
25     double u[SIZE], dx[P * P]; /* input */
26     int b, i, j, k, l; /* loop counters */
27     double start, end; /* timers */
28
29     omp_set_default_device(0);
30
31     /* dummy target region, so as not to measure startup time. */
32     #pragma omp target
33     { ; }
34
35     /* initialize input with random values */
36     srand(0);
37     for (int i = 0; i < SIZE; i++)
38         u[i] = scaled_rand();
39
40     for (int i = 0; i < P * P; i++)
41         dx[i] = scaled_rand();
42
43     start = omp_get_wtime();
44 }
```

(continues on next page)

(continued from previous page)

```

45 /* offload the kernel */
46 #pragma omp target teams distribute parallel for simd simdlen(16) collapse(4) \
47   map(to:u[0:SIZE],dx[0:P*P]) \
48   map(from:w[0:SIZE]) \
49   private(b,i,j,k,l)
50 for (b = 0; b < BLOCKS; b++) {
51   for (i = 0; i < P; i++) {
52     for (j = 0; j < P; j++) {
53       for (k = 0; k < P; k++) {
54         w[IDX4(b, i, j, k)] = 0.;
55         ur[IDX4(b, i, j, k)] = 0.;
56         us[IDX4(b, i, j, k)] = 0.;
57         ut[IDX4(b, i, j, k)] = 0.;

58         for (l = 0; l < P; l++) {
59           ur[IDX4(b, i, j, k)] += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
60           us[IDX4(b, i, j, k)] += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
61           ut[IDX4(b, i, j, k)] += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
62         }
63       }
64     }
65   }
66 }
67 }
68 }
69 }
70 }
71 end = omp_get_wtime();
72
73 /* print result */
74 printf("collapse-clause: w[0]=%lf time=%lf\n", w[0], end - start);
75
76 return 0;
77 }
```

In the above modified example, memory is allocated for arrays `ur`, `us`, and `ut` on the device, but no data transfers for these arrays take place. This is seen by grepping for "Libomptarget --> Moving" in LIBOMPTARGET\_DEBUG=1 output. We no longer see the transfer of 262,144 bytes from the host to the device for each of the arrays:

```
$ grep "Libomptarget --> Moving" test_declare_target.debug
Libomptarget --> Moving 2048 bytes (hst:0x00007fff12eafe0) -> (tgt:0xfffffd556aa660000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fff12e2efe0) -> (tgt:0xfffffd556aa6b0000)
Libomptarget --> Moving 262144 bytes (tgt:0xfffffd556aa670000) -> (hst:0x00007fff12e6e
```

An alternative approach for allocating memory on the device, without transferring any data between the host and the device, uses the `map(alloc: )` clause, instead of the `map(to: )` clause, as shown below (lines 46-48).

**Listing 180:** /examples/OpenMP/11\_device\_alloc/test\_map\_alloc.cpp

```
1 // clang-format off
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <time.h>
5
6 #include <math.h>
7 #include <omp.h>
8
9 #define P 16
10 #define BLOCKS 8
11 #define SIZE (BLOCKS * P * P * P)
12
13 #define MAX 100
14 #define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
15
16 #define IDX2(i, j) (i * P + j)
17 #define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)
18
19 int main(int argc, char *argv[]) {
20     double w[SIZE]; /* output */
21     double u[SIZE], dx[P * P]; /* input */
22     double ur[SIZE], us[SIZE], ut[SIZE]; /* work arrays */
23     int b, i, j, k, l; /* loop counters */
24     double start, end; /* timers */
25
26     omp_set_default_device(0);
27
28     /* dummy target region, so as not to measure startup time. */
29     #pragma omp target
30     { ; }
31
32     /* initialize input with random values */
33     srand(0);
34     for (int i = 0; i < SIZE; i++)
35         u[i] = scaled_rand();
36
37     for (int i = 0; i < P * P; i++)
38         dx[i] = scaled_rand();
39
40     start = omp_get_wtime();
41
42     /* offload the kernel */
43     #pragma omp target teams distribute parallel for simd simdlen(16) collapse(4) \
44         map(to:u[0:SIZE],dx[0:P*P]) \
45         map(from:w[0:SIZE]) \
46         map(alloc:ur[0:SIZE]) \
47         map(alloc:us[0:SIZE]) \
48         map(alloc:ut[0:SIZE]) \
49         private(b,i,j,k,l)
```

(continues on next page)

(continued from previous page)

```

50   for (b = 0; b < BLOCKS; b++) {
51     for (i = 0; i < P; i++) {
52       for (j = 0; j < P; j++) {
53         for (k = 0; k < P; k++) {
54           w[IDX4(b, i, j, k)] = 0.;
55           ur[IDX4(b, i, j, k)] = 0.;
56           us[IDX4(b, i, j, k)] = 0.;
57           ut[IDX4(b, i, j, k)] = 0.;

58           for (l = 0; l < P; l++) {
59             ur[IDX4(b, i, j, k)] += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
60             us[IDX4(b, i, j, k)] += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
61             ut[IDX4(b, i, j, k)] += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
62           }
63         }
64       }
65       w[IDX4(b, i, j, k)] = ur[IDX4(b, i, j, k)] * us[IDX4(b, i, j, k)] *
66                               ut[IDX4(b, i, j, k)];
67     }
68   }
69 }
70 }

71 end = omp_get_wtime();

72 /* print result */
73 printf("collapse-clause: w[0]=%lf time=%lf\n", w[0], end - start);

74 return 0;
75 }
```

In the above example, the `map(alloc: )` clauses for arrays `ur`, `us`, and `ut` cause memory to be allocated for `ur`, `us`, and `ut` on the device, and no data transfers occur – as in the `declare target` and `end declare target` case:

```
$ grep "Libomptarget --> Moving" test_map_alloc.debug
Libomptarget --> Moving 2048 bytes (hst:0x00007ffc6492ba80) -> (tgt:0xfffffd556aa660000)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffc648aba80) -> (tgt:0xfffffd556aa560000)
Libomptarget --> Moving 262144 bytes (tgt:0xfffffd556aa670000) -> (hst:0x00007ffc648eba8
```

The performance of the various versions when running on the particular ATS GPU used (1-tile only) was as follows:

<code>map(to: ) version</code>	: 0.002039 seconds
<code>declare target / end declare target version</code>	: 0.001272 seconds
<code>map(alloc: ) version</code>	: 0.001390 seconds

## Making Better Use of OpenMP Constructs

### Reduce Synchronizations by Using `nowait` If Appropriate

If appropriate, use the `nowait` clause on the `target` construct to reduce synchronizations.

By default, there is an implicit barrier at the end of a `target` region, which ensures that the host thread that encountered the `target` construct cannot continue until the `target` region is complete.

Adding the `nowait` clause on the `target` construct eliminates this implicit barrier, so the host thread that encountered the `target` construct can continue even if the `target` region is not complete. This allows the `target` region to execute asynchronously on the device without requiring the host thread to idly wait for the `target` region to complete.

Consider the following example which computes the product of two vectors, `v1` and `v2`, in a `parallel` region (line 43). Half of the computations is performed on the host by the team of threads executing the `parallel` region. The other half of the computations is performed on the device. The master thread of the team launches a `target` region to do the computations on the device.

By default, the master thread of the team has to wait for the `target` region to complete, before proceeding and participating in the computations (worksharing for loop) on the host.

**Listing 181:** /examples/OpenMP/04\_target\_nowait/test\_target\_no\_nowait.cpp

```
1 // clang-format off
2 /*
3  * This test is taken from OpenMP API 5.0.1 Examples (June 2020)
4  * https://www.openmp.org/wp-content/uploads/openmp-examples-5-0-1.pdf
5  * (4.13.2 nowait Clause on target Construct)
6 */
7
8
9 #include <stdio.h>
10 #include <stdlib.h>
11 #include <time.h>
12 #include <omp.h>
13
14 #define N 100000000 // N must be even
15
16 void init(int n, float *v1, float *v2) {
17     int i;
18
19     for(i=0; i<n; i++){
20         v1[i] = i * 0.25;
21         v2[i] = i - 1.25;
22     }
23 }
24
25 int main() {
26     int i, n=N;
27     float v1[N], v2[N], vxv[N];
```

(continues on next page)

(continued from previous page)

```

28 double start,end; // timers
29
30 init(n, v1, v2);
31
32 /* Dummy parallel and target regions, so as not to measure startup
33   time. */
34 #pragma omp parallel
35 {
36   #pragma omp master
37   #pragma omp target
38   {};
39 }
40
41 start=omp_get_wtime();
42
43 #pragma omp parallel
44 {
45   #pragma omp master
46   #pragma omp target teams distribute parallel for      \
47     map(to: v1[0:n/2])                                \
48     map(to: v2[0:n/2])                                \
49     map(from: vxv[0:n/2])
50   for(i=0; i<n/2; i++){
51     vxv[i] = v1[i]*v2[i];
52   }
53   /* Master thread will wait for target region to be completed
54     before proceeding beyond this point. */
55
56 #pragma omp for
57   for(i=n/2; i<n; i++) {
58     vxv[i] = v1[i]*v2[i];
59   }
60   /* Implicit barrier at end of worksharing for. */
61 }
62
63 end=omp_get_wtime();
64
65 printf("vxv[0]=%f, vxv[n-1]=%f, time=%lf\n", vxv[0], vxv[n-1], end-start);
66 return 0;
67 }
```

Performance would be improved if a nowait clause is specified on the target construct, so the master thread does not have to wait for the target region to complete, and can proceed to work on the worksharing for loop. The target region is guaranteed to complete by the synchronization in the implicit barrier at the end of the worksharing for loop.

**Listing 182:** /examples/OpenMP/04\_target\_nowait/test\_target\_nowait.cpp

```

1 // clang-format off
2 /*
```

(continues on next page)

(continued from previous page)

```
3 * This test is taken from OpenMP API 5.0.1 Examples (June 2020)
4 * https://www.openmp.org/wp-content/uploads/openmp-examples-5-0-1.pdf
5 * (4.13.2 nowait Clause on target Construct)
6 */
7
8
9 #include <stdio.h>
10 #include <stdlib.h>
11 #include <time.h>
12 #include <omp.h>
13
14 #define N 100000000 // N must be even
15
16 void init(int n, float *v1, float *v2) {
17     int i;
18
19     for(i=0; i<n; i++){
20         v1[i] = i * 0.25;
21         v2[i] = i - 1.25;
22     }
23 }
24
25 int main() {
26     int i, n=N;
27     float v1[N],v2[N],vxv[N];
28     double start,end; // timers
29
30     init(n, v1,v2);
31
32     /* Dummy parallel and target (nowait) regions, so as not to measure
33      startup time. */
34     #pragma omp parallel
35     {
36         #pragma omp master
37         #pragma omp target nowait
38         {};
39     }
40
41     start=omp_get_wtime();
42
43     #pragma omp parallel
44     {
45         #pragma omp master
46         #pragma omp target teams distribute parallel for nowait \
47             map(to: v1[0:n/2])                                \
48             map(to: v2[0:n/2])                                \
49             map(from: vxv[0:n/2])
50         for(i=0; i<n/2; i++){
51             vxv[i] = v1[i]*v2[i];
52         }
53     }
```

(continues on next page)

(continued from previous page)

```

54 #pragma omp for
55     for(i=n/2; i<n; i++) {
56         vxv[i] = v1[i]*v2[i];
57     }
58     /* Implicit barrier at end of worksharing for. Target region is
59     guaranteed to be completed by this point. */
60 }
61
62 end=omp_get_wtime();
63
64 printf("vxv[1]=%f, vxv[n-1]=%f, time=%lf\n", vxv[1], vxv[n-1], end-start);
65 return 0;
66 }
```

The performance of the two versions when running on the particular ATS GPU used (1-tile only) was as follows:

no nowait version : 0.150557 seconds
nowait on target version : 0.116472 seconds

## Fortran

The same nowait example shown above may be written in Fortran as shown below.

**Listing 183:** /examples/OpenMP/04\_target\_nowait/test\_target\_nowait\_ff90

```

1 !
2 ! This test is taken from OpenMP API 5.0.1 Examples (June 2020)
3 ! https://www.openmp.org/wp-content/uploads/openmp-examples-5-0-1.pdf
4 !(4.13.2 nowait Clause on target Construct)
5 !
6
7
8 subroutine init(n, v1, v2)
9     integer :: i, n
10    real :: v1(n), v2(n), vxv(n)
11
12    do i = 1, n
13        v1(i) = i * 0.25
14        v2(i) = i - 1.25
15    end do
16 end subroutine init
17
18 program test_target_nowait
19     use omp_lib
20     use iso_fortran_env
21     implicit none
22
23     integer, parameter :: NUM=100000000 ! NUM must be even
```

(continues on next page)

(continued from previous page)

```
24 real :: v1(NUM), v2(NUM), vxv(NUM)
25 integer :: n, i
26 real(kind=REAL64) :: start, end
27
28 n = NUM
29 call init(n, v1, v2)
30
31 ! Dummy parallel and target (nowait) regions, so as not to measure
32 ! startup time.
33 !$omp parallel
34     !$omp master
35         !$omp target nowait
36             !$omp end target
37         !$omp end master
38     !$omp end parallel
39
40 start=omp_get_wtime()
41
42 !$omp parallel
43
44     !$omp master
45         !$omp target teams distribute parallel do nowait &
46             !$omp& map(to: v1(1:n/2)) &
47                 !$omp& map(to: v2(1:n/2)) &
48                     !$omp& map(from: vxv(1:n/2))
49         do i = 1, n/2
50             vxv(i) = v1(i)*v2(i)
51         end do
52     !$omp end master
53
54     !$omp do
55     do i = n/2+1, n
56         vxv(i) = v1(i)*v2(i)
57     end do
58
59 !$omp end parallel
60
61 end=omp_get_wtime()
62
63 print *, "vxv(1)=", vxv(1), "vxv(n-1)=", vxv(n-1), "time=", end-start
64
65 end program test_target_nowait
```

## Memory Allocation

Memory allocations are represented as pointers in the application. A pointer on the host has the same size as a pointer on the device. In this section we look at various ways of allocating memory, and the types of allocations that are supported.

**Host allocations** are owned by the host and are intended to be allocated out of system memory. Host allocations are accessible by the host and all supported devices. So the same pointer to a host allocation may be used on the host and all supported devices. Host allocations are not expected to migrate between system memory and device-local memory. When a pointer to a host allocation is accessed on a device, data is typically sent over a bus, such as PCI-Express, that connects the device to the host.

**Device allocations** are owned by a specific device and are intended to be allocated out of device-local memory. Storage allocated can be read from and written to on that device, but is not directly accessible from the host or any other supported devices.

**Shared allocations** are accessible by the host and all supported devices. So the same pointer to a shared allocation may be used on the host and all supported devices, like in a host allocation. Shared allocations, however, are not owned by any particular device, but are intended to migrate between the host and one or more devices. This means that accesses on a device, after the migration has occurred, happen from much faster device-local memory instead of remotely accessing system memory through the higher-latency bus connection.

**Shared-system allocations** are a sub-class of shared allocations, where the memory is allocated by a system allocator (such as `malloc` or `new`) rather than by an allocation API (such as the OpenMP memory allocation API). Shared-system allocations have no associated device; they are inherently cross-device. Like other shared allocations, Shared-system allocations are intended to migrate between the host and supported devices, and the same pointer to a shared-system allocation may be used on the host and all supported devices.

Note:

- Currently, shared-system allocations are not supported on ATS and PVC systems. However, shared allocations where memory is allocated by an allocation API are supported on ATS and PVC.

The following table summarizes the characteristics of the various types of memory allocation.

Type of Allocation	Initial Location	Accessible on host?	Accessible on device?
Host	Host	Yes	Yes
Device	Device	No	Yes
Shared	Host, Device, or Unspecified	Yes	Yes
Shared-System	Host	Yes	Yes

Host allocations offer wide accessibility (can be accessed directly from the host and all supported devices), but have potentially high per-access costs since data is typically sent over a bus such as PCI-Express.

Shared allocations also offer wide accessibility, but the per-access costs are potentially lower than host allocations since data is migrated to the accessing device.

Device allocations have access limitations (cannot be accessed directly from the host or other supported devices), but offer higher performance since accesses are to device-local memory.

## OpenMP Runtime Routines for Memory Allocation

Intel compilers support a number of OpenMP runtime routines for doing memory allocations. These routines are shown in the table below.

OpenMP Memory Allocation Routine	Intel Extension?	Type of Allocation
omp_target_alloc	No	Device
omp_target_alloc_device	Yes	Device
omp_target_alloc_host	Yes	Host
omp_target_alloc_shared	Yes	Shared

Note that the three routines, `omp_target_alloc_device`, `omp_target_alloc_host`, and `omp_target_alloc_shared` are Intel extensions to the OpenMP specification.

We present below examples that use the above OpenMP memory allocation routines, and compare those to using `map` clauses.

For more information about memory allocation, see:

- [Data Parallel C++, by James Reinders et al](#)
- [SYCL 2020 Specification](#)
- [oneAPI Level Zero Specification](#)
- The the DPC++ part of this Guide

## Using the `map` Clause

We start off with an example that uses `map` clauses to allocate memory on a device and copy data between the host and the device.

In the following example, arrays A, B, and C are allocated in system memory by calling the C/C++ standard library routine, `malloc`.

The target construct on line 53 is the main kernel that computes the values of array C on the device. The `map(tofrom: C[0:length])` clause is specified on this target construct since the values of C need to be transferred from the host to the device before the computation, and from the device to the host at the end of the computation. The `map(to: A[0:length], B[0:length])` is specified for arrays "A" and B since the values of these arrays need to be transferred from the host to the device, and the device only reads these values. Under the covers, the `map` clauses cause storage for the arrays to be allocated on the device and data to be copied from the host to the device, and vice versa.

**Listing 184:** /examples/OpenMP/21\_omp\_target\_alloc/test\_target\_map.cpp

```
1 // clang-format off
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <stdint.h>
5 #include <math.h>
6 #include <omp.h>
7
8 #define iterations 1000
9 #define length      64*1024*1024
10
11 int main(int argc, char * argv[])
12 {
13     size_t bytes = length*sizeof(double);
14     double * __restrict A;
15     double * __restrict B;
16     double * __restrict C;
17     double scalar = 3.0;
18     double nstream_time = 0.0;
19
20     // Allocate arrays on the host using plain malloc()
21
22     A = (double *) malloc(bytes);
23     if (A == NULL){
24         printf(" ERROR: Cannot allocate space for A using plain malloc().\n");
25         exit(1);
26     }
27
28     B = (double *) malloc(bytes);
29     if (B == NULL){
30         printf(" ERROR: Cannot allocate space for B using plain malloc().\n");
31         exit(1);
32     }
33
34     C = (double *) malloc(bytes);
35     if (C == NULL){
36         printf(" ERROR: Cannot allocate space for C using plain malloc().\n");
37         exit(1);
38     }
39
40     // Initialize the arrays
41
42     #pragma omp parallel for
43     for (size_t i=0; i<length; i++) {
44         A[i] = 2.0;
45         B[i] = 2.0;
46         C[i] = 0.0;
47     }
48
49     // Perform the computation
```

(continues on next page)

(continued from previous page)

```
51 nstream_time = omp_get_wtime();
52 for (int iter = 0; iter<iterations; iter++) {
53     #pragma omp target teams distribute parallel for \
54         map(to: A[0:length], B[0:length]) \
55         map(tofrom: C[0:length])
56     for (size_t i=0; i<length; i++) {
57         C[i] += A[i] + scalar * B[i];
58     }
59 }
60 nstream_time = omp_get_wtime() - nstream_time;
61
62 // Validate and output results
63
64 double ar = 2.0;
65 double br = 2.0;
66 double cr = 0.0;
67 for (int iter = 0; iter<iterations; iter++) {
68     for (int i=0; i<length; i++) {
69         cr += ar + scalar * br;
70     }
71 }
72
73 double asum = 0.0;
74 #pragma omp parallel for reduction(+:asum)
75 for (size_t i=0; i<length; i++) {
76     asum += fabs(C[i]);
77 }
78
79 free(A);
80 free(B);
81 free(C);
82
83 double epsilon=1.e-8;
84 if (fabs(cr - asum)/asum > epsilon) {
85     printf("Failed Validation on output array\n"
86             "    Expected checksum: %lf\n"
87             "    Observed checksum: %lf\n"
88             "ERROR: solution did not validate\n", cr, asum);
89     return 1;
90 } else {
91     printf("Solution validates\n");
92     double avgtime = nstream_time/iterations;
93     printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
94 }
95
96 return 0;
97 }
```

The `map` clauses on the `target` construct inside the `iterations` loop cause data (values of `A`, `B`, `C`) to be transferred from the host to the device at the beginning of each `target` region, and cause data (values of `C`) to be transferred from the device to the host at the end of each `target` region. These data transfers incur a significant performance overhead. A better approach using `map` clauses would be to put the whole `iterations` loop inside

a target data construct with the map clauses. This causes the transfers to occur once at the beginning of the iterations loop, and another time at the end of the iterations loop. The modified example using target data and map clauses is shown below.

**Listing 185:** /examples/OpenMP/21\_omp\_target\_alloc/test\_target\_map2.cpp

```

1 // clang-format off
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <stdint.h>
5 #include <math.h>
6 #include <omp.h>
7
8 #define iterations 1000
9 #define length      64*1024*1024
10
11 int main(int argc, char * argv[])
12 {
13     size_t bytes = length*sizeof(double);
14     double * __restrict A;
15     double * __restrict B;
16     double * __restrict C;
17     double scalar = 3.0;
18     double nstream_time = 0.0;
19
20     // Allocate arrays on the host using plain malloc()
21
22     A = (double *) malloc(bytes);
23     if (A == NULL){
24         printf(" ERROR: Cannot allocate space for A using plain malloc().\n");
25         exit(1);
26     }
27
28     B = (double *) malloc(bytes);
29     if (B == NULL){
30         printf(" ERROR: Cannot allocate space for B using plain malloc().\n");
31         exit(1);
32     }
33
34     C = (double *) malloc(bytes);
35     if (C == NULL){
36         printf(" ERROR: Cannot allocate space for C using plain malloc().\n");
37         exit(1);
38     }
39
40     // Initialize the arrays
41
42     #pragma omp parallel for
43     for (size_t i=0; i<length; i++) {
44         A[i] = 2.0;
45         B[i] = 2.0;
46         C[i] = 0.0;

```

(continues on next page)

(continued from previous page)

```
47 }
48
49 // Perform the computation
50
51 nstream_time = omp_get_wtime();
52 #pragma omp target data map(to: A[0:length], B[0:length]) \
53   map(tofrom: C[0:length])
54 {
55   for (int iter = 0; iter<iterations; iter++) {
56     #pragma omp target teams distribute parallel for
57     for (size_t i=0; i<length; i++) {
58       C[i] += A[i] + scalar * B[i];
59     }
60   }
61 }
62 nstream_time = omp_get_wtime() - nstream_time;
63
64 // Validate and output results
65
66 double ar = 2.0;
67 double br = 2.0;
68 double cr = 0.0;
69 for (int iter = 0; iter<iterations; iter++) {
70   for (int i=0; i<length; i++) {
71     cr += ar + scalar * br;
72   }
73 }
74
75 double asum = 0.0;
76 #pragma omp parallel for reduction(+:asum)
77 for (size_t i=0; i<length; i++) {
78   asum += fabs(C[i]);
79 }
80
81 free(A);
82 free(B);
83 free(C);
84
85 double epsilon=1.e-8;
86 if (fabs(cr - asum)/asum > epsilon) {
87   printf("Failed Validation on output array\n"
88         "      Expected checksum: %lf\n"
89         "      Observed checksum: %lf\n"
90         "ERROR: solution did not validate\n", cr, asum);
91   return 1;
92 } else {
93   printf("Solution validates\n");
94   double avgtime = nstream_time/iterations;
95   printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
96 }
```

(continues on next page)

(continued from previous page)

```
98     return 0;
99 }
```

## omp\_target\_alloc

We modify the above example to use device allocations instead of map clauses. Storage for arrays A, B, and C is directly allocated on the device by calling the OpenMP runtime routine `omp_target_alloc`. The routine takes two arguments: the number of bytes to allocate on the device, and the number of the device on which to allocate the storage. The routine returns a device pointer that references the device address of the storage allocated on the device. If the call to `omp_target_alloc` returns NULL, then this indicates that the allocation was not successful.

To access the allocated memory in a target construct, the device pointer returned by a call to `omp_target_alloc` is listed in an `is_device_ptr` clause on the target construct. This ensures that there is no data transfer before and after kernel execution since the kernel operates on data that is already on the device.

At the end of the program, the runtime routine `omp_target_free` is used to deallocate the storage for A, B, and C on the device.

**Listing 186:** /examples/OpenMP/21\_omp\_target\_alloc/test\_omp\_target\_alloc.cpp

```
1 // clang-format off
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <stdint.h>
5 #include <math.h>
6 #include <omp.h>
7
8 #define iterations 1000
9 #define length      64*1024*1024
10
11 int main(int argc, char * argv[])
12 {
13     int device_id = omp_get_default_device();
14     size_t bytes = length*sizeof(double);
15     double * __restrict A;
16     double * __restrict B;
17     double * __restrict C;
18     double scalar = 3.0;
19     double nstream_time = 0.0;
20
21     // Allocate arrays in device memory
22
23     A = (double *) omp_target_alloc(bytes, device_id);
24     if (A == NULL){
25         printf(" ERROR: Cannot allocate space for A using omp_target_alloc().\n");
26         exit(1);
27     }
```

(continues on next page)

(continued from previous page)

```
28
29     B = (double *) omp_target_alloc(bytes, device_id);
30     if (B == NULL){
31         printf(" ERROR: Cannot allocate space for B using omp_target_alloc().\n");
32         exit(1);
33     }
34
35     C = (double *) omp_target_alloc(bytes, device_id);
36     if (C == NULL){
37         printf(" ERROR: Cannot allocate space for C using omp_target_alloc().\n");
38         exit(1);
39     }
40
41 // Initialize the arrays
42
43 #pragma omp target teams distribute parallel for \
44     is_device_ptr(A,B,C)
45 for (size_t i=0; i<length; i++) {
46     A[i] = 2.0;
47     B[i] = 2.0;
48     C[i] = 0.0;
49 }
50
51 // Perform the computation
52
53 nstream_time = omp_get_wtime();
54 for (int iter = 0; iter<iterations; iter++) {
55     #pragma omp target teams distribute parallel for \
56         is_device_ptr(A,B,C)
57     for (size_t i=0; i<length; i++) {
58         C[i] += A[i] + scalar * B[i];
59     }
60 }
61 nstream_time = omp_get_wtime() - nstream_time;
62
63 // Validate and output results
64
65 double ar = 2.0;
66 double br = 2.0;
67 double cr = 0.0;
68 for (int iter = 0; iter<iterations; iter++) {
69     for (int i=0; i<length; i++) {
70         cr += ar + scalar * br;
71     }
72 }
73
74 double asum = 0.0;
75 #pragma omp target teams distribute parallel for reduction(+:asum) \
76     is_device_ptr(C)
77 for (size_t i=0; i<length; i++) {
78     asum += fabs(C[i]);
```

(continues on next page)

(continued from previous page)

```

79 }
80
81     omp_target_free(A, device_id);
82     omp_target_free(B, device_id);
83     omp_target_free(C, device_id);
84
85     double epsilon=1.e-8;
86     if (fabs(cr - asum)/asum > epsilon) {
87         printf("Failed Validation on output array\n"
88             "    Expected checksum: %lf\n"
89             "    Observed checksum: %lf\n"
90             "ERROR: solution did not validate\n", cr, asum);
91         return 1;
92     } else {
93         printf("Solution validates\n");
94         double avgtime = nstream_time/iterations;
95         printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
96     }
97
98     return 0;
99 }
```

**Notes:**

- When calling `omp_target_alloc`, the device number specified must be one of the supported devices, other than the host device. This will be the device on which storage will be allocated.
- Since the arrays A, B, and C are not accessible from the host, the initialization of the arrays, kernel execution, and summation of elements of C all need to be done inside OpenMP target regions.
- A device allocation can only be accessed by the device specified in the `omp_target_alloc` call, but may be copied to memory allocated on the host or other devices by calling `omp_target_memcpy`.

## **omp\_target\_alloc\_device**

The Intel® extension, `omp_target_alloc_device`, is similar to `omp_target_alloc`. It is also called with two arguments: the number of bytes to allocate on the device, and the number of the device on which to allocate the storage. The routine returns a device pointer that references the device address of the storage allocated on the device. If the call to `omp_target_alloc_device` returns NULL, then this indicates that the allocation was not successful.

The above `omp_target_alloc` example can be rewritten using `omp_target_alloc_device`, by simply replacing the call to `omp_target_alloc` by a call to `omp_target_alloc_device` as shown below.

At the end of the program, the runtime routine `omp_target_free` is used to deallocate the storage for A, B, and C on the device.

**Listing 187:** /examples/OpenMP/21\_omp\_target\_alloc/test\_omp\_target\_alloc\_device.cpp

```

21 // Allocate arrays in device memory
22
23 A = (double *) omp_target_alloc_device(bytes, device_id);
24 if (A == NULL){
25     printf(" ERROR: Cannot allocate space for A using omp_target_alloc_device().\n");
26     exit(1);
27 }
28
29 B = (double *) omp_target_alloc_device(bytes, device_id);
30 if (B == NULL){
31     printf(" ERROR: Cannot allocate space for B using omp_target_alloc_device().\n");
32     exit(1);
33 }
34
35 C = (double *) omp_target_alloc_device(bytes, device_id);
36 if (C == NULL){
37     printf(" ERROR: Cannot allocate space for C using omp_target_alloc_device().\n");
38     exit(1);
39 }
```

Note:

- All of the above Notes that apply to `omp_target_alloc` also apply to `omp_target_alloc_device`.

### **omp\_target\_alloc\_host**

The above example can also be rewritten by doing a host allocation for A, B, and C. This allows the memory to be accessible to the host and all supported devices.

In the following modified example, the `omp_target_alloc_host` runtime routine (an Intel® extension) is called to allocate storage for each of arrays A, B, and C. The routine takes two arguments: the number of bytes to allocate, and a device number. The device number must be one of the supported devices, other than the host device. The routine returns a pointer to a storage location in host memory. If the call to `omp_target_alloc_host` returns `NULL`, then this indicates that the allocation was not successful.

Note the directive `requires unified_address` is specified at the top of the program. This requires that the implementation guarantee that all devices accessible through OpenMP API routines and directives use a unified address space. In this address space, a pointer will always refer to the same location in memory from all devices, and the `is_device_ptr` clause is not necessary to obtain device addresses from device pointers for use inside target regions. When using Intel compilers, the `requires unified_address` directive is actually not needed, since unified address space is guaranteed by default. However, we include the directive in the code for portability.

The pointer returned by a call to `omp_target_alloc_host` can be used to access the storage from the host and all supported devices. No `map` clauses and no `is_device_ptr` clauses are needed on a `target` construct to access the memory from a device since a unified address space is used.

At the end of the program, the runtime routine `omp_target_free` is used to deallocate the storage for A, B, and C.

**Listing 188:** /examples/OpenMP/21\_omp\_target\_alloc/test\_omp\_target\_alloc\_host.cpp

```
1 // clang-format off
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <stdint.h>
5 #include <math.h>
6 #include <omp.h>
7
8 #pragma omp requires unified_address
9
10#define iterations 1000
11#define length      64*1024*1024
12
13int main(int argc, char * argv[])
14{
15    int device_id = omp_get_default_device();
16    size_t bytes = length*sizeof(double);
17    double * __restrict A;
18    double * __restrict B;
19    double * __restrict C;
20    double scalar = 3.0;
21    double nstream_time = 0.0;
22
23    // Allocate arrays in host memory
24
25    A = (double *) omp_target_alloc_host(bytes, device_id);
26    if (A == NULL){
27        printf(" ERROR: Cannot allocate space for A using omp_target_alloc_host().\n");
28        exit(1);
29    }
30
31    B = (double *) omp_target_alloc_host(bytes, device_id);
32    if (B == NULL){
33        printf(" ERROR: Cannot allocate space for B using omp_target_alloc_host().\n");
34        exit(1);
35    }
36
37    C = (double *) omp_target_alloc_host(bytes, device_id);
38    if (C == NULL){
39        printf(" ERROR: Cannot allocate space for C using omp_target_alloc_host().\n");
40        exit(1);
41    }
42
43    // Initialize the arrays
44
45    #pragma omp parallel for
46    for (size_t i=0; i<length; i++) {
47        A[i] = 2.0;
48        B[i] = 2.0;
49        C[i] = 0.0;
```

(continues on next page)

(continued from previous page)

```
50 }
51
52 // Perform the computation
53
54 nstream_time = omp_get_wtime();
55 for (int iter = 0; iter<iterations; iter++) {
56     #pragma omp target teams distribute parallel for
57     for (size_t i=0; i<length; i++) {
58         C[i] += A[i] + scalar * B[i];
59     }
60 }
61 nstream_time = omp_get_wtime() - nstream_time;
62
63 // Validate and output results
64
65 double ar = 2.0;
66 double br = 2.0;
67 double cr = 0.0;
68 for (int iter = 0; iter<iterations; iter++) {
69     for (int i=0; i<length; i++) {
70         cr += ar + scalar * br;
71     }
72 }
73
74 double asum = 0.0;
75 #pragma omp parallel for reduction(+:asum)
76 for (size_t i=0; i<length; i++) {
77     asum += fabs(C[i]);
78 }
79
80 omp_target_free(A, device_id);
81 omp_target_free(B, device_id);
82 omp_target_free(C, device_id);
83
84 double epsilon=1.e-8;
85 if (fabs(cr - asum)/asum > epsilon) {
86     printf("Failed Validation on output array\n"
87             "      Expected checksum: %lf\n"
88             "      Observed checksum: %lf\n"
89             "ERROR: solution did not validate\n", cr, asum);
90     return 1;
91 } else {
92     printf("Solution validates\n");
93     double avgtime = nstream_time/iterations;
94     printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
95 }
96
97 return 0;
98 }
```

## Notes:

- When calling `omp_target_alloc_host`, the device number specified must be one of the supported de-

vices, other than the host device.

- Since the arrays A, B, and C are accessible from the host and device, the initialization of the arrays and summation of elements of C may be done either on the host (outside of a target construct) or on the device (inside a target construct).

### **omp\_target\_alloc\_shared**

We modify the above example, so shared allocations are used instead of host allocations. The `omp_target_alloc_shared` runtime routine is called to allocate storage for each of arrays A, B, and C. The routine takes two arguments: the number of bytes to allocate on the device, and a device number. The device number must be one of the supported devices, other than the host device. The routine returns a pointer to a storage location in shared memory. If the call to `omp_target_alloc_shared` returns NULL, then this indicates that the allocation was not successful.

Note the requires `unified_address` directive is specified at the top of the program, for portability.

The pointer returned by a call to `omp_target_alloc_shared` can be used to access the storage from the host and all supported devices. No `map` clauses and no `is_device_ptr` clauses are needed on a target construct to access the memory from a device since a unified address space is used.

At the end of the program, the runtime routine `omp_target_free` is used to deallocate the storage for A, B, and C.

**Listing 189:** /examples/OpenMP/21\_omp\_target\_alloc/test\_omp\_target\_alloc\_shared.cpp

```

23 // Allocate arrays in shared memory
24
25 A = (double *) omp_target_alloc_shared(bytes, device_id);
26 if (A == NULL){
27     printf(" ERROR: Cannot allocate space for A using omp_target_alloc_shared().\n");
28     exit(1);
29 }
30
31 B = (double *) omp_target_alloc_shared(bytes, device_id);
32 if (B == NULL){
33     printf(" ERROR: Cannot allocate space for B using omp_target_alloc_shared().\n");
34     exit(1);
35 }
36
37 C = (double *) omp_target_alloc_shared(bytes, device_id);
38 if (C == NULL){
39     printf(" ERROR: Cannot allocate space for C using omp_target_alloc_shared().\n");
40     exit(1);
41 }
```

Notes:

- When calling `omp_target_alloc_shared`, the device number specified must be one of the supported devices, other than the host device.

- Since the arrays are accessible from the host and device, the initialization and verification may be done either on the host or on the device (inside a target construct).

## omp\_target\_memcpy

The following example shows how the runtime routine `omp_target_memcpy` may be used to copy memory from the host to the device, and vice versa. First arrays `h_A`, `h_B`, and `h_C` are allocated in system memory using plain `malloc`, and then initialized. Corresponding arrays `d_A`, `d_B`, and `d_C` are allocated on the device using `omp_target_alloc`.

Before the start of the target construct on line 99, the values in `h_A`, `h_B`, and `h_C` are copied to `d_A`, `d_B`, and `d_C` by calling `omp_target_memcpy`. After the target region, new `d_C` values computed on the device are copied to `h_C` by calling `omp_target_memcpy`.

**Listing 190:** /examples/OpenMP/21\_omp\_target\_alloc/test\_omp\_target\_memcpy.cpp

```
1 // clang-format off
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <stdint.h>
5 #include <math.h>
6 #include <omp.h>
7
8 #define iterations 1000
9 #define length      64*1024*1024
10
11 int main(int argc, char * argv[])
12 {
13     int device_id = omp_get_default_device();
14     int host_id = omp_get_initial_device();
15     size_t bytes = length*sizeof(double);
16     double * __restrict h_A;
17     double * __restrict h_B;
18     double * __restrict h_C;
19     double * __restrict d_A;
20     double * __restrict d_B;
21     double * __restrict d_C;
22     double scalar = 3.0;
23     double nstream_time = 0.0;
24
25     // Allocate arrays h_A, h_B, and h_C on the host using plain malloc()
26
27     h_A = (double *) malloc(bytes);
28     if (h_A == NULL){
29         printf(" ERROR: Cannot allocate space for h_A using plain malloc().\n");
30         exit(1);
31     }
32
33     h_B = (double *) malloc(bytes);
34     if (h_B == NULL){
```

(continues on next page)

(continued from previous page)

```
35     printf(" ERROR: Cannot allocate space for h_B using plain malloc().\n");
36     exit(1);
37 }
38
39 h_C = (double *) malloc(bytes);
40 if (h_C == NULL){
41     printf(" ERROR: Cannot allocate space for h_C using plain malloc().\n");
42     exit(1);
43 }
44
45 // Allocate arrays d_A, d_B, and d_C on the device using omp_target_alloc()
46
47 d_A = (double *) omp_target_alloc(bytes, device_id);
48 if (d_A == NULL){
49     printf(" ERROR: Cannot allocate space for d_A using omp_target_alloc().\n");
50     exit(1);
51 }
52
53 d_B = (double *) omp_target_alloc(bytes, device_id);
54 if (d_B == NULL){
55     printf(" ERROR: Cannot allocate space for d_B using omp_target_alloc().\n");
56     exit(1);
57 }
58
59 d_C = (double *) omp_target_alloc(bytes, device_id);
60 if (d_C == NULL){
61     printf(" ERROR: Cannot allocate space for d_C using omp_target_alloc().\n");
62     exit(1);
63 }
64
65 // Initialize the arrays on the host
66
67 #pragma omp parallel for
68 for (size_t i=0; i<length; i++) {
69     h_A[i] = 2.0;
70     h_B[i] = 2.0;
71     h_C[i] = 0.0;
72 }
73
74 // Call omp_target_memcpy() to copy values from host to device
75
76 int rc = 0;
77 rc = omp_target_memcpy(d_A, h_A, bytes, 0, 0, device_id, host_id);
78 if (rc) {
79     printf("ERROR: omp_target_memcpy(A) returned %d\n", rc);
80     exit(1);
81 }
82
83 rc = omp_target_memcpy(d_B, h_B, bytes, 0, 0, device_id, host_id);
84 if (rc) {
85     printf("ERROR: omp_target_memcpy(B) returned %d\n", rc);
```

(continues on next page)

(continued from previous page)

```
86     exit(1);
87 }
88
89 rc = omp_target_memcpy(d_C, h_C, bytes, 0, 0, device_id, host_id);
90 if (rc) {
91     printf("ERROR: omp_target_memcpy(C) returned %d\n", rc);
92     exit(1);
93 }
94
95 // Perform the computation
96
97 nstream_time = omp_get_wtime();
98 for (int iter = 0; iter<iterations; iter++) {
99     #pragma omp target teams distribute parallel for \
100         is_device_ptr(d_A,d_B,d_C)
101     for (size_t i=0; i<length; i++) {
102         d_C[i] += d_A[i] + scalar * d_B[i];
103     }
104 }
105 nstream_time = omp_get_wtime() - nstream_time;
106
107 // Call omp_target_memcpy() to copy values from device to host
108
109 rc = omp_target_memcpy(h_C, d_C, bytes, 0, 0, host_id, device_id);
110 if (rc) {
111     printf("ERROR: omp_target_memcpy(A) returned %d\n", rc);
112     exit(1);
113 }
114
115 // Validate and output results
116
117 double ar = 2.0;
118 double br = 2.0;
119 double cr = 0.0;
120 for (int iter = 0; iter<iterations; iter++) {
121     for (int i=0; i<length; i++) {
122         cr += ar + scalar * br;
123     }
124 }
125
126 double asum = 0.0;
127 #pragma omp parallel for reduction(+:asum)
128 for (size_t i=0; i<length; i++) {
129     asum += fabs(h_C[i]);
130 }
131
132 free(h_A);
133 free(h_B);
134 free(h_C);
135 omp_target_free(d_A, device_id);
136 omp_target_free(d_B, device_id);
```

(continues on next page)

(continued from previous page)

```

137     omp_target_free(d_C, device_id);

138
139     double epsilon=1.e-8;
140     if (fabs(cr - asum)/asum > epsilon) {
141         printf("Failed Validation on output array\n"
142             "    Expected checksum: %lf\n"
143             "    Observed checksum: %lf\n"
144             "ERROR: solution did not validate\n", cr, asum);
145         return 1;
146     } else {
147         printf("Solution validates\n");
148         double avgtime = nstream_time/iterations;
149         printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
150     }
151
152     return 0;
153 }
```

## Performance Considerations

In the above examples (using the map clause, `omp_target_alloc`, `omp_target_alloc_device`, `omp_target_alloc_host`, `omp_target_alloc_shared`, `omp_target_memcpy`), the main kernel is the target construct which computes the values of array C. To get more accurate timings, this target construct is enclosed in a loop, so the offload happens `iterations` number of times (where `iterations = 1000`). We compute average kernel time by deviding the total time taken by the `iterations` loop by 1000.

**Listing 191:** /examples/OpenMP/21\_omp\_target\_alloc/test\_omp\_target\_alloc.cpp

```

51 // Perform the computation
52
53 nstream_time = omp_get_wtime();
54 for (int iter = 0; iter<iterations; iter++) {
55     #pragma omp target teams distribute parallel for \
56         is_device_ptr(A,B,C)
57     for (size_t i=0; i<length; i++) {
58         C[i] += A[i] + scalar * B[i];
59     }
60 }
61 nstream_time = omp_get_wtime() - nstream_time;
```

LIBOMPTARGET\_DEBUG=1 output shows that all the above examples have the same ND\_range partitioning.

**Listing 192:** /examples/OpenMP/21\_omp\_target\_alloc/test\_omp\_target\_alloc.debug

```

Libomptarget --> Launching target execution __omp_offloading_811_6e21ac8__Z4main_l55 with
↳ pointer 0x0000000001fc20c8 (index=1).
Libomptarget --> Manifesting used target pointers:
```

(continues on next page)

(continued from previous page)

```
Target LEVEL0 RTL --> Executing a kernel 0x0000000001fc20c8...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 67108863, Stride = 1
Target LEVEL0 RTL --> Group sizes = {64, 1, 1}
Target LEVEL0 RTL --> Group counts = {1048576, 1, 1}
```

The average times taken by the kernel in the various versions when running on the particular ATS GPU used (1-tile only) are shown below.

Version	Time (seconds)
map	0.139208
map + target data	0.008491
omp_target_alloc	0.007247
omp_target_alloc_device	0.007484
omp_target_alloc_host	0.088674
omp_target_alloc_shared	0.008319
omp_target_memcpy	0.008284

The above performance numbers show that the `map` version is the slowest version (0.139208 seconds). This is because of the data transfers that occur at the beginning and end of each kernel launch. The main kernel is launched 1000 times. At the beginning of each kernel launch, storage for arrays A, B and C is allocated on the device, and the values of these arrays are copied from the host to the device. At the end of the kernel, the values of array C are copied from the device to the host. Putting the whole `iterations` loop inside a `target_data` construct with `map` clauses reduced the runtime to 0.008491 seconds, as the transfers occur once at the launch of the first kernel in the `iterations` loop, and another time at the end of the last kernel in the `iterations` loop.

The `omp_target_alloc` and `omp_target_alloc_device` versions have the best performance (0.007247 and 0.007484 seconds, respectively). In these versions, storage for A, B, and C is allocated directly in device memory. So accesses on the device happen from device-local memory.

The `omp_target_alloc_shared` version also performs well, but is somewhat slower (0.008319 seconds). In this version, storage for A, B, and C is allocated in shared memory. So the data can migrate between the host and devices. There is the overhead of migration but, after migration, accesses on a device happen from much faster device-local memory.

The `omp_target_alloc_host` version takes almost 11x more time than the `omp_target_alloc_shared` version (0.088674 seconds). This is because data allocated in host memory does not migrate from the host to the device. Rather, the data is typically sent over a bus, such as PCI-Express, that connects the device to the host. Generally, data should be allocated in host memory using `omp_target_alloc_host` if only a small amount of that data will be accessed on the device.

Finally, a note regarding data transfers: The amount of data transferred in the `map` version can be seen in LIBOMP-TARGET\_DEBUG=1 output by grepping for "Libomptarget --> Moving". We notice that for each launch of the main kernel, we have the following data transfers:

```
$ grep "Libomptarget --> Moving" test_target_map.debug
Libomptarget --> Moving 536870912 bytes (hst:0x00001477908f6010) -> (tgt:0xffffd556aaa00000)
Libomptarget --> Moving 536870912 bytes (hst:0x00001477d08f8010) -> (tgt:0xffffd556caa00000)
Libomptarget --> Moving 536870912 bytes (hst:0x00001477b08f7010) -> (tgt:0xffffd556eaa00000)
Libomptarget --> Moving 536870912 bytes (tgt:0xffffd556aaa00000) -> (hst:0x00001477908f6010)
```

On the other hand, data transfers in the `omp_target_alloc...` versions are handled by a lower layer of the run-time system. So grepping for "Libomptarget --> Moving" in `LIBOMPTARGET_DEBUG=1` output for these versions will not show the data transfers that took place.

## Fortran

The Fortran version of the example using `target_data` and `map` clauses is shown below.

**Listing 193:** /examples/OpenMP/21\_omp\_target\_alloc/test\_target\_map2\_f.f90

```

1 program main
2   use iso_fortran_env
3   use omp_lib
4   implicit none
5
6   integer, parameter :: iterations=1000
7   integer, parameter :: length=64*1024*1024
8   real(kind=REAL64), parameter :: epsilon=1.D-8
9   real(kind=REAL64), allocatable :: A(:)
10  real(kind=REAL64), allocatable :: B(:)
11  real(kind=REAL64), allocatable :: C(:)
12  real(kind=REAL64) :: scalar=3.0
13  real(kind=REAL64) :: ar, br, cr, asum
14  real(kind=REAL64) :: nstream_time, avgtime
15  integer :: err, i, iter
16
17 !
18 ! Allocate arrays on the host using plain allocate
19
20  allocate( A(length), stat=err )
21  if (err .ne. 0) then
22    print *, "Allocation of A returned ", err
23    stop 1
24  endif
25
26  allocate( B(length), stat=err )
27  if (err .ne. 0) then
28    print *, "Allocation of B returned ", err
29    stop 1
30  endif
31
32  allocate( C(length), stat=err )
33  if (err .ne. 0) then
```

(continues on next page)

(continued from previous page)

```
34     print *, "Allocation of C returned ", err
35     stop 1
36   endif
37
38 !
39 ! Initialize the arrays
40
41 !$omp parallel do
42 do i = 1, length
43     A(i) = 2.0
44     B(i) = 2.0
45     C(i) = 0.0
46 end do
47
48 !
49 ! Perform the computation
50
51 nstream_time = omp_get_wtime()
52 !$omp target data map(to: A, B) map(tofrom: C)
53
54 do iter = 1, iterations
55     !$omp target teams distribute parallel do
56     do i = 1, length
57         C(i) = C(i) + A(i) + scalar * B(i)
58     end do
59 end do
60
61 !$omp end target data
62 nstream_time = omp_get_wtime() - nstream_time
63
64 !
65 ! Validate and output results
66
67 ar = 2.0
68 br = 2.0
69 cr = 0.0
70 do iter = 1, iterations
71     do i = 1, length
72         cr = cr + ar + scalar * br
73     end do
74 end do
75
76 asum = 0.0
77 !$omp parallel do reduction(+:asum)
78 do i = 1, length
79     asum = asum + abs(C(i))
80 end do
81
82 if (abs(cr - asum)/asum > epsilon) then
83     print *, "Failed Validation on output array: ", "Expected =", cr, "Observed =", asum
84 else
```

(continues on next page)

(continued from previous page)

```

85     avgtime = nstream_time/iterations
86     print *, "Solution validates: ", "Checksum =", asum, "Avg time (s) =", avgtime
87   endif
88
89   deallocate(A)
90   deallocate(B)
91   deallocate(C)
92
93 end program main

```

The Fortran version of the example using `omp_target_alloc_device` is shown below. In this example, `allocate` directives, with the allocator `omp_target_device_mem_alloc`, are used to allocate arrays A, B, and C on the device. The `use_device_addr(A, B, C)` clause is used on the `target data` directive (line 32) to indicate that the arrays have device addresses, and these addresses should be used in the target region.

**Listing 194:** /examples/OpenMP/21\_omp\_target\_alloc/test\_omp\_target\_alloc\_device\_ff90

```

1  program main
2    use iso_fortran_env
3    use omp_lib
4    implicit none
5
6    integer, parameter :: iterations=1000
7    integer, parameter :: length=64*1024*1024
8    real(kind=REAL64), parameter :: epsilon=1.D-8
9    real(kind=REAL64), allocatable :: A(:)
10   real(kind=REAL64), allocatable :: B(:)
11   real(kind=REAL64), allocatable :: C(:)
12   real(kind=REAL64) :: scalar=3.0
13   real(kind=REAL64) :: ar, br, cr, asum
14   real(kind=REAL64) :: nstream_time, avgtime
15   integer :: i, iter
16
17   !
18   ! Allocate arrays in device memory
19
20   !$omp allocate allocator(omp_target_device_mem_alloc)
21   allocate(A(length))
22
23   !$omp allocate allocator(omp_target_device_mem_alloc)
24   allocate(B(length))
25
26   !$omp allocate allocator(omp_target_device_mem_alloc)
27   allocate(C(length))
28
29   !
30   ! Begin target data
31
32   !$omp target data use_device_addr(A, B, C)

```

(continues on next page)

(continued from previous page)

```
34 !
35 ! Initialize the arrays
36
37 !$omp target teams distribute parallel do
38 do i = 1, length
39     A(i) = 2.0
40     B(i) = 2.0
41     C(i) = 0.0
42 end do
43
44 !
45 ! Perform the computation
46
47 nstream_time = omp_get_wtime()
48 do iter = 1, iterations
49     !$omp target teams distribute parallel do
50     do i = 1, length
51         C(i) = C(i) + A(i) + scalar * B(i)
52     end do
53 end do
54 nstream_time = omp_get_wtime() - nstream_time
55
56 !
57 ! Validate and output results
58
59 ar = 2.0
60 br = 2.0
61 cr = 0.0
62 do iter = 1, iterations
63     do i = 1, length
64         cr = cr + ar + scalar * br
65     end do
66 end do
67
68 asum = 0.0
69 !$omp target teams distribute parallel do reduction(+:asum)
70 do i = 1, length
71     asum = asum + abs(C(i))
72 end do
73
74 !
75 ! End target data
76
77 !$omp end target data
78
79 if (abs(cr - asum)/asum > epsilon) then
80     print *, "Failed Validation on output array:", "Expected =", cr, "Observed =", asum
81 else
82     avgtime = nstream_time/iterations
83     print *, "Solution validates:", "Checksum =", asum, "Avg time (s) =", avgtime
84 endif
```

(continues on next page)

(continued from previous page)

```

85     deallocate(A)
86     deallocate(B)
87     deallocate(C)
88
89 end program main

```

## Calling MKL Routines from Offload Regions

The Intel® oneAPI Math Kernel Library (oneMKL) provides a number of math routines for software applications that solve large computational problems. You can use OpenMP offload to run the math routines on Intel GPUs.

If a oneMKL routine is called from within an OpenMP target variant dispatch region and offload is possible, then the GPU implementation of the routine is dispatched.

If the oneMKL routine is called outside of an OpenMP target variant dispatch region or if offload is disabled, then the CPU implementation is dispatched.

By default, the execution of a oneMKL routine called from within a target dispatch variant region is synchronous. The host thread encountering the target target dispatch variant construct has to wait for the region to complete execution. OpenMP offload computations may be done asynchronously by adding the nowait clause to the target variant dispatch construct. This ensures that the host thread will not be blocked, but can proceed with executing further code.

The following example illustrates the use of target variant dispatch to call the GPU implementation of the `dgemm` routine.

**Listing 195:** /examples/OpenMP/22\_mkl\_target\_variant\_dispatch/cblas\_dgemm.cpp

```

1 // clang-format off
2 ****
3 * Copyright 2019-2021 Intel Corporation.
4 *
5 * This software and the related documents are Intel copyrighted materials, and
6 * your use of them is governed by the express license under which they were
7 * provided to you (License). Unless the License provides otherwise, you may not
8 * use, modify, copy, publish, distribute, disclose or transmit this software or
9 * the related documents without Intel's prior written permission.
10 *
11 * This software and the related documents are provided as is, with no express
12 * or implied warranties, other than those that are expressly stated in the
13 * License.
14 ****
15
16 #include <stdio.h>
17 #include <omp.h>
18 #include "mkl.h"
19 #include "mkl_omp_offload.h"
20 #include "common.h"
21

```

(continues on next page)

(continued from previous page)

```
22 int dnum = 0;
23
24 int main() {
25
26     double *a, *b, *c, *c_ref, alpha, beta;
27     MKL_INT m, n, k, lda, ldb, ldc, i, j;
28     MKL_INT sizea, sizeb, sizec;
29
30     alpha = 1.0;
31     beta = 1.0;
32
33     m = 1449;
34     n = 1120;
35     k = 1083;
36
37     lda = m;
38     ldb = k;
39     ldc = m;
40
41     sizea = lda * k;
42     sizeb = ldb * n;
43     sizec = ldc * n;
44
45     // allocate matrices
46     a = (double *)mkl_malloc((lda * k) * sizeof(double), 64);
47     b = (double *)mkl_malloc(ldb * n * sizeof(double), 64);
48     c = (double *)mkl_malloc(ldc * n * sizeof(double), 64);
49     c_ref = (double *)mkl_malloc(ldc * n * sizeof(double), 64);
50
51     if ((a == NULL) || (b == NULL) || (c == NULL)) {
52         printf("Cannot allocate matrices\n");
53         return 1;
54     }
55
56     // initialize matrices
57     init_double_array(lda * k, a, 1);
58     init_double_array(ldb * n, b, 1);
59     init_double_array(sizec, c, 1);
60     for (i = 0; i < sizec; i++) c_ref[i] = c[i];
61
62     MKL_INT bound_m = (m > 10) ? 10 : m;
63     MKL_INT bound_n = (n > 10) ? 10 : n;
64
65     // run gemm on host, use standard oneMKL interface
66     cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, m, n, k, alpha, a, lda, b, ldb, beta,
67     ↵ c_ref, ldc);
68
69     // run gemm on gpu, use standard oneMKL interface within a variant dispatch construct
70     #pragma omp target data map(to:a[0:sizea],b[0:sizeb]) map(tofrom:c[0:sizec]) device(dnum)
71     {
72         #pragma omp target variant dispatch device(dnum) use_device_ptr(a, b, c)
```

(continues on next page)

(continued from previous page)

```

72     {
73         cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, m, n, k, alpha, a, lda, b,
74         ldb, beta, c, ldc);
75     }
76 }
77
78     double real;
79     for (i = 0; i < m; i++) {
80         for (j = 0; j < n; j++) {
81             real = c[i + ldc * j] - c_ref[i + ldc * j];
82             real = (real > 0) ? real : -real;
83             if (real > 0.0001) {
84 #ifdef MKL_ILP64
85                 printf("c[%lld][%lld] != c_ref[%lld][%lld], computed value is %f, reference
86                 value is %f, difference is %f\n",
87                 i, j, i, j, c[i + ldc * j], c_ref[i + ldc * j], real);
88 #else
89                 printf("c[%d][%d] != c_ref[%d][%d], computed value is %f, reference value is %f,
90                 difference is %f\n",
91                 i, j, i, j, c[i + ldc * j], c_ref[i + ldc * j], real);
92 #endif
93             mkl_free(a);
94             mkl_free(b);
95             mkl_free(c);
96             mkl_free(c_ref);
97             return 1;
98         }
99     }
100    printf("Upper left corner of the C matrix:\n");
101    printf("C matrix:\n");
102    for (i = 0; i < bound_m; i++) {
103        for (j = 0; j < bound_n; j++) {
104            printf("%f\t", c[i + ldc * j]);
105        }
106        printf("\n");
107    }
108
109    printf("Reference matrix:\n");
110    for (i = 0; i < bound_m; i++) {
111        for (j = 0; j < bound_n; j++) {
112            printf("%f\t", c_ref[i + ldc * j]);
113        }
114        printf("\n");
115    }
116
117    mkl_free(a);
118    mkl_free(b);
119    mkl_free(c);

```

(continues on next page)

(continued from previous page)

```
120     mkl_free(c_ref);
121     return 0;
122 }
```

For more information about the Intel oneAPI Math Kernel Library, see:

- [Developer Reference for Intel® oneAPI Math Kernel Library - C](#)

Note: The following configuration was used when collecting OpenMP performance numbers:

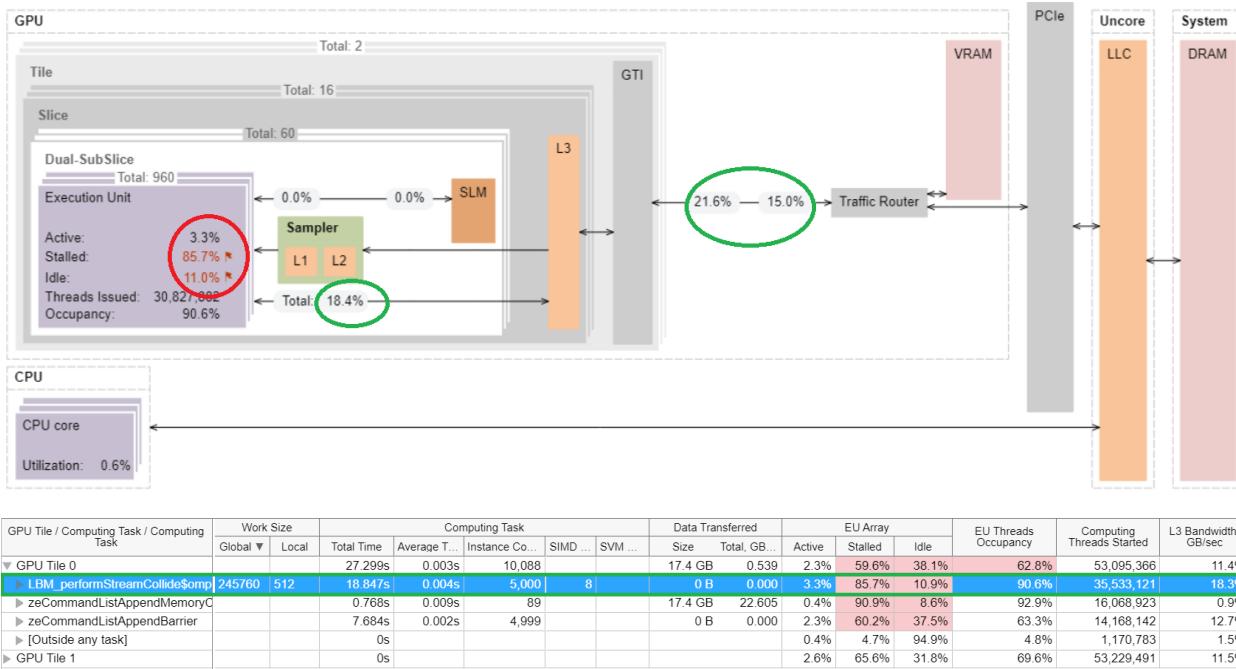
- Internal versions of the Intel compilers and GPU driver
- GPU: ATS-P B0, 2-Tile
- L0-plugin
- Introduced a dummy target construct at the beginning of a program, so as not to measure startup time.
- Used Just-In-Time (JIT) compilation mode.
- Used 1-Tile only (no implicit or explicit scaling).

## 12.5 NDA capabilities of VTune™ Profiler

VTune Profiler has some capabilities that are not publicly available. These capabilities include support for hardware that is not currently released. These NDA features include multi-tile and multi-GPU functionality. This chapter will document how to access these features. A special build of VTune and certain environment settings are required to use these features. We will also cover an overview of performance tuning considerations that come with this new GPU functionality.

VTune Profiler has some powerful feature that can assist in tracking down bottlenecks running on your GPU, some useful features include:

- Assess scaling efficiency
- See tiles utilization overtime
- Parallel vs sequential use
- Investigate Unexpected cross tile memory accesses
- Frequency-related problems.



## 12.5.1 Running VTune on a Multi-tile GPU

Some of the latest Intel® GPUs have a multi-tile architecture. This architecture presents some new performance considerations. If you are using an older version of VTune (prior to 2021.7), To collect GPU HW metrics from all tiles set the following environment variable.

```
export AMPLXE_EXPERIMENTAL=gpu-multi-tile-metrics
```

On a multi-tile GPU there are two ways for you to run your application: Implicit scaling, explicit scaling. Implicit scaling does not require code modifications, the GPU driver distributes thread-groups of a **device** kernel to both tiles. For implicit scaling case: all LO and OCL tasks are attributed to Tile0 currently. Pending for LO API extension [https://gitlab.devtools.intel.com/one-api/level\\_zero/issues/522](https://gitlab.devtools.intel.com/one-api/level_zero/issues/522) to allow per-tile tasks attribution for implicit scaling.

To use implicit scaling, set the following environment variable.

```
export EnableWalkerPartition=1
```

Explicit scaling requires code modifications, you need to specify to which tile you would like your kernel to run. Using explicit scaling you can often achieve better performance because of additional control that it gives you. Full profiling support is available for explicit scaling mode including per-Tile attribution of GPU HW metrics and compute kernels

## Application scaling

To test whether application is scaling on 2 tiles. First try running on a single tile using the ZE\_AFFINITY\_MASK environment variable.

```
export ZE_AFFINITY_MASK=0.0
```

```
export ZE_AFFINITY_MASK=0.1
```

Then compare whether application is running faster on single tile vs 2 tile with implicit scaling.

Because of NUMA (and other issues), 2 tiles may not scale without tuning. There are some architecture details that you need to consider:

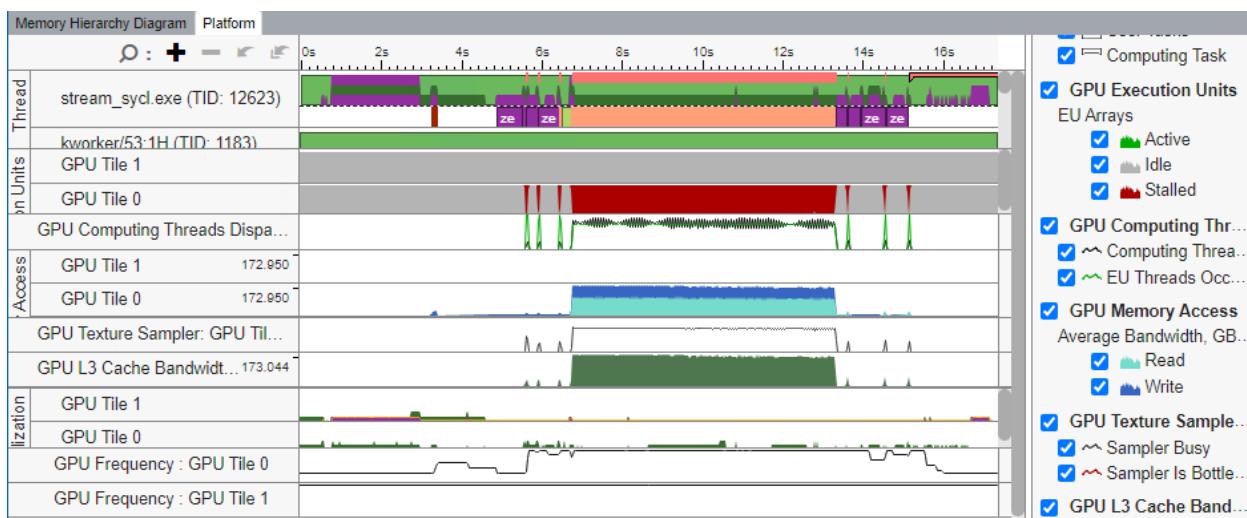
- Each ATS tile has its own L3 cache; these caches are **not coherent**.
- All communication between tiles takes place via memory.
- Extra L3 flushes are required for:
  - Atomic operations
  - Synchronization between kernels
- ATS shared allocations are managed by the GPU driver.
- Increases PCIe traffic and prevents certain use-cases:
  - Entire allocations are migrated.
  - The host and device cannot access allocations concurrently.
  - Allocations must be created with special function calls.
- Devices cannot access memory allocated on another device.

To understand your kernels performance it is important to understand:

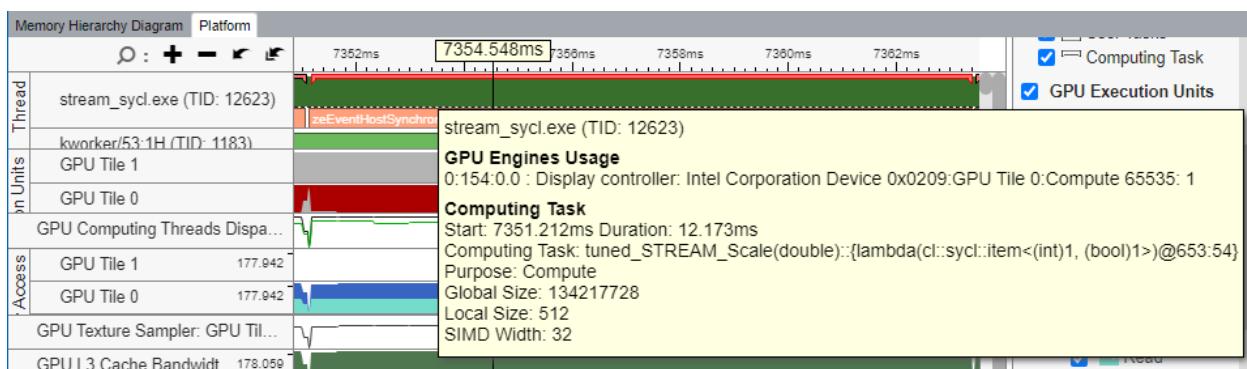
- How kernels utilize the execution resources of different tiles
- How allocations are spread across the memory of different tiles
- Where allocations “live” and how they move around the system

## Using the VTune GUI for multi-tile analysis

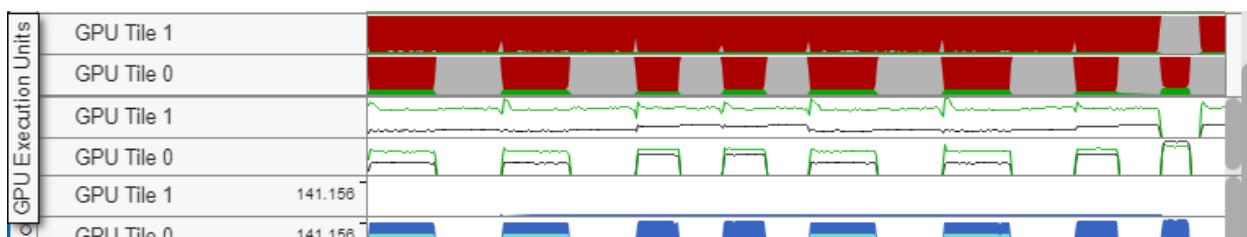
In the VTune GUI you can see all of your GPU execution units on a per tile basis, as shown below.



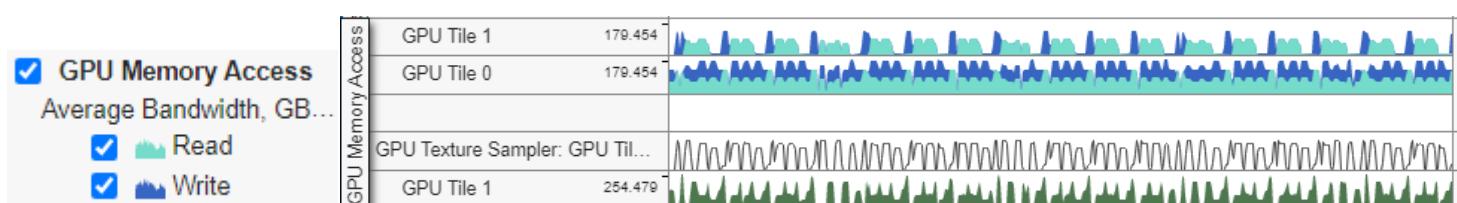
GPU Tile / Computing Task	Work Size		Computing Task					Size
	Global ▾	Local	Total Time	Average Time	Instance Count	SIMD Width	SVM Usage Type	
▼ GPU Tile 0			9.024s	0.022s	405			5 G
▶ tuned_STREAM_Copy(void)::(lambda<int>@653:54)	1342177...	512	1.251s	0.013s	100	32		0
▶ tuned_STREAM_Scale(double)::(lambda<(int)1, (bool)1>@653:54)	1342177...	512	1.250s	0.012s	100	32		0



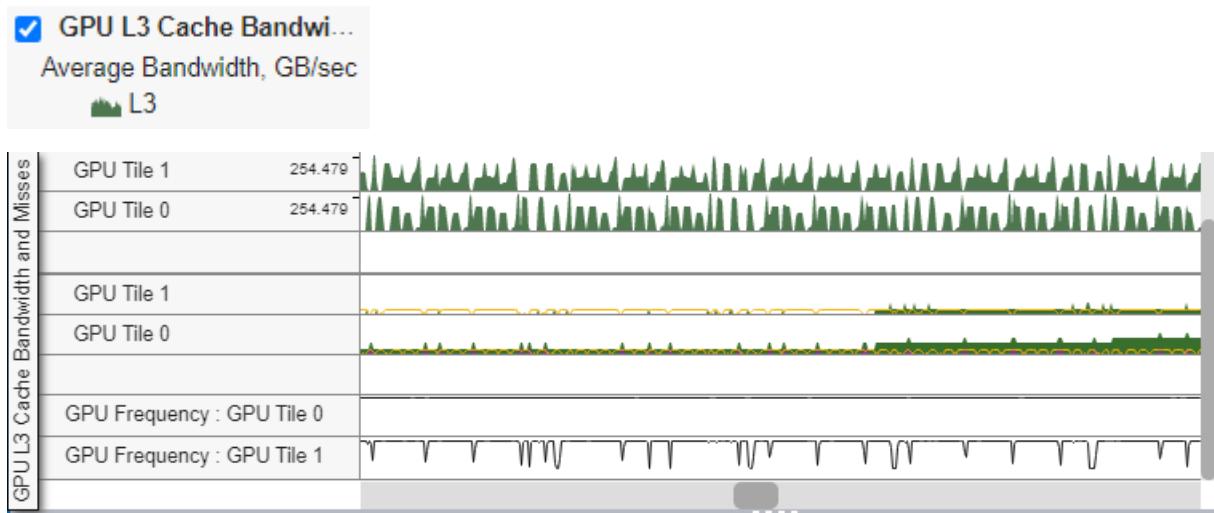
You can also correlate your activity on the tiles and see when one is idle or stalled.



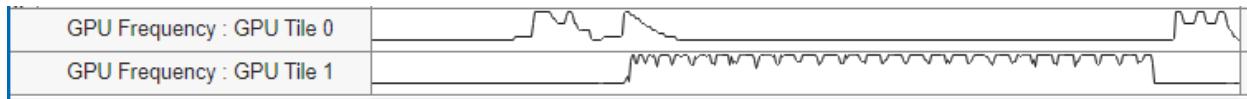
VTune also shows you your read/write memory bandwidth on each tile across the kernel timeline.



You can also view how well you are using your L3 cache and the cache bandwidth.



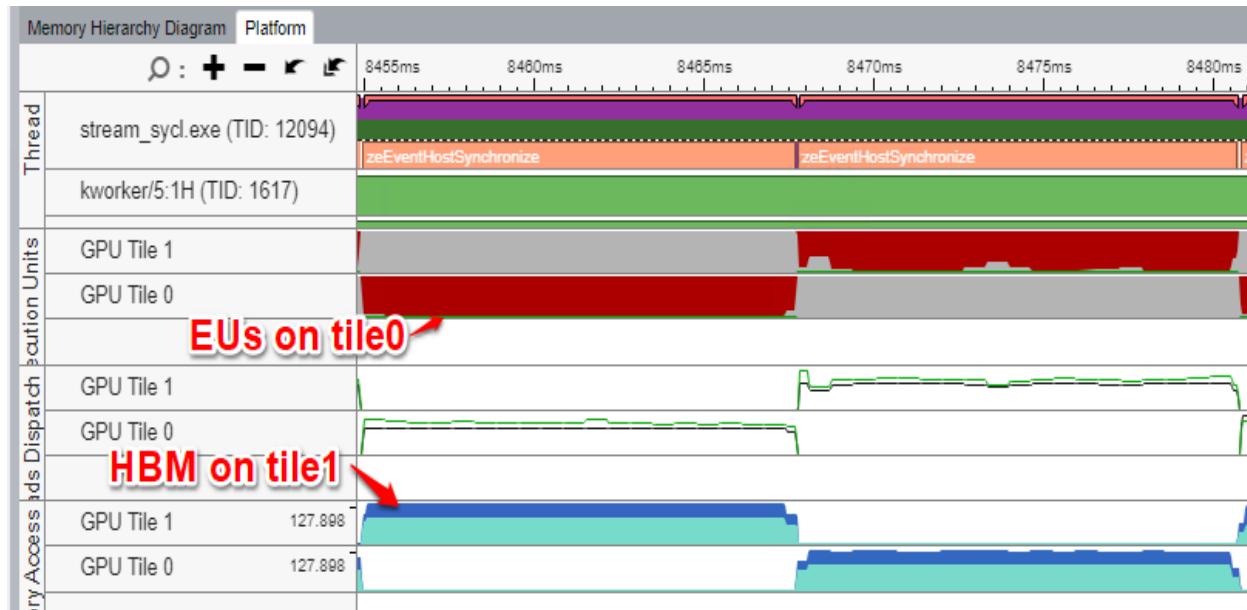
GPU Frequency is one factor that can affect your performance. You should note frequency changes, especially if the tiles are operating at difference frequencies. This will require additional investigation



## Bringing it all together

As we have shown, VTune has many different metrics to assist in performance tuning. It is often the case that you will need to correlate across these metrics. For example, you can correlate your execution with your memory access.

In the example below, we are executing on tile 0 when we are accessing memory on Tile 1. This requires additional tuning and investigation.



## 12.5.2 Occupancy analysis

Occupancy – sum of all cycles when thread slots have a thread scheduled

The idea is to estimate Peak Occupancy based on kernel parameters

Limiting factors:

- Global and local sizes
- SLM size requested
- Barriers usage (in progress)
- Tiny/huge kernels scheduling issues (in analysis)





## GPU Compute/Media Hotspots (preview)

Analyze the most time-consuming GPU kernels, characterize GPU utilization based on GPU hardware metrics, identify performance issues caused by memory latency or inefficient kernel algorithms, and analyze GPU instruction frequency per certain instruction types. [Learn more](#)

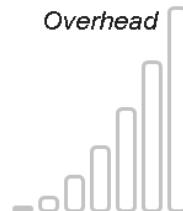
### Characterization ?

Compute Basic (with global/local memory accesses) ▼

GPU sampling interval, ms

1

### Overhead



Analyze memory bandwidth

Trace GPU programming APIs

### Source Analysis ?

#### Details >

```
vtune --collect gpu-hotspots -knob characterization-mode=global-local-accesses -- ./app
```

### EU Array Stalled/Idle ?: 56.7% ↗ ⬇

Analyze the average value of EU Array Stalled/Idle metric and identify why EUs were waiting for resources instead of doing computations. This metric is critical for compute-bound applications. Explore typical reasons for this kind of inefficiency listed below.

GPU L3 Bandwidth Bound ?: 38.6% ⬇

DRAM Bandwidth Bound ?: 0.0% ⬇

Occupancy ?: 80.4% ↗ ⬇

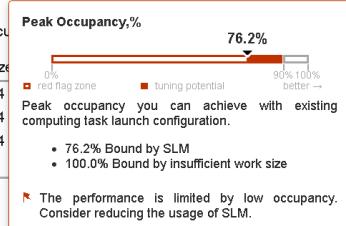
Identify too large or too small computing tasks with low occupancy that make the EU array idle while waiting for the scheduler. Note that frequent SLM accesses and barriers may affect the maximum possible occupancy.

#### Hottest GPU Computing Tasks with Low Occupancy ⬇

This section lists the most active computing tasks running on the GPU with a low Occupancy.

Computing Task	Total Time <small>?</small>	Global Size <small>?</small>	Local Size <small>?</small>
kernel_ocl_path_trace_shadow_blocked_dl	32.492s	128 x 185	64
kernel_ocl_path_trace_shader_sort	21.426s	128 x 185	64
kernel_ocl_path_trace_shader_eval	17.506s	128 x 185	64
[Others]	14.209s		

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



Occupancy <small>?</small>	Occupancy <small>?</small>
100.0%	88.6% <small>↗</small>
76.2% <small>↗</small>	60.5% <small>↗</small>
100.0%	78.1% <small>↗</small>
0.0%	73.2% <small>↗</small>

### 12.5.3 Extended math and systolic pipelines profiling

The support of extended math and systolic pipelines is available as tech preview features in VTune. To use these features, set the following environment variable.

```
export AMPLXE_EXPERIMENTAL=gpu-euactivity
```

Extended math

```
vtune --collect gpu-hotspots -knob characterization-mode=euactivity7 -- ./app
```

OR for systolic

```
vtune --collect gpu-hotspots -knob characterization-mode=euactivity8 -- ./app
```



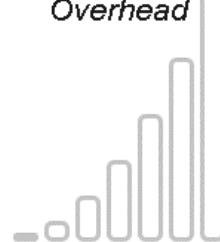
### GPU Compute/Media Hotspots (preview)

Analyze the most time-consuming GPU kernels, characterize GPU utilization based on GPU hardware metrics, identify performance issues caused by memory latency or inefficient kernel algorithms, and analyze GPU instruction frequency per certain instruction types. [Learn more](#)

Characterization ?

EU Activity 7

*Overhead*



GPU sampling interval, ms

1

Analyze memory and cross-socket bandwidth

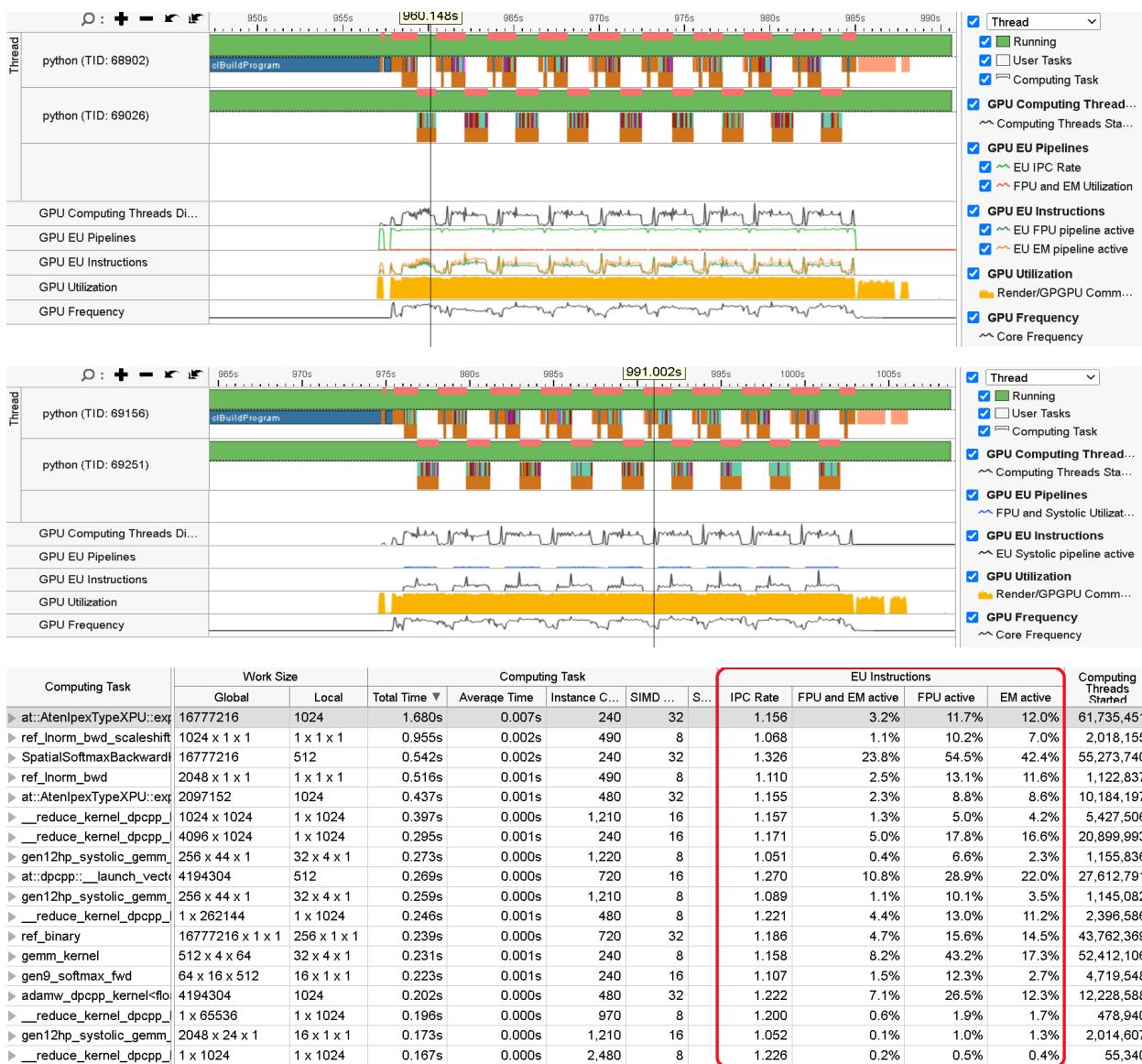
Trace GPU programming APIs

Analyze power usage

Source Analysis ?

Details

>



## 12.5.4 Multi-GPU

In addition to being able to analyze multiple tiles, VTune can also analyze multiple GPUs. Multiple GPUs are when you have more than one GPU cards attached to your system. If you are using an older version of VTune (prior to 2021.7), To analyze multiple GPUs, use the following environment variables.

```
export AMPLXE_EXPERIMENTAL=gpu-multi-adapter-metrics,gpu-multi-tile-metrics
```

To collect on a specific GPU, you need to identify each GPU device.

On **Linux**: run

```
'$ {lspci -D -d 8086::300;lspci -D -d 8086::380;}'
```

**0000:4d:00.0** Display controller: Intel Corporation Device 0206

0000:9a:00.0 Display controller: Intel Corporation Device 0206

You need to use this format [B:D:F] when specifying your GPU on collection.

Set -knob target-gpu = BDF in **decimal** format

For example, the following specifies GPU with BDF='0000:4d:00.0' to be profiled by VTune:

```
vtune -c gpu-hotspots -knob target-gpu=\ **0:77:0.0** -- *<your application>*
```

## 12.6 AOT Compiler Flags for ATS BO/AO

When compiling in AOT mode for ATS BO (instead of ATS A1) hardware you will need to add an extra option to get all the benefits of BO part. Here are some notes:

For AOT compilation, we currently recommend using "xehp" as the keyword:

```
icx -fiopenmp -fopenmp-targets=spir64_gen -Xopenmp-target-backend \
"-device xehp" offloading_success.c -o offloading_success.exe
```

IGC Compiler does different optimizations on ATS A1-part versus B0-part in JIT-mode. For example, in B0, the following environment variables are not needed: **IGC\_EnableA64WA** = 0 **IGC\_NoMaskWA** = 0.

OpenCL Offline Compiler (OCLOC) accepts "-revision\_id" parameter, aligned with how bspec defines steppings. ATS steppings are defined in <https://gfxspecs.intel.com/Predator/Home/Index/44470> (A1 == 0x1, B0 == 0x4)

For best performance on ATS B0 with AOT mode, use:

```
icx -fiopenmp -fopenmp-targets=spir64_gen -Xopenmp-target-backend \
"-device xehp" -Xopenmp-target-backend "-revision_id 4" source.c
```

A similar option can also be used with DPCPP AOT mode:

```
dpcpp -fsycl-targets=spir64_gen-unknown-unknown-sycldevice \
-Xsycl-target-backend "-device ats" -Xsycl-target-backend \
"-revision_id 4" source.c
```

For best performance on ATS A0 with AOT mode, use the same options but specify "-revision\_id 1":

```
icx -fiopenmp -fopenmp-targets=spir64_gen -Xopenmp-target-backend \
"-device xehp" -Xopenmp-target-backend "-revision_id 1" source.c
```

A similar option can also be used with DPCPP AOT mode:

```
dpcpp -fsycl-targets=spir64_gen-unknown-unknown-sycldevice \
-Xsycl-target-backend "-device ats" -Xsycl-target-backend \
"-revision_id 1" source.c
```

Note that in JIT-mode, IGC compiler knows the type of the hardware and will adjust automatically. The extra "-revision\_id" option above is only needed for AOT-mode.



## 13.0 Reference

For more information, see:

- [Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference](#)
- [Intel® oneAPI Programming Guide](#)
- [Intel® Fortran Compiler Classic and Intel® Fortran Compiler Developer Guide and Reference](#)
- [Get Started with OpenMP Offload to GPU for the Intel® oneAPI DPC/C++ Compiler and Intel® Fortran Compiler](#)
- [OpenMP Features and Extensions Supported in Intel® oneAPI DPC++/C++ Compiler](#)
- [Fortran Language and OpenMP Features Implemented in Intel® Fortran Compiler \(Beta\)](#)
- [Developer Reference for Intel® oneAPI Math Kernel Library - C](#)
- [OpenMP API 5.1 Specification](#)
- [OpenMP API 5.1 Examples](#)
- [Data Parallel C++, by James Reinders et al](#)
- [SYCL 2020 Specification](#)
- [oneAPI Level Zero Specification](#)



## 14.0 Terms and Conditions

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel, and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries. Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit [www.intel.com/benchmarks](http://www.intel.com/benchmarks).

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available security updates. See backup for configuration details. No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

\*Other names and brands may be claimed as the property of others. © Intel Corporation.