

# **oneAPI GPU Optimization Guide**

# Contents

## Chapter 1: oneAPI GPU Optimization Guide

Introduction .....	4
Getting Started .....	5
Parallelization .....	7
Intel® Xe GPU Architecture .....	7
General-Purpose Computing on GPU .....	17
Execution Model Overview .....	18
Thread Mapping and GPU Occupancy .....	19
Kernels .....	33
Sub-Groups and SIMD Vectorization .....	33
Removing Conditional Checks .....	40
Registers and Performance .....	43
Shared Local Memory .....	64
Pointer Aliasing and the Restrict Directive .....	71
Synchronization among Threads in a Kernel .....	75
Considerations for Selecting Work-Group Size .....	85
Prefetch .....	89
Reduction .....	93
Kernel Launch .....	99
Executing Multiple Kernels on the Device at the Same Time .....	101
Submitting Kernels to Multiple Queues .....	103
Avoiding Redundant Queue Constructions .....	106
Programming Intel® XMX Using SYCL Joint Matrix Extension .....	110
Doing I/O in the Kernel .....	112
Using Libraries for GPU Offload .....	115
Using Performance Libraries .....	115
Using Standard Library Functions in SYCL Kernels .....	117
Efficiently Implementing Fourier Correlation Using oneAPI Math Kernel Library (oneMKL) .....	119
Boost Matrix Multiplication Performance with Intel® Xe Matrix Extensions .....	127
Host/Device Memory, Buffer and USM .....	129
Unified Shared Memory Allocations .....	130
Performance Impact of USM and Buffers .....	131
Avoiding Moving Data Back and Forth between Host and Device .....	135
Optimizing Data Transfers .....	139
Avoiding Declaring Buffers in a Loop .....	144
Buffer Accessor Modes .....	147
Host/Device Coordination .....	152
Asynchronous and Overlapping Data Transfers Between Host and Device .....	152
Using Multiple Heterogeneous Devices .....	157
Compilation .....	159
Just-In-Time Compilation .....	159
Ahead-Of-Time Compilation .....	162
Specialization Constants .....	162
Accuracy Versus Performance Tradeoffs in Floating-Point Computations .....	168

---

OpenMP Offloading Tuning Guide.....	180
OpenMP Directives .....	180
OpenMP Execution Model .....	181
Terminology .....	181
Compiling and Running an OpenMP Application .....	182
Offloading oneMKL Computations onto the GPU.....	185
Tools for Analyzing Performance of OpenMP Applications .....	222
OpenMP Offload Best Practices.....	223
Multi-GPU and Multi-Stack Architecture and Programming.....	302
Multi-Stack GPU Architecture.....	302
Exposing the Device Hierarchy.....	308
FLAT Mode Programming.....	310
COMPOSITE Mode Programming .....	315
Using Intel® oneAPI Math Kernel Library (oneMKL).....	335
Using Intel® MPI Library .....	345
Advanced Topics .....	357
Terminology .....	370
Level Zero.....	375
Immediate Command Lists.....	375
Performance Profiling and Analysis .....	376
Using the Timers.....	377
Intel® VTuneTM Profiler .....	379
Intel® Advisor .....	399
Intel® Intercept Layer for OpenCLTM Applications .....	421
Performance Tools in Intel® Profiling Tools Interfaces for GPU .....	421
Configuring GPU Device .....	422
Media Graphics Computing on GPU .....	424
Optimizing Media Pipelines .....	424
Media Engine Hardware .....	424
Media API Options for Hardware Acceleration .....	426
Media Pipeline Parallelism .....	427
Media Pipeline Inter-operation and Memory Sharing .....	429
SYCL-Blur Example.....	434
Performance Analysis with Intel® Graphics Performance Analyzers.....	435
References .....	449
Terms and Conditions.....	450

# oneAPI GPU Optimization Guide

---

Welcome to the oneAPI GPU Optimization Guide. This document gives tips for getting the best GPU performance for oneAPI programs.

- [Introduction](#)
- [Getting Started](#)
- [Parallelization](#)
- [Intel® Xe GPU Architecture](#)
- [General-Purpose Computing on GPU](#)
- [Media Graphics Computing on GPU](#)
- [References](#)
- [Terms and Conditions](#)

## Introduction

---

Designing high-performance heterogeneous-computing software taking advantages of accelerators like, GPUs, for example, requires you to think differently than you do for traditional homogeneous-computing software. You need to be aware of the hardware on which your code is intended to run, and the characteristics that control the performance of that hardware. Your goal is to structure the code such that it produces correct answers in a way that maximizes the hardware's ability.

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.

## Productive Performance

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.

## Performance Considerations

The first consideration 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 access 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, including memory access to computation ratio, and help you identify the best opportunities for parallel execution. The profilers in Intel® 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 code 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 device, parallelism across the compute cores, overlap between compute and memory accesses, concurrent pipelines, and vector computations. Exploiting all levels of parallelism requires a good understanding of the GPU architecture, the programming language, the libraries, and the analysis tools providing performance insights at your disposal.

*Keep all the compute resources busy.* There must be enough 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 and these 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.

## Profiling and Tuning

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 efforts.

## Source Code Examples

Throughout the guide, we use real code examples to illustrate optimization techniques. All the examples in this guide can be found in the [oneAPI-samples GitHub repo](#). Now it is the perfect time to download the examples and set up your environment by following the instructions there.

We try hard to keep the examples as short and easy to follow as possible so optimization techniques in each example are not clouded by complexities of the example and can be quickly grasped. Code snippets from the examples are referenced throughout the text.

There is an old saying “I hear and I forget. I see and I remember. I do and I understand.”. It is strongly suggested that you pause to try the example on a real machine when a code snippet is encountered while reading the text.

Welcome to Intel® oneAPI GPU Optimization Guide!

---

# Getting Started

---

Three key concepts govern software optimization for an accelerator. These concepts should guide your optimization efforts.

## 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 you execution time is concentrated, you can still do well.

## Locality

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, and caches 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.

## Work Size

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, imagine 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.

## 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.

### Programming Language and APIs

There are many parallel programming languages and APIs that can be used to express parallelism. oneAPI is an open industry standard for heterogeneous computing. It supports parallel program development through the SYCL\* framework. The Intel® oneAPI products have a number of code generation tools to convert source 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.

### 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.

### Libraries

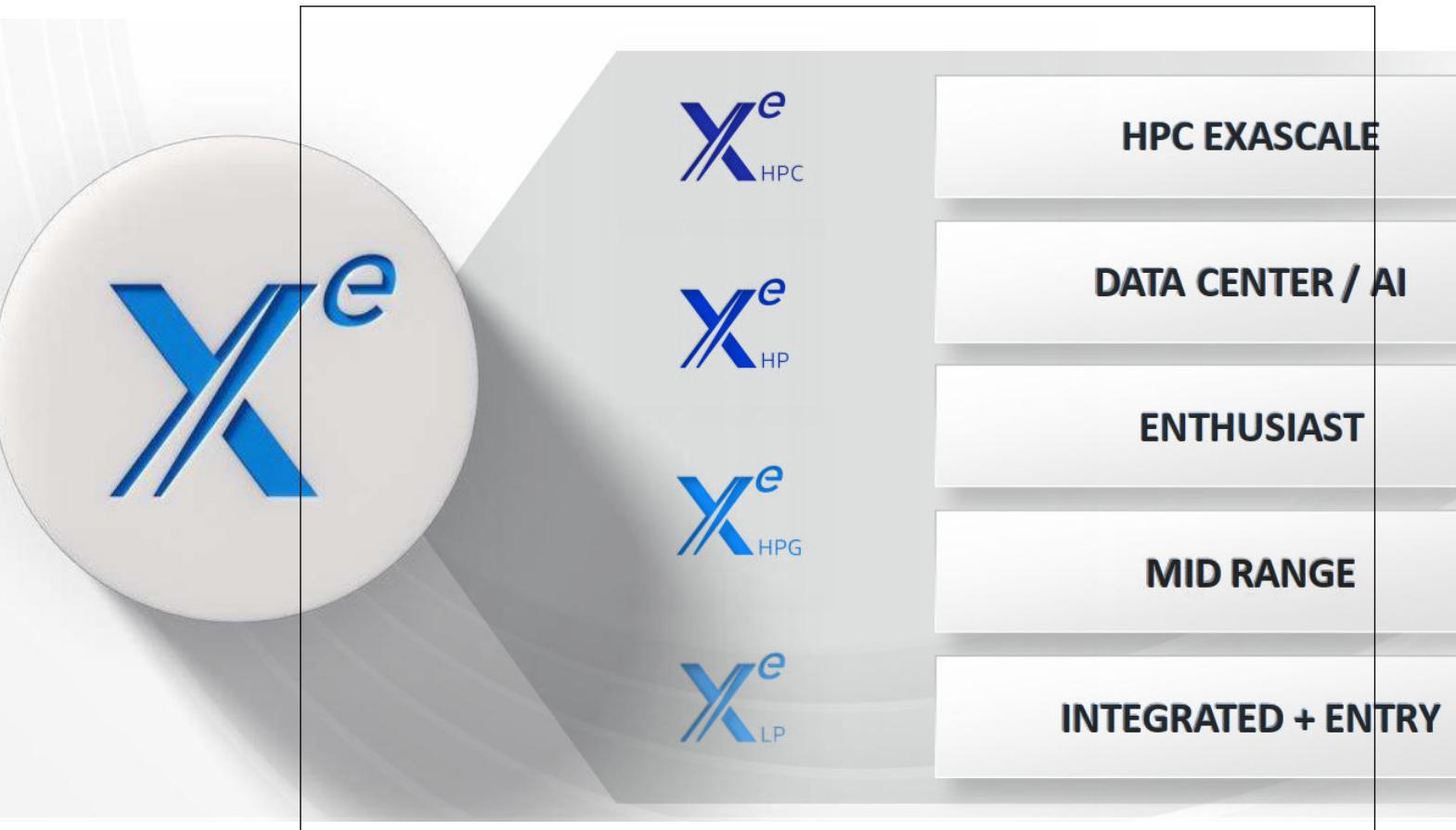
oneAPI includes a number of libraries like oneTBB, oneMKL, oneDNN, and Intel Video Processing Library (Intel VPL) that provide highly-optimized versions of common computational operations run across a variety of accelerator architectures. Depending on the needs of the application, a user can directly 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.

## Intel® Xe GPU Architecture

---

The Intel® Xe GPU family consists of a series of microarchitectures, ranging from integrated/low power (Xe-LP), to enthusiast/high performance gaming (Xe-HPG), data center/AI (Xe-HP) and high performance computing (Xe-HPC).

### Intel® Iris® Xe family



This chapter introduces Xe GPU family microarchitectures and configuration parameters.

## Xe-LP Execution Units (EUs)

An Execution Unit (EU) is the smallest thread-level building block of the Intel® Iris® Xe-LP GPU architecture. Each EU is simultaneously multithreaded (SMT) with seven threads. The primary computation unit consists of a 8-wide Single Instruction Multiple Data (SIMD) Arithmetic Logic Units (ALU) supporting SIMD8 FP/INT operations and a 2-wide SIMD ALU supporting SIMD2 extended math operations. Each hardware thread has 128 general-purpose registers (GRF) of 32B wide.

### Xe-LP-EU



Xe-LP EU supports diverse data types FP16, INT16 and INT8 for AI applications. The below table shows the EU operation throughput of Xe-LP GPU for various data types.

#### Intel® Iris® Xe-LP GPU Compute Throughput Rates (Ops/clock/EU)

FP32	FP16	INT32	INT16	INT 8
8	16	8	16	32 (DP4A)

## Xe-LP Dual Subslices

Each Xe-LP Dual Subslice (DSS) consists of an EU array of 16 EUs, an instruction cache, a local thread dispatcher, Shared Local Memory (SLM), and a data port of 128B/cycle. It is called dual subslice because the hardware can pair two EUs for SIMD16 executions.

The SLM consists of 128KB low latency and high bandwidth memory accessible from the EUs in the subslice. One important usage of SLM is to share atomic data and signals among the 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 128KB 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.

The following table summarizes the computing capacity of a Intel® Iris® Xe-LP subslice.

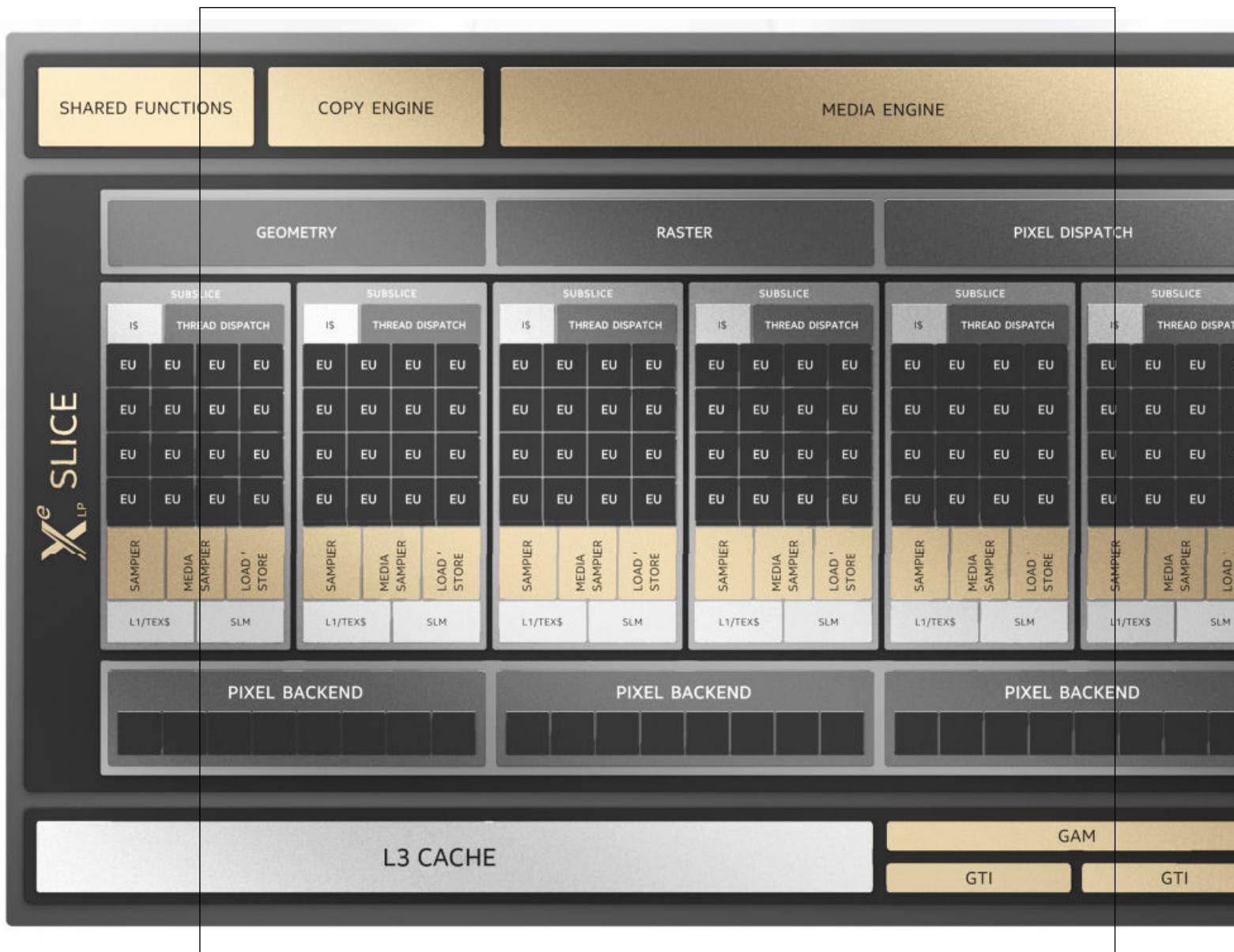
**Intel® Iris® Xe-LP subslice computing capacity**

EU <sup>s</sup>	Threads	Operations
16	$7 \times 16 = 112$	$112 \times 8 = 896$

## X<sup>e</sup>-LP Slice

Each X<sup>e</sup>-LP slice consists of six (dual) subslices for a total of 96 EUs, up to 16MB L2 cache, 128B/cycle bandwidth to L2 and 128B/cycle bandwidth to memory.

### Xe-LP slice



## X<sup>e</sup>-Core

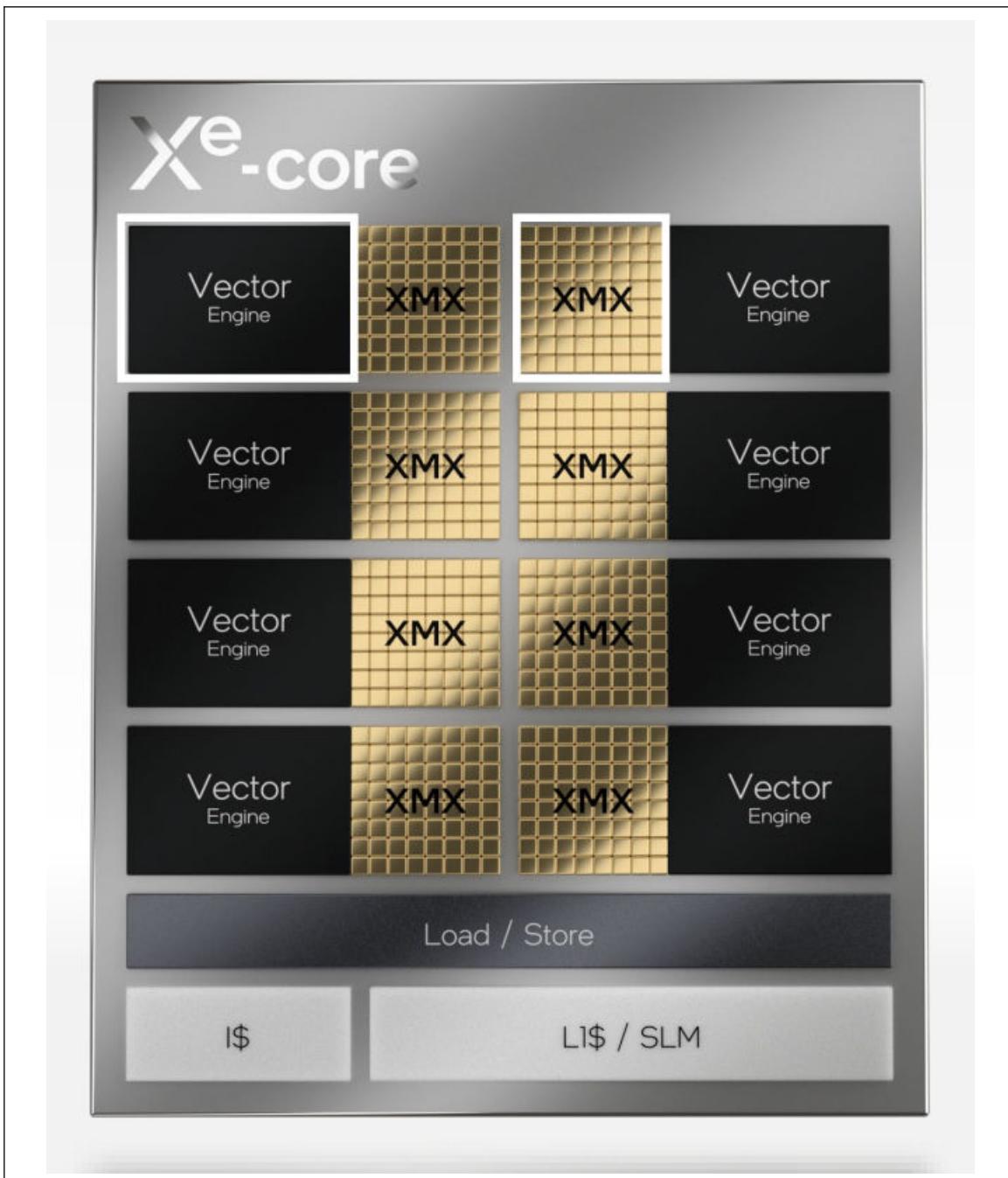
Unlike the X<sup>e</sup>-LP and prior generations of Intel GPUs that used the Execution Unit (EU) as a compute unit, X<sup>e</sup>-HPG and X<sup>e</sup>-HPC use the X<sup>e</sup>-core. This is similar to an X<sup>e</sup>-LP dual subslice.

An X<sup>e</sup>-core contains vector and matrix ALUs, which are referred to as vector and matrix engines.

An X<sup>e</sup>-core of the X<sup>e</sup>-HPC GPU contains 8 vector and 8 matrix engines, alongside a large 512KB L1 cache/SLM. It powers the Intel® Data Center GPU Max Series. Each vector engine is 512 bit wide supporting 16 FP32 SIMD operations with fused FMAs. With 8 vector engines, the X<sup>e</sup>-core delivers 512 FP16, 256 FP32 and

128 FP64 operations/cycle. Each matrix engine is 4096 bit wide. With 8 matrix engines, the X<sup>e</sup>-core delivers 8192 int8 and 4096 FP16/BF16 operations/cycle. The X<sup>e</sup>-core provides 1024B/cycle load/store bandwidth to the memory system.

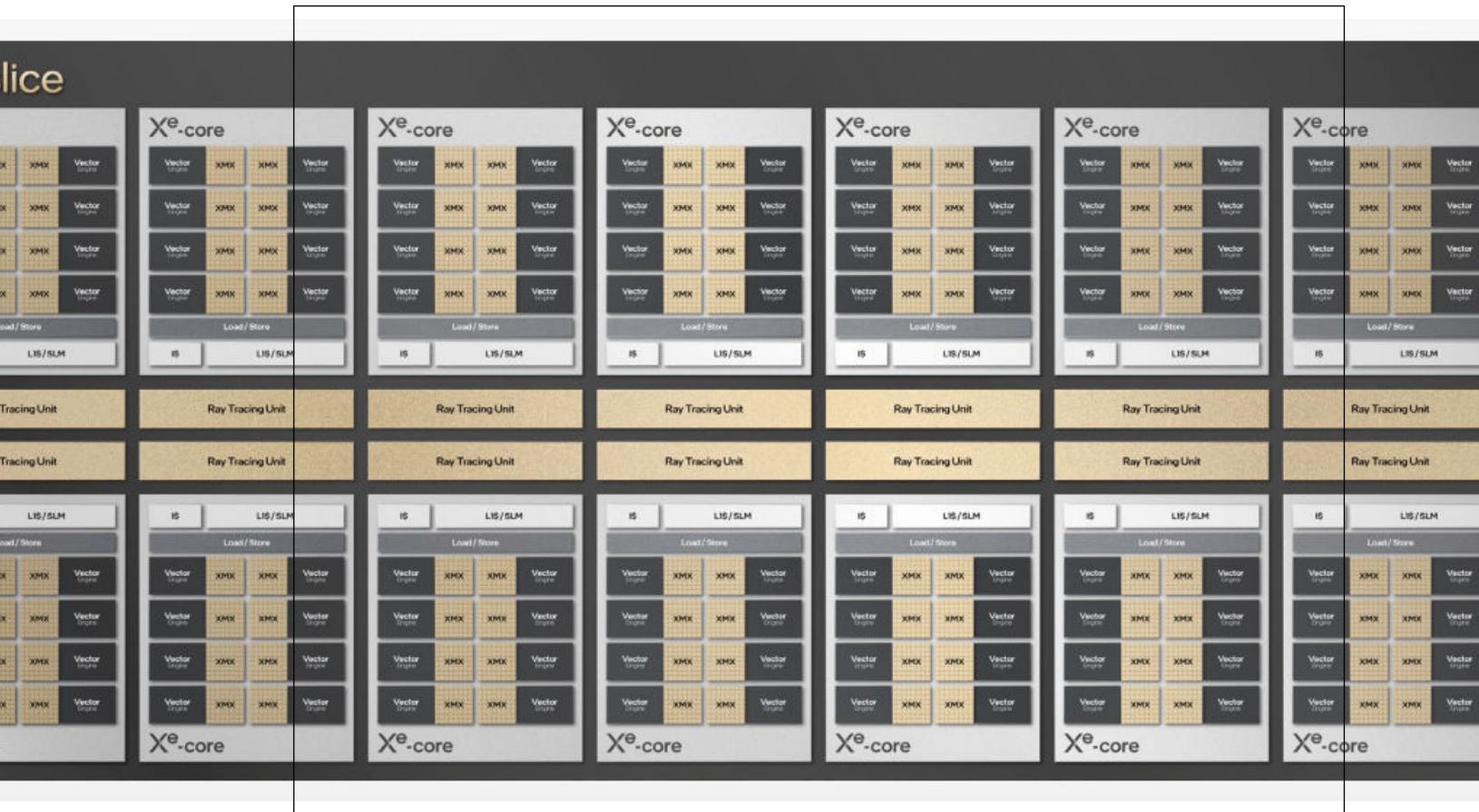
### Xe-core



## X<sup>e</sup>-Slice

An X<sup>e</sup>-slice contains 16 X<sup>e</sup>-core for a total of 8MB L1 cache, 16 ray tracing units and 1 hardware context.

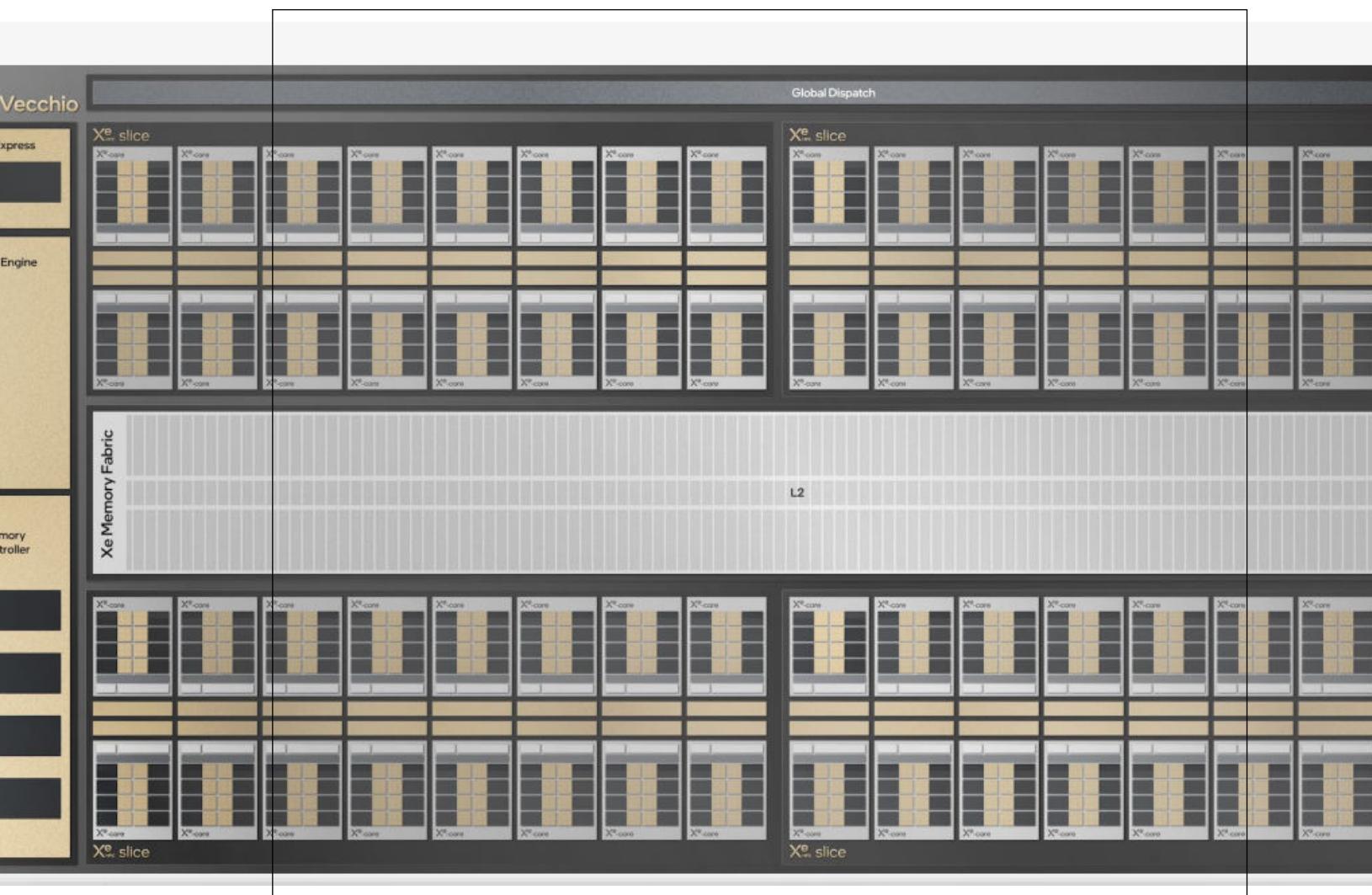
### Xe-slice



## Xe-Stack

An Xe-stack contains up to 4 Xe-slice: 64 Xe-cores, 64 ray tracing units, 4 hardware contexts, 4 HBM2e controllers, 1 media engine, and 8 Xe-Links of high speed coherent fabric. It also contains a shared L2 cache.

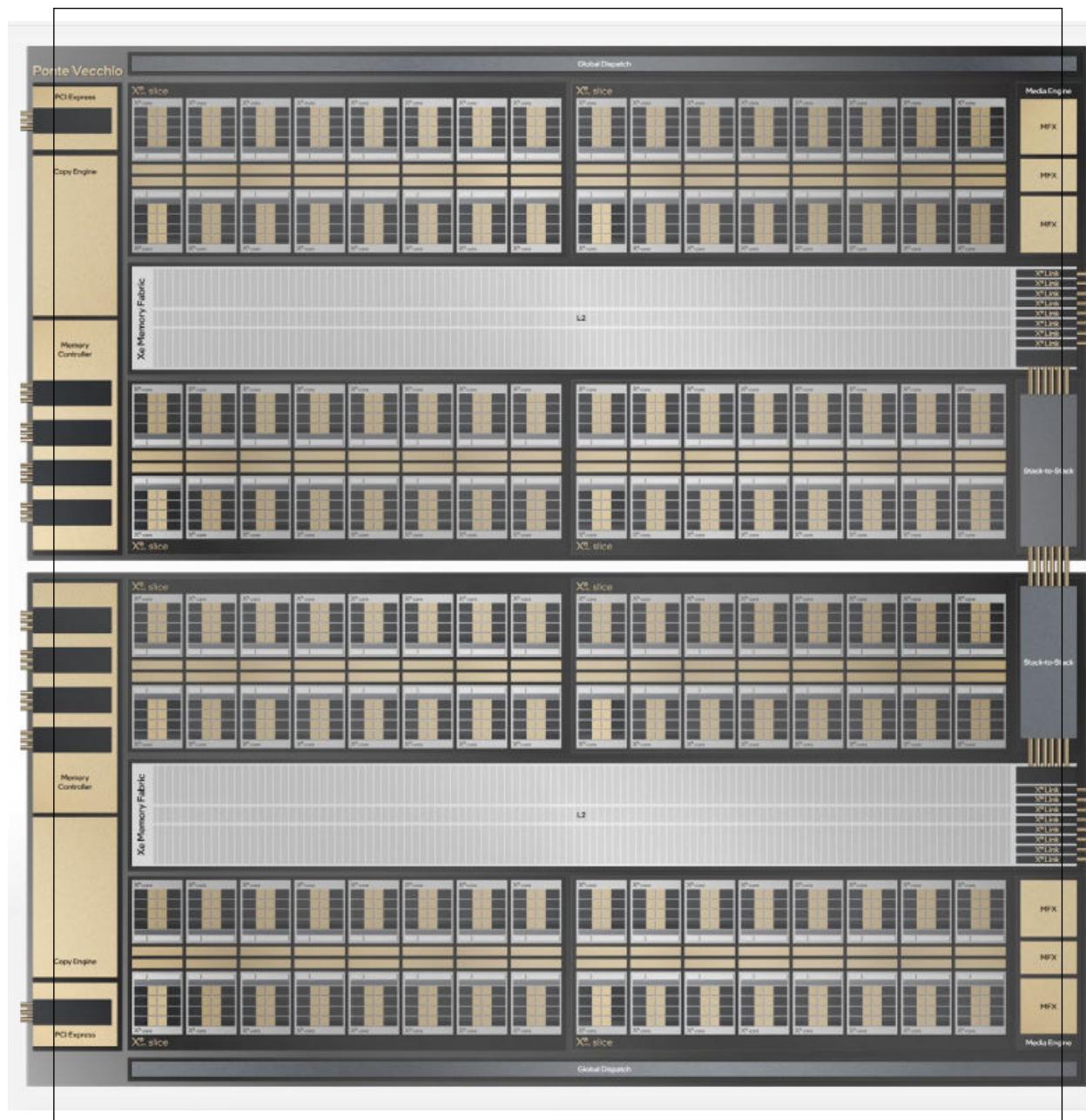
### Xe-stack



## X<sup>e</sup>-HPC 2-Stack Data Center GPU Max

An X<sup>e</sup>-HPC 2-stack Data Center GPU Max, previously code named Ponte Vecchio or PVC, consists of up to 2 stacks:: 8 slices, 128 X<sup>e</sup>-cores, 128 ray tracing units, 8 hardware contexts, 8 HBM2e controllers, and 16 X<sup>e</sup>-Links.

## Xe-HPC 2-Stack



## X<sup>e</sup>-HPG GPU

X<sup>e</sup>-HPG is the enthusiast or high performance gaming variant of the X<sup>e</sup> architecture. The microarchitecture is focused on graphics performance and supports hardware-accelerated ray tracing.

An X<sup>e</sup>-core of the X<sup>e</sup>-HPG GPU contains 16 vector and 16 matrix engines. It powers the Intel® Arc GPUs. Each vector engine is 256 bit wide, supporting 8 FP32 SIMD operations with fused FMAs. With 16 vector engines, the X<sup>e</sup>-core delivers 256 FP32 operations/cycle. Each matrix engine is 1024 bit wide. With 16 matrix engines, the X<sup>e</sup>-core delivers 4096 int8 and 2048 FP16/BF16 operations/cycle. The X<sup>e</sup>-core provides 512B/cycle load/store bandwidth to the memory system.

An X<sup>e</sup>-HPG GPU consists of 8 X<sup>e</sup>-HPG-slice, which contains up to 4 X<sup>e</sup>-HPG-cores for a total of 4096 FP32 ALU units/shader cores.

## X<sup>e</sup>- Intel® Data Center GPU Flex Series

Intel® Data Center GPU Flex Series come in two configurations. The 150W option has 32 X<sup>e</sup>-cores on a PCIe Gen4 card. The 75W option has two GPUs for 16 X<sup>e</sup>-cores (8 X<sup>e</sup>-cores per GPU). Both configurations come with 4 X<sup>e</sup> media engines, the industry's first AV1 hardware encoder and accelerator for data center, GDDR6 memory, ray tracing units, and built-in XMX AI acceleration.

Intel® Data Center GPU Flex Series are derivatives of the X<sup>e</sup>-HPG GPUs. An Intel® Data Center GPU Flex 170 consists of 8 X<sup>e</sup>-HPG-slices for a total of 32 X<sup>e</sup>-cores with 4096 FP32 ALU units/shader cores.

Targeting data center cloud gaming, media streaming and video analytics applications, Intel® Data Center GPU Flex Series provide hardware accelerated AV1 encoder, delivering a 30% bit-rate improvement without compromising on quality. It supports 8 simultaneous 4K streams or more than 30 1080p streams per card. AI models can be applied to the decoded streams utilizing Intel® Data Center GPU Flex Series' X<sup>e</sup>-cores.

Media streaming and delivery software stacks lean on Intel® Video Processing Library (Intel VPL) to decode and encode acceleration for all the major codecs including AV1. Media distributors can choose from the two leading media frameworks FFmpeg or GStreamer, both enabled for acceleration with Intel VPL on Intel CPUs and GPUs.

In parallel to Intel VPL accelerating decoding and encoding of media streams, oneDNN (oneAPI Deep Neural Network library) delivers AI optimized kernels enabled to accelerate inference modes in TensorFlow or PyTorch frameworks, or with the OpenVINO model optimizer and inference engine to further accelerate inference and speed customer deployment of their workloads.

## Terminology and Configuration Summary

The following table maps legacy GPU terminologies (used in Generation 9 through Generation 12 Intel® Core™ architectures) to their new names in the Intel® Iris® X<sup>e</sup> GPU (Generation 12.7 and newer) architecture paradigm.

### Architecture Terminology Changes

Old Term	New Intel Term	Generic Term	New Abbreviation
Execution Unit (EU)	X <sup>e</sup> Vector Engine	Vector Engine	XVE
Systolic/"DPAS part of EU"	X <sup>e</sup> Matrix eXtension	Matrix Engine	XMX
Subslice (SS) or Dual Subslice (DSS)	X <sup>e</sup> -core	NA	XC
Slice	Render Slice / Compute Slice	Slice	SLC
Tile	Stack	Stack	STK

The following table lists the hardware characteristics across the X<sup>e</sup> family GPUs.

## Xe Configurations

Architecture	Xe-LP (TGL)	Xe-HPG (Arc A770)	Xe-HPG (Data Center GPU Flex 170)	Xe-HPC (Data Center GPU Max 1550)
Slice count	1	8	8	4 x 2
XC (DSS/SS) count	6	32	32	64 x 2
XVE (EU) / XC	16	16	16	8
XVE count	96	512	512	512 x 2
Threads / XVE	7	8	8	8
Thread count	672	4096	4096	4096 x 2
FLOPs / clk - single precision, MAD	1536	8192	8192	16384 x 2
FLOPs / clk - double precision, MAD	NA	NA	NA	16384 x 2
FLOPs / clk - FP16 DP4AS	NA	65536	65536	262144 x 2
GTI bandwidth bytes / unslice-clk	r:128, w:128	r:512, w:512	r:512, w:512	r:1024, w:1024
LL cache size	3.84MB	16MB	16MB	up to 408MB
SLM size	$6 \times 128KB$	$32 \times 128KB$	$32 \times 128KB$	$64 \times 128KB$
FMAD, SP (ops / XVE / clk)	8	8	8	16
SQRT, SP (ops / XVE / clk)	2	2	2	4

## General-Purpose Computing on GPU

Traditionally, GPUs are used for creating computer graphics such as images, videos, etc. Due to their large number of execution units for massively parallelism, modern GPUs are also used for computing tasks that are conventionally performed on CPU. This is commonly referred to as General-Purpose Computing on GPU or GPGPU.

Many high performance computing and machine learning applications benefit greatly from GPGPU.

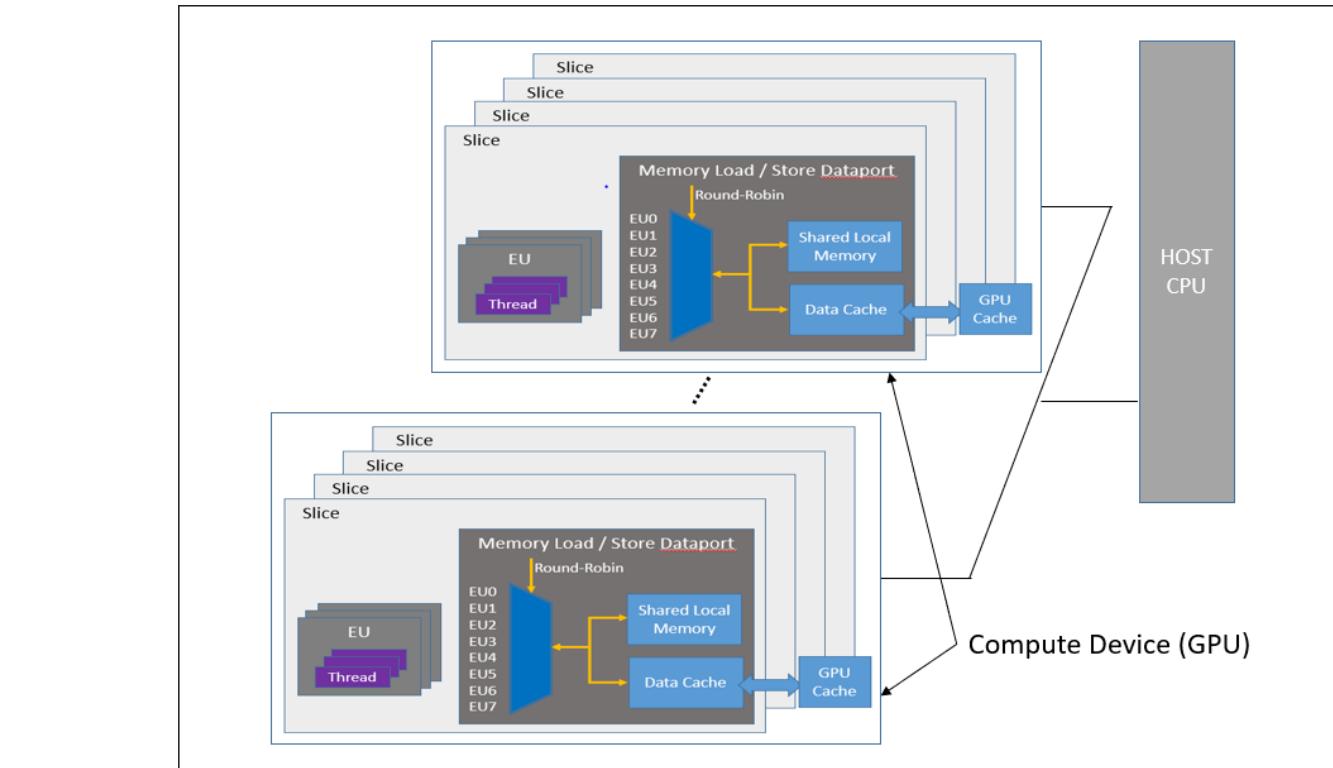
- [Execution Model Overview](#)
- [Thread Mapping and GPU Occupancy](#)
- [Kernels](#)
- [Using Libraries for GPU Offload](#)
- [Host/Device Memory, Buffer and USM](#)
- [Inter-process Communication](#)
- [Host/Device Coordination](#)
- [Using Multiple Heterogeneous Devices](#)

- [Compilation](#)
- [OpenMP Offloading Tuning Guide](#)
- [Multi-GPU and Multi-Stack Architecture and Programming](#)
- [Level Zero](#)
- [Performance Profiling and Analysis](#)
- [Configuring GPU Device](#)

## 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 many GPU Compute Engines (CE), also known as Execution Units (EU) or X<sup>e</sup> Vector Engines (XVE). The compute devices may also include caches, shared local memory (SLM), high-bandwidth memory (HBM), and so on, as shown in the figure below. Applications are then built as a combination of host software (per the host framework) and kernels submitted by the host to run on the VEs with a predefined decoupling point.

### 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 a VE 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 VEs 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 many Vector Engines (VEs), and each VE 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 a kernel is a heuristic driven compiler choice. Common SIMD-width examples are SIMD-8, SIMD-16, and SIMD-32.

For a given SIMD-width, if all kernel instances within a thread are executing the same instruction, 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 VE's branch unit keeps track of such branch divergence and branch nesting.

## Thread Mapping and GPU Occupancy

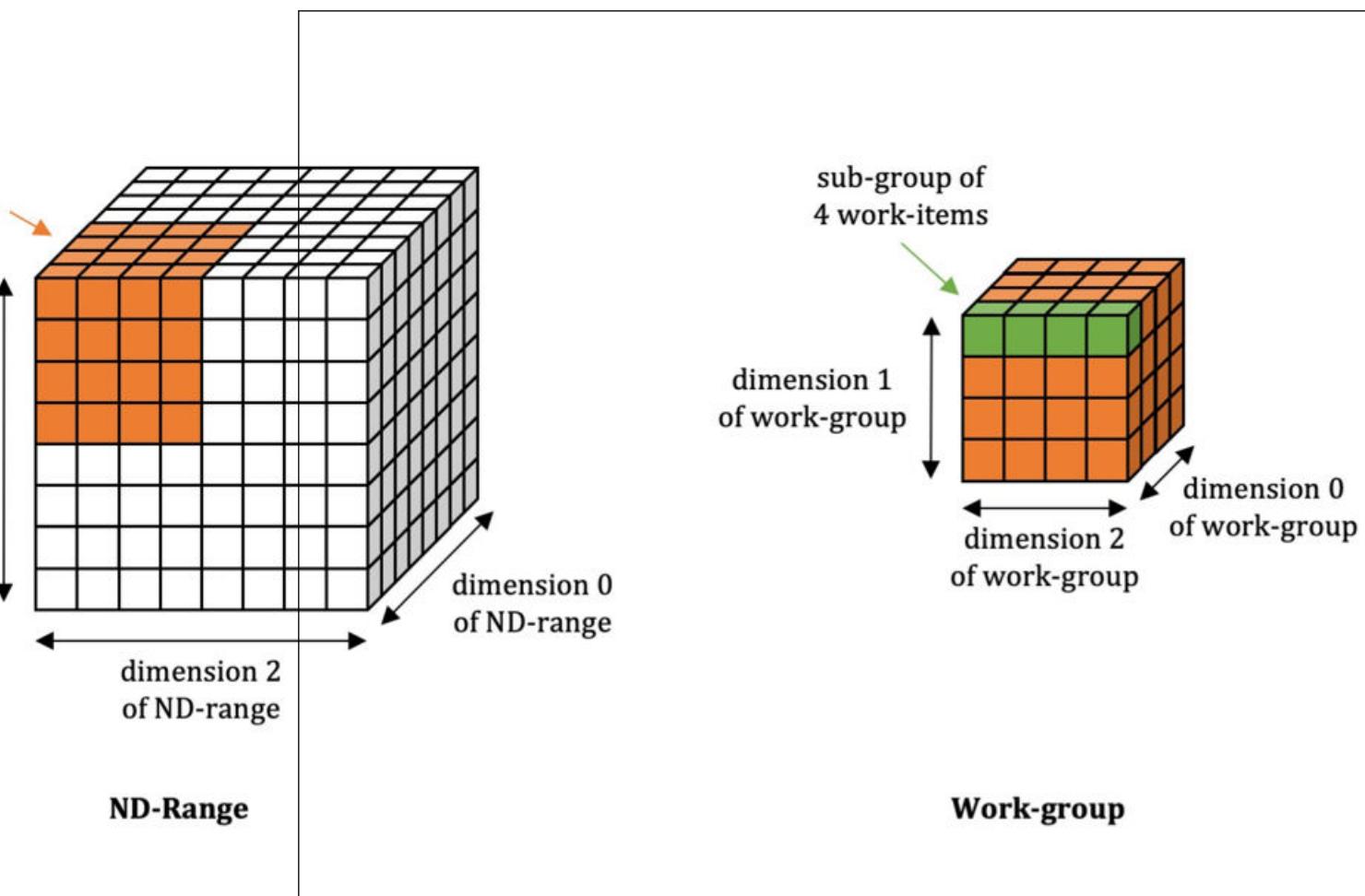
The SYCL execution model exposes an abstract view of GPU execution. The SYCL thread hierarchy consists of a 1-, 2-, or 3-dimensional grid of work-items. These work-items are grouped into equal sized thread groups called work-groups. Threads in a work-group are further divided into equal sized vector groups called sub-groups (see the illustration that follows).

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 SYCL, synchronization across work-items is only possible with barriers for the work-items within the same work-group.

## 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.

### Thread Hierarchy



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

### Thread Synchronization

SYCL provides two synchronization mechanisms that can be called within a kernel function. Both are only defined for work-items within the same work-group. SYCL does not provide any global synchronization mechanism inside a kernel for 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.

### Mapping Work-Groups to X<sup>e</sup>-cores for Maximum Occupancy

The rest of this chapter explains how to pick a proper work-group size to maximize the occupancy of the GPU resources. The example system is the Tiger Lake processors with X<sup>e</sup>-LP GPU as the execution target. The examples also use the new terminologies X<sup>e</sup>-core (XC) for Dual Subslice, and X<sup>e</sup> Vector Engine (XVE) for Execution Unit.

We will use the architecture parameters for X<sup>e</sup>-LP Graphics (TGL) GPU summarized below:

### X<sup>e</sup>-LP (TGL) GPU

	<b>VEs</b>	<b>Threads</b>	<b>Operations</b>	<b>Maximum Work-Group Size</b>
Each X <sup>e</sup> -core	16	$7 \times 16 = 112$	$112 \times 8 = 896$	512
Total	$16 \times 6 = 96$	$112 \times 6 = 672$	$896 \times 6 = 5376$	512

The maximum work-group size is a constraint imposed by the hardware and GPU driver. You can query the maximum work-group size using

`device::get_info<cl::sycl::info::device::max_work_group_size>()` function.

Let's start with a simple kernel:

```
auto command_group =
    [&] (auto &cgh) {
        cgh.parallel_for(sycl::range<3>(64, 64, 64), // global range
                        [=](item<3> it) {
                            // (kernel code)
                        })
    }
```

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 SYCL 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. This chapter focuses on work-group and sub-group size selection.

Thread contexts are easier to utilize than SIMD vector. Therefore, start with selecting the number of threads in a work-group. Each X<sup>e</sup>-core has 112 thread contexts, but usually you 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).

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

- $Work - groupsize = Threads \times Sub - groupSize$

You can increase the sub-group size as long as there are a sufficient number of registers for the kernel after widening. Note that each VE has 128 SIMD8 registers so there is a lot of room for widening on simple kernels. The effect of increasing sub-group size is similar to loop unrolling: while each VE still executes eight 32-bit operations per cycle, the amount of work per work-group interaction is doubled/quadrupled. In SYCL, 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 the selection criteria of threads and sub-group sizes to keep all GPU resources occupied for a Intel® Iris® Xe-LP GPU:

### Configurations to ensure full occupancy

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

In general, choosing a larger work-group size has the advantage of reducing the number of rounds of work-group dispatching. Increasing sub-group size can reduce the number of threads required for a work-group at the expense of longer latency and higher register pressure for each sub-group execution.

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

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

```
auto command_group =
    [&] (auto &cgh) {
        cgh.parallel_for(nd_range(sycl::range(64, 64, 128), // global range
                                  sycl::range(1, R, 128) // local range
                                 ),
                     [=] (sycl::nd_item<3> item) {
                         // (kernel code)
                         // Internal synchronization
                         item.barrier(access::fence_space::global_space);
                         // (kernel code)
                     })
    }
```

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. You must pick a work-group local range using `nd_range` and `nd_item`. All the work-items of a work-group must be allocated to the same Xe-core, which affects Xe-core 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 VE occupancy. In the case of `R=1`, the local group range is `(1, 1, 128)` and work-group size is 128. The Xe-core 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 Xe-core 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, and the kernel will fail to launch. In the case of `R=4`, an Xe-core 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 VE capacities. Note that the driver may still be able to dispatch a partial work-group to an unused Xe-core. However, because of the barrier in the kernel, the partially dispatched 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 between group size, number of threads, Xe-core utilization, and occupancy.

### Utilization for various configurations

Work-items	Group Size	Threads	Xe-core Utilization	Xe-core Occupancy
$64 \times 64 \times 128 = 524288$	$R=1, 128$	16	$16/112 = 14\%$	100%

<b>Work-items</b>	<b>Group Size</b>	<b>Threads</b>	<b>X<sup>e</sup>-core Utilization</b>	<b>X<sup>e</sup>-core Occupancy</b>
$64 \times 64 \times 128 = 524288$	$128 \times 2$	$2 \times 16 = 32$	$32/112 = 28.6\%$	with 7 work-groups 86%
$64 \times 64 \times 128 = 524288$	$128 \times 4$	$3 \times 16 = 48$	$48/112 = 42.9\%$	with 3 work-groups 86%
$64 \times 64 \times 128 = 524288$	$128 \times 4$	$4 \times 16 = 64$	$64/112 = 57\%$	with 2 work-groups maximum 57%
$64 \times 64 \times 128 = 524288$	$640+$			Fail to launch

### Impact of Local Memory Within Work-group

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

```
auto command_group =
    [&] (auto &cgh) {
        // local memory variables shared among work items
        sycl::accessor<int, 1, sycl::access::mode::read_write,
                      sycl::access::target::local>
            myLocal(sycl::range(R), cgh);
        cgh.parallel_for(nd_range(sycl::range<3>(64, 64, 128), // global range
                                  sycl::range<3>(1, R, 128) // local range
                                 ),
                    [=] (ngroup<3> myGroup) {
                        // (work group code)
                        myLocal[myGroup.get_local_id()[1]] = ...
                    })
    }
```

Because work-group local variables are shared among its work-items, they are allocated in a X<sup>e</sup>-core's SLM. Therefore, this work-group must be allocated to a single X<sup>e</sup>-core, same as the intra-group synchronization. In addition, you must also weigh the sizes of local variables under different group size options such that the local variables fit within an X<sup>e</sup>-core's 128KB SLM capacity limit.

### A Detailed Example

Before concluding this section, let's look at the hardware occupancies from the variants of a simple vector add example. Using Intel® Iris® X<sup>e</sup> graphics from TGL platform as the underlying hardware with the resource parameters specified.

```
auto d_selector = sycl::default_selector_v;

// Array type and data size for this example.
constexpr size_t array_size = 3 * 5 * 7 * (1 << 17);
typedef std::array<int, array_size> IntArray;

#define mysize (1 << 17)

int VectorAdd1(sycl::queue &q, const IntArray &a, const IntArray &b,
```

```

        IntArray &sum, int iter) {
sycl::range num_items{a.size()};

sycl::buffer a_buf(a);
sycl::buffer b_buf(b);
sycl::buffer sum_buf(sum.data(), num_items);

auto start = std::chrono::steady_clock::now();
auto e = q.submit([&] (auto &h) {
    // Input accessors
    sycl::accessor a_acc(a_buf, h, sycl::read_only);
    sycl::accessor b_acc(b_buf, h, sycl::read_only);
    // Output accessor
    sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

    h.parallel_for(num_items, [=](auto i) {
        for (int j = 0; j < iter; j++)
            sum_acc[i] = a_acc[i] + b_acc[i];
    });
});
q.wait();
auto end = std::chrono::steady_clock::now();
std::cout << "VectorAdd1 completed on device - took " << (end - start).count()
     << " u-secs\n";
return ((end - start).count());
} // end VectorAdd1

template <int groups>
int VectorAdd2(sycl::queue &q, const IntArray &a, const IntArray &b,
               IntArray &sum, int iter) {
    sycl::range num_items{a.size()};

    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf(sum.data(), num_items);
    size_t num_groups = groups;
    size_t wg_size = 512;
    // get the max wg_size instead of 512 size_t wg_size = 512;
    auto start = std::chrono::steady_clock::now();
    q.submit([&] (auto &h) {
        // Input accessors
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        // Output accessor
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

        h.parallel_for(
            sycl::nd_range<1>(num_groups * wg_size, wg_size),
            [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(32)]] {
                size_t grp_id = index.get_group()[0];
                size_t loc_id = index.get_local_id();
                size_t start = grp_id * mysize;
                size_t end = start + mysize;
                for (int j = 0; j < iter; j++)
                    for (size_t i = start + loc_id; i < end; i += wg_size) {
                        sum_acc[i] = a_acc[i] + b_acc[i];
                    }
            });
    });
}

```

```
};

q.wait();

auto end = std::chrono::steady_clock::now();
std::cout << "VectorAdd2<" << groups << "> completed on device - took "
    << (end - start).count() << " u-secs\n";
return ((end - start).count());
} // end VectorAdd2

void InitializeArray(IntArray &a) {
    for (size_t i = 0; i < a.size(); i++)
        a[i] = i;
}

void Initialize(IntArray &a) {
    for (size_t i = 0; i < a.size(); i++)
        a[i] = 0;
}
IntArray a, b, sum;

int main() {

    sycl::queue q(d_selector);

    InitializeArray(a);
    InitializeArray(b);

    std::cout << "Running on device: "
        << q.get_device().get_info<sycl::info::device::name>() << "\n";
    std::cout << "Vector size: " << a.size() << "\n";

    // check results
    Initialize(sum);
    VectorAdd1(q, a, b, sum, 1);

    for (int i = 0; i < mysize; i++)
        if (sum[i] != 2 * i) {
            std::cout << "add1 Did not match\n";
        }

    Initialize(sum);
    VectorAdd2<1>(q, a, b, sum, 1);
    for (int i = 0; i < mysize; i++)
        if (sum[i] != 2 * i) {
            std::cout << "add2 Did not match\n";
        }

    // time the kernels
    Initialize(sum);
    int t = VectorAdd1(q, a, b, sum, 1000);
    Initialize(sum);
    t = VectorAdd2<1>(q, a, b, sum, 1000);
    t = VectorAdd2<2>(q, a, b, sum, 1000);
    t = VectorAdd2<3>(q, a, b, sum, 1000);
    t = VectorAdd2<4>(q, a, b, sum, 1000);
    t = VectorAdd2<5>(q, a, b, sum, 1000);
    t = VectorAdd2<6>(q, a, b, sum, 1000);
    t = VectorAdd2<7>(q, a, b, sum, 1000);
    t = VectorAdd2<8>(q, a, b, sum, 1000);
```

```

t = VectorAdd2<12>(q, a, b, sum, 1000);
t = VectorAdd2<16>(q, a, b, sum, 1000);
t = VectorAdd2<20>(q, a, b, sum, 1000);
t = VectorAdd2<24>(q, a, b, sum, 1000);
t = VectorAdd2<28>(q, a, b, sum, 1000);
t = VectorAdd2<32>(q, a, b, sum, 1000);
return 0;
} // end of codeblock

```

The `VectorAdd1` section of the program 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.

```

int VectorAdd2(sycl::queue &q, const IntArray &a, const IntArray &b,
               IntArray &sum, int iter) {
    sycl::range num_items{a.size()};
    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf(sum.data(), num_items);
    size_t num_groups = groups;
    size_t wg_size = 512;
    // get the max wg_size instead of 512 size_t wg_size = 512;
    auto start = std::chrono::steady_clock::now();
    q.submit([&](auto &h) {
        // Input accessors
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        // Output accessor
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

        h.parallel_for(
            sycl::nd_range<1>(num_groups * wg_size, wg_size),
            [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(32)]] {
                size_t grp_id = index.get_group()[0];
                size_t loc_id = index.get_local_id();
                size_t start = grp_id * mysize;
                size_t end = start + mysize;
                for (int j = 0; j < iter; j++) {
                    for (size_t i = start + loc_id; i < end; i += wg_size) {
                        sum_acc[i] = a_acc[i] + b_acc[i];
                    }
                });
            });
        q.wait();
        auto end = std::chrono::steady_clock::now();
        std::cout << "VectorAdd2<" << groups << "> completed on device - took "
              << (end - start).count() << " u-secs\n";
        return ((end - start).count());
    }) // end VectorAdd2
}

```

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

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.

**Occupancy**

<b>Program Occupancy</b>	<b>Work-groups</b>	<b>Work-items</b>	<b>Work-group Size</b>	<b>SIMD</b>	<b>Threads Work-group</b>	<b>Threads</b>	<b>Occupancy</b>
VectorAdd 1	53760	27.5M	512	32	16	860K	100%
VectorAdd 2<1>	1	512	512	32	16	16	16/672 = 2.4%
VectorAdd 2<2>	2	1024	512	32	16	32	32/672 = 4.8%
VectorAdd 2<3>	3	1536	512	32	16	48	48/672 = 7.1%
VectorAdd 2<4>	4	2048	512	32	16	64	64/672 = 9.5%
VectorAdd 2<5>	5	2560	512	32	16	80	80/672 = 11.9%
VectorAdd 2<6>	6	3072	512	32	16	96	96/672 = 14.3%
VectorAdd 2<7>	7	3584	512	32	16	112	112/672 = 16.7%
VectorAdd 2<8>	8	4096	512	32	16	128	128/672 = 19%
VectorAdd 2<12>	12	6144	512	32	16	192	192/672 = 28.6%
VectorAdd 2<16>	16	8192	512	32	16	256	256/672 = 38.1%
VectorAdd 2<20>	20	10240	512	32	16	320	320/672 = 47.7%
VectorAdd 2<24>	24	12288	512	32	16	384	384/672 = 57.1%
VectorAdd 2<28>	28	14336	512	32	16	448	448/672 = 66.7%
VectorAdd 2<32>	32	16384	512	32	16	512	512/672 = 76.2%
VectorAdd 2<36>	36	18432	512	32	16	576	576/672 = 85.7%
VectorAdd 2<40>	40	20480	512	32	16	640	640/672 = 95.2%
VectorAdd 2<42>	42	21504	512	32	16	672	672/672 = 100%

<b>Program Occupancy</b>	<b>Work-groups</b>	<b>Work-items</b>	<b>Work-group Size</b>	<b>SIMD</b>	<b>Threads Work-group</b>	<b>Threads</b>	<b>Occupancy</b>
VectorAdd 2<44>	44	22528	512	32	16	704	100% then 4.7%
VectorAdd 2<48>	48	24576	512	32	16	768	100% then 14.3%

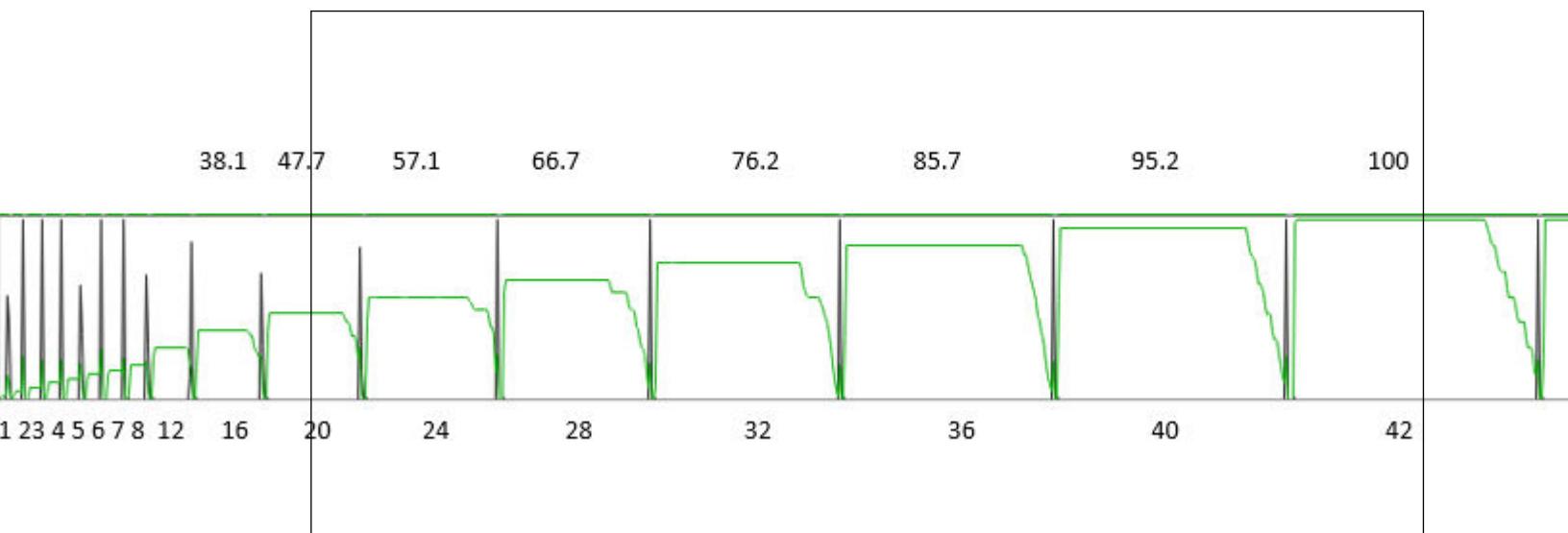
The following VTune analyzer chart for VectorAdd2 with various work-group sizes confirms the accuracy of our estimate. The numbers in the grid view vary slightly from the estimate because the grid view gives an average across the entire execution.

### Occupancy for VectorAdd2 as Shown in VTune

	Work Size		Computing Task					Data Transferred		Act
	Global	Local	Total Time	Average Time	Instance Count	SIMD Width	S...	Size	Total, GB/sec	
			0s							0
ueue&, std::a	512	512	0.046s	0.046s	1	32		0 B	0.000	4
ueue&, std::a	1024	512	0.056s	0.056s	1	32		0 B	0.000	7
ueue&, std::a	1536	512	0.080s	0.080s	1	32		0 B	0.000	7
ueue&, std::a	2048	512	0.087s	0.087s	1	32		0 B	0.000	9
ueue&, std::a	2560	512	0.089s	0.089s	1	32		0 B	0.000	12
ueue&, std::a	3072	512	0.094s	0.094s	1	32		0 B	0.000	13
ueue&, std::a	3584	512	0.101s	0.101s	1	32		0 B	0.000	14
ueue&, std::a	4096	512	0.111s	0.111s	1	32		0 B	0.000	15
queue&, std::	6144	512	0.243s	0.243s	1	32		0 B	0.000	10
queue&, std::	8192	512	0.416s	0.416s	1	32		0 B	0.000	8
queue&, std::	10240	512	0.599s	0.599s	1	32		0 B	0.000	7
queue&, std::	12288	512	0.842s	0.842s	1	32		0 B	0.000	6
queue&, std::	14336	512	0.944s	0.944s	1	32		0 B	0.000	6
queue&, std::	16384	512	1.177s	1.177s	1	32		0 B	0.000	6
queue&, std::	18432	512	1.336s	1.336s	1	32		0 B	0.000	6
queue&, std::	20480	512	1.467s	1.467s	1	32		0 B	0.000	6
queue&, std::	24576	512	1.783s	1.783s	1	32		0 B	0.000	6
queue&, std::	22528	512	1.648s	1.648s	1	32		0 B	0.000	6
queue&, std::	21504	512	1.568s	1.568s	1	32		0 B	0.000	6

The following timeline view gives the occupancy over a period of time. Note that the occupancy metric is accurate for a 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.

### 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.

```
template <int groups, int wg_size, int sg_size>
int VectorAdd3(sycl::queue &q, const IntArray &a, const IntArray &b,
               IntArray &sum, int iter) {
    sycl::range num_items{a.size()};

    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf(sum.data(), num_items);
    size_t num_groups = groups;
    auto start = std::chrono::steady_clock::now();
    q.submit([&](auto &h) {
        // Input accessors
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        // Output accessor
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

        h.parallel_for(
            sycl::nd_range<1>(num_groups * wg_size, wg_size),
            [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(sg_size)]] {
                size_t grp_id = index.get_group()[0];
                size_t loc_id = index.get_local_id();
                size_t start = grp_id * mysize;
                size_t end = start + mysize;
                for (int j = 0; j < iter; j++) {
                    for (size_t i = start + loc_id; i < end; i += wg_size) {
```

```

        sum_acc[i] = a_acc[i] + b_acc[i];
    }
});

q.wait();
auto end = std::chrono::steady_clock::now();
std::cout << "VectorAdd3<" << groups << "> completed on device - took "
       << (end - start).count() << " u-secs\n";
return ((end - start).count());
} // 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 is functionally not needed, but will significantly impact the way in which threads are scheduled on the hardware.

```

template <int groups, int wg_size, int sg_size>
int VectorAdd4(sycl::queue &q, const IntArray &a, const IntArray &b,
               IntArray &sum, int iter) {
    sycl::range num_items{a.size()};
    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf(sum.data(), num_items);
    size_t num_groups = groups;
    auto start = std::chrono::steady_clock::now();
    q.submit([&] (auto &h) {
        // Input accessors
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        // Output accessor
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

        h.parallel_for(
            sycl::nd_range<1>(num_groups * wg_size, wg_size),
            [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(sg_size)]] {
                index.barrier(sycl::access::fence_space::local_space);
                size_t grp_id = index.get_group()[0];
                size_t loc_id = index.get_local_id();
                size_t start = grp_id * mysize;
                size_t end = start + mysize;
                for (int j = 0; j < iter; j++) {
                    for (size_t i = start + loc_id; i < end; i += wg_size) {
                        sum_acc[i] = a_acc[i] + b_acc[i];
                    }
                }
            });
    });
    q.wait();
    auto end = std::chrono::steady_clock::now();
    std::cout << "VectorAdd4<" << groups << "> completed on device - took "
           << (end - start).count() << " u-secs\n";
    return ((end - start).count());
} // end VectorAdd4

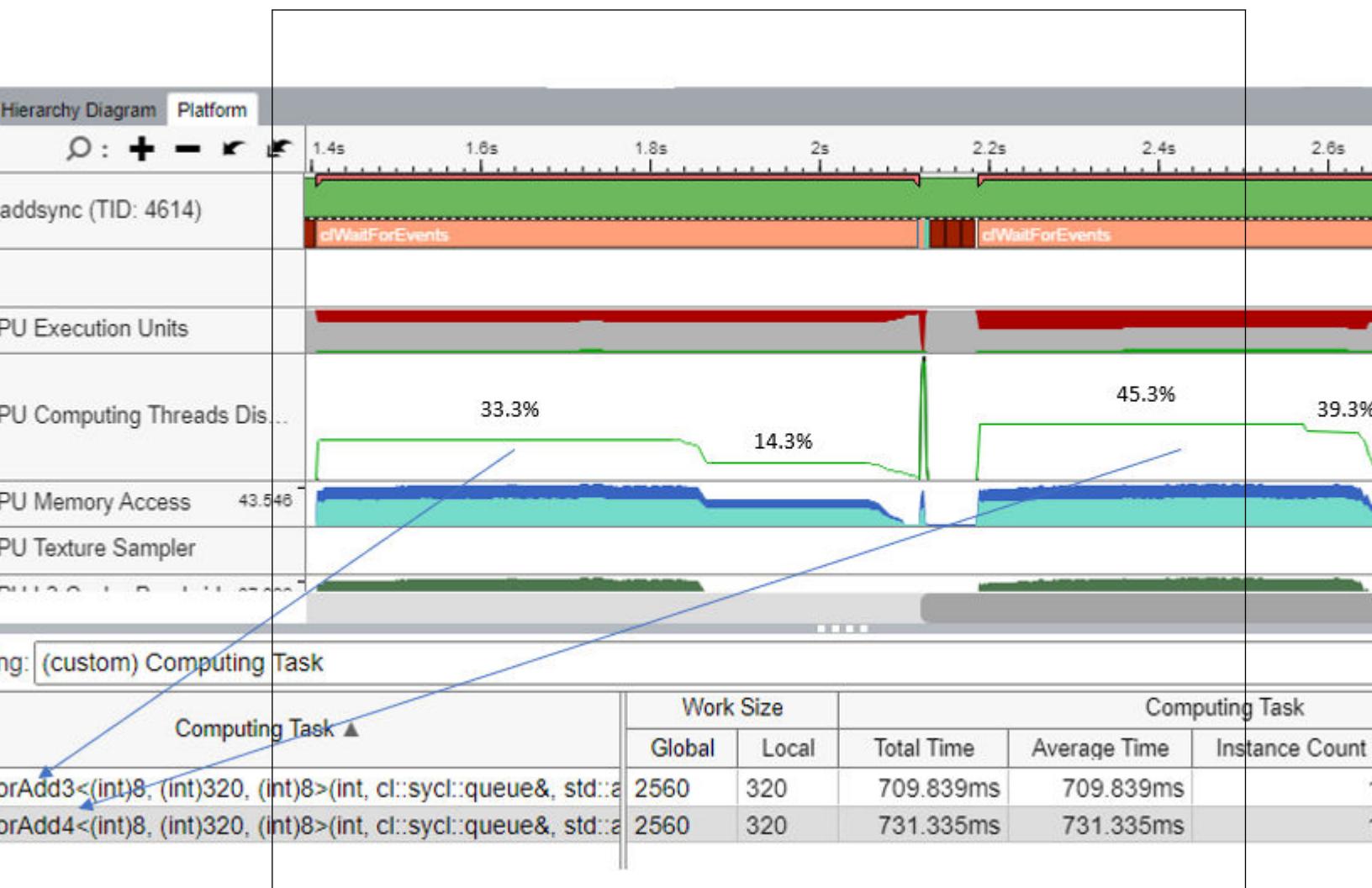
```

To show how 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, 40 threads per work-group need to be scheduled by the hardware.

```
Initialize(sum);
VectorAdd3<8, 320, sgsize>(q, a, b, sum, 10000);
Initialize(sum);
VectorAdd4<8, 320, sgsize>(q, a, b, sum, 10000);
```

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

### GPU Occupancy for `VectorAdd3` and `VectorAdd4` Kernels



For the `VectorAdd3` kernel, there are two phases for occupancies: 33.3% (224 threads occupancy) and 14.3% (96 threads occupancy) on a TGL machine that has a total of 672 threads. Since there are a total of eight work-groups, with each work-group having 40 threads, there are two Xe-cores (each of which have 112 threads) into which the threads of six work-groups are scheduled. This means that 40 threads each from four work-groups are scheduled, and 32 threads each from two other work-groups are scheduled in the first phase. Then in the second phase, 40 threads from the remaining two work-groups are scheduled for execution.

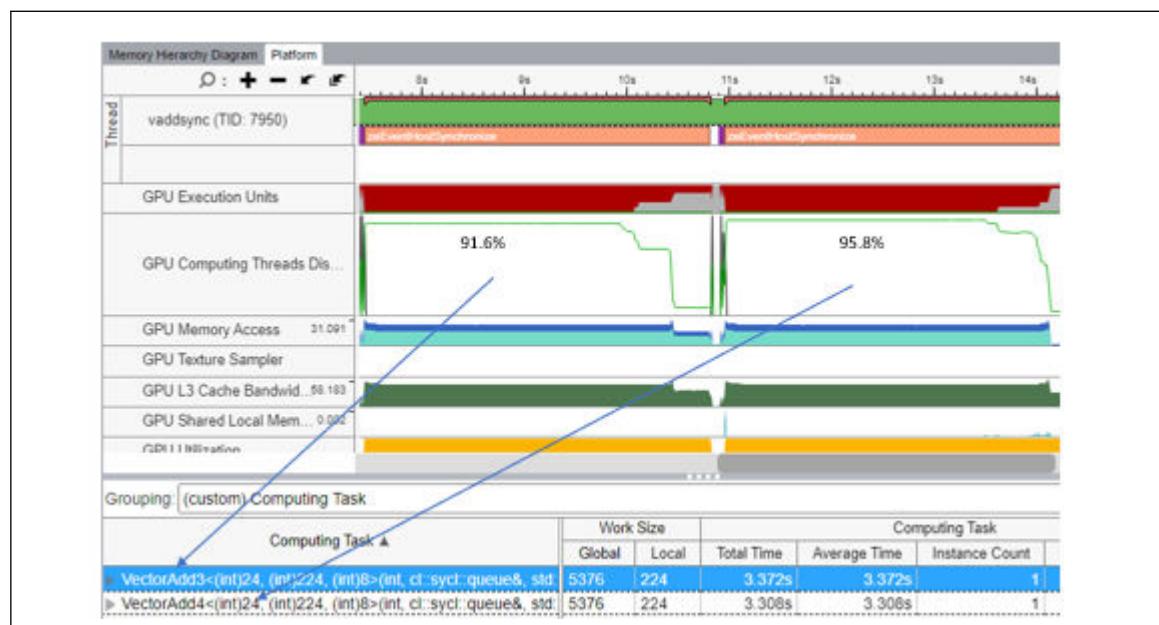
As seen in the `VectorAdd4` kernel, there are three phases of occupancies: 45.3% (304 threads), 39.3% (264 threads), and 11.9% (80 threads). In the first phase, all eight work-groups are scheduled together on 3 X<sup>e</sup>-cores, with two X<sup>e</sup>-cores getting 112 threads each (80 from two work-groups and 32 from one work-group) and one X<sup>e</sup>-core 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 these complete the execution.

The same kernels as above when run with a work-group size that is a multiple of the number of threads in a X<sup>e</sup>-core and lot more work-groups gets good utilization of the hardware achieving close to 100% occupancy, as shown below.

```
Initialize(sum);
VectorAdd3<24, 224, sgsize>(q, a, b, sum, 10000);
Initialize(sum);
VectorAdd4<24, 224, sgsize>(q, a, b, sum, 10000);
```

This kernel execution has a different thread occupancy since we have many more threads and also the work-group size is a multiple of the number of threads in a X<sup>e</sup>-core. This is shown below in the thread occupancy metric on the VTune timeline.

### Thread Occupancy Metric in 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.

You can run different experiments with the above kernels to gain better understanding of how the GPU hardware schedules the software threads on the Execution Units. Be careful about the work-group and sub-group sizes, in addition to a large number of work-groups, to ensure effective utilization of the GPU hardware.

### Intel® GPU Occupancy Calculator

In summary, a SYCL work-group is typically dispatched to an X<sup>e</sup>-core. All the work-items in a work-group share the same SLM of an X<sup>e</sup>-core for intra work-group thread barriers and memory fence synchronization. Multiple work-groups can be dispatched to the same X<sup>e</sup>-core if there are sufficient VE ALUs, SLM, and thread contexts to accommodate them.

You can achieve higher performance by fully utilizing all available X<sup>e</sup>-cores. Parameters affecting a kernel's GPU occupancy are work-group size and SIMD sub-group size, which also determines the number of threads in the work-group.

The [Intel® GPU Occupancy Calculator](#) can be used to calculate the occupancy on an Intel® GPU for a given kernel, and its work-group parameters.

## 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.

- [Sub-Groups and SIMD Vectorization](#)
- [Removing Conditional Checks](#)
- [Registers and Performance](#)
- [Shared Local Memory](#)
- [Pointer Aliasing and the Restrict Directive](#)
- [Synchronization among Threads in a Kernel](#)
- [Considerations for Selecting Work-Group Size](#)
- [Prefetch](#)
- [Reduction](#)
- [Kernel Launch](#)
- [Executing Multiple Kernels on the Device at the Same Time](#)
- [Submitting Kernels to Multiple Queues](#)
- [Avoiding Redundant Queue Constructions](#)
- [Programming Intel® XMX Using SYCL Joint Matrix Extension](#)
- [Doing I/O in the Kernel](#)

## Sub-Groups and SIMD Vectorization

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 vector engines (VE) is implementation-dependent. All the work-groups run concurrently but may be scheduled to run at different times 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 VE 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:

```
q.submit([](auto &h) {
    sycl::stream out(65536, 256, h);
    h.parallel_for(sycl::nd_range(sycl::range{32}, sycl::range{32}),
        [=](sycl::nd_item<1> it) {
            int groupId = it.get_group(0);
            int globalId = it.get_global_linear_id();
            auto sg = it.get_sub_group();
            int sgSize = sg.get_local_range()[0];
            int sgGroupId = sg.get_group_id()[0];
            int sgId = sg.get_local_id()[0];

            out << "globalId = " << sycl::setw(2) << globalId
    });
});
```

```

        << " groupId = " << groupId
        << " sgGroupId = " << sgGroupId << " sgId = " << sgId
        << " sgSize = " << sycl::setw(2) << sgSize
        << sycl::endl;
    });
});

```

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
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 VE 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 the 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.

```

q.submit([&](auto &h) {
    sycl::stream out(65536, 256, h);
    h.parallel_for(sycl::nd_range(sycl::range{32}, sycl::range{32}),
                  [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(32)]] {
                    int groupId = it.get_group(0);
                    int globalId = it.get_global_linear_id();
                    auto sg = it.get_sub_group();
                    int sgSize = sg.get_local_range()[0];
                    int sgGroupId = sg.get_group_id()[0];

```

```

        int sgId = sg.get_local_id()[0];

        out << "globalId = " << sycl::setw(2) << globalId
            << " groupId = " << groupId
            << " sgGroupId = " << sgGroupId << " sgId = " << sgId
            << " sgSize = " << sycl::setw(2) << sgSize
            << sycl::endl;
    });
}
);

```

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
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:

```

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

```

The valid sub-group sizes supported may be:

```
Subgroup Sizes: 8 16 32
```

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

## Vectorization and Memory Access

The Intel® graphics device has multiple VEs. Each VE is a multithreaded SIMD processor. The compiler generates SIMD instructions to pack multiple work-items in a sub-group to execute simultaneously in a VE thread. The SIMD width (thus the sub-group size), selected by the compiler is based on device characteristics and heuristics, or requested explicitly by the kernel, and 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 the dynamic instruction count. SIMD divergence negatively impacts performance. The compiler works to minimize divergence, but 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:

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

auto e = q.submit([&](auto &h) {
    h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}),
                  [=](sycl::nd_item<1> it) {
                      int i = it.get_global_linear_id();
                      i = i * 16;
                      for (int j = i; j < (i + 16); j++) {
                          data[j] = -1;
                      }
                  });
    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.

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

auto e = q.submit([&](auto &h) {
    h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}),
                  [=](sycl::nd_item<1> it) {
                      int i = it.get_global_linear_id();
                      auto sg = it.get_sub_group();
                      int sgSize = sg.get_local_range()[0];
                      i = (i / sgSize) * sgSize * 16 + (i % sgSize);
                      for (int j = 0; j < sgSize * 16; j += sgSize) {
                          data[i + j] = -1;
                      }
                  });
    q.wait();
});
```

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

```
constexpr int N = 1024 * 1024;
int *data = sycl::malloc_shared<int>(N, q);
int *data2 = sycl::malloc_shared<int>(N, q);
memset(data2, 0xFF, sizeof(int) * N);

auto e = q.submit([&](auto &h) {
```

```

    h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}),
                  [=](sycl::nd_item<1> it) {
                      int i = it.get_global_linear_id();
                      i = i * 16;
                      for (int j = i; j < (i + 16); j++) {
                          data[j] = data2[j];
                      }
                  });
  );
}
);

```

This kernel copies an array of  $1024 \times 1024$  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 to change the code to read and store contiguous integers in each sub-group instead of each work-item.

```

constexpr int N = 1024 * 1024;
int *data = sycl::malloc_shared<int>(N, q);
int *data2 = sycl::malloc_shared<int>(N, q);
memset(data2, 0xFF, sizeof(int) * N);

auto e = q.submit([&](auto &h) {
    h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}),
                  [=](sycl::nd_item<1> it) {
                      int i = it.get_global_linear_id();
                      auto sg = it.get_sub_group();
                      int sgSize = sg.get_local_range()[0];
                      i = (i / sgSize) * sgSize * 16 + (i % sgSize);
                      for (int j = 0; j < sgSize * 16; j += sgSize) {
                          data[i + j] = data2[i + j];
                      }
                  });
  );
});

```

## Maximizing Memory Bandwidth Utilization

Each work item in the above example loads and stores 1 integer or 4 bytes in every loop iteration( $\text{data}[i + j] = \text{data2}[i + j]$ ), or each vectorized memory operation loads/stores 64 bytes(assuming sub-group size 16), leaving the memory bandwidth unsaturated.

Increasing the payload or the size of data each work item loads or stores in one memory operation will result in better bandwidth utilization.

```

constexpr int N = 1024 * 1024;
int *data = sycl::malloc_shared<int>(N, q);
int *data2 = sycl::malloc_shared<int>(N, q);
memset(data2, 0xFF, sizeof(int) * N);

auto e = q.submit([&](auto &h) {
    h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}),
                  [=](sycl::nd_item<1> it) {
                      int i = it.get_global_linear_id();
                      auto sg = it.get_sub_group();
                      int sgSize = sg.get_local_range()[0];
                      i = (i / sgSize) * sgSize * 16 + (i % sgSize) * 4;
                      for (int j = 0; j < 4; j++) {
                          sycl::vec<int, 4> x;
                          sycl::vec<int, 4> *q =
                              (sycl::vec<int, 4> *)(&(data2[i + j * sgSize * 4]));
                          x = *q;
                      }
                  });
  );
});

```

```

        sycl::vec<int, 4> *r =
            (sycl::vec<int, 4> *)(&(data[i + j * sgSize * 4]));
        *r = x;
    }
});
}
);

```

Each work item loads/stores a `sycl::vec<int, 4>` instead of an integer in every loop iteration. Reading/writing 256 contiguous bytes (assuming sub-group size 16) of memory, the payload of every vectorized memory operation is quadrupled.

The maximum bandwidth is different from hardware to hardware. One can use Intel® VTune Profiler to measure the bandwidth and find the optimal size.

Using a vector type, however, can potentially increase register pressure. It is advised to use long vectors only if registers do not spill. Please refer to the chapter “Registerization and Avoiding Register Spills” for techniques for avoiding register spills.

## Memory Block Load and Store

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

```

constexpr int N = 1024 * 1024;
int *data = sycl::malloc_shared<int>(N, q);
int *data2 = sycl::malloc_shared<int>(N, q);
memset(data2, 0xFF, sizeof(int) * N);

auto e = q.submit([&] (auto &h) {
    h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}),
                  [=] (sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
                      auto sg = it.get_sub_group();
                      sycl::vec<int, 8> x;

                      int base =
                          (it.get_group(0) * 32 +
                           sg.get_group_id()[0] * sg.get_local_range()[0]) *
                           16;

                      x = sg.load<8>(get_multi_ptr(&(data2[base + 0])));
                      sg.store<8>(get_multi_ptr(&(data[base + 0])), x);
                      x = sg.load<8>(get_multi_ptr(&(data2[base + 128])));
                      sg.store<8>(get_multi_ptr(&(data[base + 128])), x);
                  });
    });

```

This example also uses `sycl::vec` for performance, the integers in `x`, however, are not contiguous in memory, but with a stride of the sub-group size!

You may have noticed that the sub-group size 16 was explicitly requested. 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.

## 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.

```

constexpr size_t BLOCK_SIZE = 16;
sycl::buffer<uint, 2> m(matrix.data(), sycl::range<2>(N, N));

auto e = q.submit([&](auto &h) {
    sycl::accessor marr(m, h);
    sycl::local_accessor<uint, 2> barr1(
        sycl::range<2>(BLOCK_SIZE, BLOCK_SIZE), h);
    sycl::local_accessor<uint, 2> barr2(
        sycl::range<2>(BLOCK_SIZE, BLOCK_SIZE), h);

    h.parallel_for(
        sycl::nd_range<2>(sycl::range<2>(N / BLOCK_SIZE, N),
                           sycl::range<2>(1, BLOCK_SIZE)),
        [=](sycl::nd_item<2> it) [[intel::reqd_sub_group_size(16)]] {
            int gi = it.get_group(0);
            int gj = it.get_group(1);

            auto sg = it.get_sub_group();
            uint sgId = sg.get_local_id()[0];

            uint bcol[BLOCK_SIZE];
            int ai = BLOCK_SIZE * gi;
            int aj = BLOCK_SIZE * gj;

            for (uint k = 0; k < BLOCK_SIZE; k++) {
                bcol[k] = sg.load(marr.get_pointer() + (ai + k) * N + aj);
            }

            uint tcol[BLOCK_SIZE];
            for (uint n = 0; n < BLOCK_SIZE; n++) {
                if (sgId == n) {
                    for (uint k = 0; k < BLOCK_SIZE; k++) {
                        tcol[k] = sg.shuffle(bcol[n], k);
                    }
                }
            }

            for (uint k = 0; k < BLOCK_SIZE; k++) {
                sg.store(marr.get_pointer() + (ai + k) * N + aj, tcol[k]);
            }
        });
    });
});

```

This kernel transposes a  $16 \times 16$  matrix. It looks more complicated than the previous examples, but the idea is simple: a sub-group loads a  $16 \times 16$  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  $4096 \times 4096$ , 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. This is left to the reader as an exercise.

SYCL has multiple variants of sub-group shuffle functions available. Each variant is optimized for its specific purpose on specific devices. It is always a good idea to use these optimized functions (if they fit your needs) instead of creating your own.

## 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:

```
auto e = q.submit([&] (auto &h) {
    sycl::stream out(65536, 128, h);
    h.parallel_for(sycl::nd_range<1>(7, 7),
                  [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
                    int i = it.get_global_linear_id();
                    auto sg = it.get_sub_group();
                    int sgSize = sg.get_local_range()[0];
                    int sgMaxSize = sg.get_max_local_range()[0];
                    int sId = sg.get_local_id()[0];
                    int j = data[i];
                    int k = data[i + sgSize];
                    out << "globalId = " << i << " sgMaxSize = " << sgMaxSize
                        << " sgSize = " << sgSize << " sId = " << sId
                        << " j = " << j << " k = " << k << sycl::endl;
                });
});
q.wait();
```

The output of this example looks like this:

```
globalId = 0 sgMaxSize = 16 sgSize = 7 sId = 0 j = 0 k = 7
globalId = 1 sgMaxSize = 16 sgSize = 7 sId = 1 j = 1 k = 8
globalId = 2 sgMaxSize = 16 sgSize = 7 sId = 2 j = 2 k = 9
globalId = 3 sgMaxSize = 16 sgSize = 7 sId = 3 j = 3 k = 10
globalId = 4 sgMaxSize = 16 sgSize = 7 sId = 4 j = 4 k = 11
globalId = 5 sgMaxSize = 16 sgSize = 7 sId = 5 j = 5 k = 12
globalId = 6 sgMaxSize = 16 sgSize = 7 sId = 6 j = 6 k = 13
```

The sub-group size is seven, though the maximum sub-group size is still 16! 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.

## Removing Conditional Checks

In [Sub-Groups and SIMD Vectorization](#), 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.

## Padding Buffers to Remove Conditional Checks

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

```

sycl::buffer<int> ibuf(input.data(), N);
sycl::buffer<int> obuf(output.data(), N);
sycl::buffer<int> kbuf(kernel.data(), M);

auto e = q.submit([&] (auto &h) {
    sycl::accessor iacc(ibuf, h, sycl::read_only);
    sycl::accessor oacc(obuf, h);
    sycl::accessor kacc(kbuf, h, sycl::read_only);

    h.parallel_for(sycl::nd_range<1>(sycl::range{N}, sycl::range{256}),
                  [=](sycl::nd_item<1> it) {
                      int i = it.get_global_linear_id();
                      int group = it.get_group()[0];
                      int gSize = it.get_local_range()[0];

                      int t = 0;
                      int _M = static_cast<int>(M);
                      int _N = static_cast<int>(N);

                      if ((group == 0) || (group == _N / gSize - 1)) {
                          if (i < _M / 2) {
                              for (int j = _M / 2 - i, k = 0; j < _M; ++j, ++k) {
                                  t += iacc[k] * kacc[j];
                              }
                          } else {
                              if (i + _M / 2 >= _N) {
                                  for (int j = 0, k = i - _M / 2;
                                       j < _M / 2 + _N - i; ++j, ++k) {
                                      t += iacc[k] * kacc[j];
                                  }
                              } else {
                                  for (int j = 0, k = i - _M / 2; j < _M; ++j, ++k) {
                                      t += iacc[k] * kacc[j];
                                  }
                              }
                          }
                      } else {
                          for (int j = 0, k = i - _M / 2; j < _M; ++j, ++k) {
                              t += iacc[k] * kacc[j];
                          }
                      }

                      oacc[i] = t;
                  });
});
```

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 bounds. If we pad enough 0s before and after the input array, these conditional checks can be safely removed:

```

std::vector<int> input(N + M / 2 + M / 2);
std::vector<int> output(N);
std::vector<int> kernel(M);

srand(2009);
for (size_t i = M / 2; i < N + M / 2; ++i) {
```

```

    input[i] = rand();
}

for (size_t i = 0; i < M / 2; ++i) {
    input[i] = 0;
    input[i + N + M / 2] = 0;
}

for (size_t i = 0; i < M; ++i) {
    kernel[i] = rand();
}

{
    sycl::buffer<int> ibuf(input.data(), N + M / 2 + M / 2);
    sycl::buffer<int> obuf(output.data(), N);
    sycl::buffer<int> kbuf(kernel.data(), M);

    auto e = q.submit([&](auto &h) {
        sycl::accessor iacc(ibuf, h, sycl::read_only);
        sycl::accessor oacc(obuf, h);
        sycl::accessor kacc(kbuf, h, sycl::read_only);

        h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{256}),
                      [=](sycl::nd_item<1> it) {
                          int i = it.get_global_linear_id();
                          int t = 0;

                          for (size_t j = 0; j < M; ++j) {
                              t += iacc[i + j] * kacc[j];
                          }

                          oacc[i] = t;
                      });
    });
    q.wait();

    size_t kernel_ns = (e.template get_profiling_info<
                            sycl::info::event_profiling::command_end>() -
                        e.template get_profiling_info<
                            sycl::info::event_profiling::command_start>());
    std::cout << "Kernel Execution Time Average: total = " << kernel_ns * 1e-6
           << " msec" << std::endl;
}

```

## 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 a built-in function if one is available. SYCL provides a rich set of built-in relational functions like `select()`, `min()`, `max()`. In many cases you can use these functions 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()`.

```

    sycl::buffer<int> ibuf(input.data(), N);
    sycl::buffer<int> obuf(output.data(), N);
    sycl::buffer<int> kbuf(kernel.data(), M);

```

```

auto e = q.submit([&](auto &h) {
    sycl::accessor iacc(ibuf, h, sycl::read_only);
    sycl::accessor oacc(obuf, h);
    sycl::accessor kacc(kbuf, h, sycl::read_only);

    h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{256}),
                   [=](sycl::nd_item<1> it) {
                       int i = it.get_global_linear_id();
                       int t = 0;
                       int startj = sycl::max<int>(M / 2 - i, 0);
                       int endj = sycl::min<int>(M / 2 + N - i, M);
                       int startk = sycl::max<int>(i - M / 2, 0);
                       for (int j = startj, k = startk; j < endj; j++, k++) {
                           t += iacc[k] * kacc[j];
                       }
                       oacc[i] = t;
                   });
});

```

## Registers and Performance

The register is the fastest storage in the memory hierarchy. Keeping data in registers as long as possible is critical to performance. But unfortunately, register space is limited and much smaller than memory space. The Intel® Data Center GPU Max Series product, for example, has 64KB general-purpose register file (GRF) space for each vector engine, or 128 general-purpose registers, each 64 bytes wide, for each XVE thread in small register mode.

Thus, the register space can be allocated only to a small set of variables at any point during execution. Fortunately, A given register can hold different variables at different times because different sets of variables are needed at different times.

In SYCL, the compiler allocates registers to private variables in work items. Multiple work items in a sub-group are packed into one XVE thread. The compiler aims to assign as many variables to registers as possible. 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 16 instead of 32) if a sub-group size is not explicitly requested. It can also cause register spilling, i.e., moving some variables currently in registers to memory to make room for other variables, or cause 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. Memory traffic can be increased if register spills or accesses to not-promoted-to-register variables occur inside hot loops. In both cases, performance can be significantly degraded.

Though the compiler uses intelligent algorithms to allocate variables in registers and to minimize register spills, optimizations by developers can help the compiler to do a better job and often make a big performance difference.

- [Finding Kernels with Register Spills](#)
- [Small Register Mode vs. Large Register Mode](#)
- [Optimizing Register Spills](#)
- [Porting Code with High Register Pressure to Intel® Max GPUs](#)

### Finding Kernels with Register Spills

The compiler outputs a warning if a kernel is compiled ahead of time and has register spills:

```
$ icpx -fsyycl -fsyycl-targets=spir64_gen -Xsyycl-target-backend "-device pvc"
app.cpp
```

```
Compilation from IR - skipping loading of FCL
warning: kernel _ZTSZ4mainEULT_E0_ compiled SIMD32 allocated 128 regs and
spilled around 396
Build succeeded.
```

However, the compiler does not report if a kernel has one or more private variables that are not prompted to registers.

The open source tool [unitrace](#) reports both kernels with register spills and kernels with not-prompted-to-register private variable(s). Plus, it works for both ahead-of-time compilation and just-in-time compilation:

```
$ icpx -fsycl app.cpp
$ unitrace -d ./a.out
...
== L0 Backend ==

Kernel, Calls,      Time(ns), Time(%), Average(ns),   Min(ns),   Max(ns)
main::{lambda(auto:_1) #4},    100, 91349596800, 99.896, 913495968, 913304160, 913975360
main::{lambda(auto:_1) #3},    100, 77196960, 0.084, 771969, 2080, 76974560
main::{lambda(auto:_1) #5},    500, 4981120, 0.005, 9962, 1440, 42880
main::{lambda(auto:_1) #7},    500, 4600000, 0.005, 9200, 1440, 38720
main::{lambda(auto:_1) #6},    500, 4590880, 0.005, 9181, 1440, 39040
main::{lambda(auto:_1) #1},    100, 3494400, 0.003, 34944, 1760, 3305120
main::{lambda(auto:_1) #2},    100, 189280, 0.000, 1892, 1440, 31360

== Kernel Properties ==

Kernel, Private Memory Per Thread, Spill Memory Per Thread
main::{lambda(auto:_1) #4},           16384, 0
main::{lambda(auto:_1) #3},           0, 8192
main::{lambda(auto:_1) #5},           0, 0
main::{lambda(auto:_1) #7},           0, 0
main::{lambda(auto:_1) #6},           0, 0
main::{lambda(auto:_1) #1},           0, 0
main::{lambda(auto:_1) #2},           0, 0
```

A non-zero value in bytes of the Spill Memory Per Thread indicates the kernel spills registers and a non-zero value in bytes of the Private Memory Per Thread indicates the kernel has at least one private variable that is not prompted to registers.

The tool also reports timing statistics for kernels executed on the device. These statistics can be helpful to developers to evaluate the performance impact of register spills and to prioritize the kernels to be optimized.

## Small Register Mode vs Large Register Mode

Intel® Data Center GPU Max Series products support two GRF modes: small GRF mode and large GRF mode. Each XVE has a total of 64 KB of register space. In Small GRF mode, a single hardware thread can access 128 GRF registers, each of which is 64B wide. In this mode, 8 hardware threads are available per XVE. In Large GRF mode, a single hardware thread can access 256 GRF registers, each of which is 64B wide. In this mode, 4 hardware threads are available per XVE.

There are two ways to control how Intel® Graphics Compiler (IGC) selects between these two modes: (1) command line and (2) per-kernel specification. In this chapter, we provide a step-by-step guideline on how users can provide this control for both SYCL and OpenMP applications.

## GRF Mode Specification at Command Line

The `-ftarget-register-alloc-mode=<arg>` compiler option provides the ability to guide GRF mode selection in the IGC graphics compiler. The format of `<arg>` is `Device0:Mode0[,Device1:Mode1...]`. Currently the only supported Device is `pvc`. The supported modes are:

- `default` Provide no specification to IGC on the register file mode to select. Currently, IGC always chooses small register file mode with no specification.
- `small` Forces IGC to select small register file mode for ALL kernels
- `large` Forces IGC to select large register file mode for ALL kernels
- `auto` Enables IGC to select small/large GRF mode on a per-kernel basis based on heuristics

If this option is not specified, IGC selects a GRF mode on a per-kernel basis based on heuristics on Linux for the Intel® Data Center GPU Max Series and small GRF mode otherwise.

### OpenMP - GRF Mode Selection (AOT)

Following are the various commands that can be used to specify the requisite backend option during AOT compilation for OpenMP backends. Here, `test.cpp` can be any valid program:

```
icpx -fiopenmp -fopenmp-targets=spir64_gen
-ftarget-register-alloc-mode=pvc:large
-Xopenmp-target-backend "-device pvc" test.cpp
// IGC will force large GRF mode for all kernels
```

```
icpx -fiopenmp -fopenmp-targets=spir64_gen
-ftarget-register-alloc-mode=pvc:auto
-Xopenmp-target-backend "-device pvc" test.cpp
// IGC will use compiler heuristics to pick between small and large GRF
mode on a per-kernel basis
```

```
icpx -fiopenmp -fopenmp-targets=spir64_gen
-ftarget-register-alloc-mode=pvc:small
-Xopenmp-target-backend "-device pvc" test.cpp
// IGC will automatically use small GRF mode for all kernels
```

### OpenMP - GRF Mode Selection (JIT)

Following are the various commands that can be used to specify the requisite backend option during JIT compilation for OpenMP backends. Here, `test.cpp` can be any valid program:

```
icpx -fiopenmp -fopenmp-targets=spir64
-ftarget-register-alloc-mode=pvc:large
test.cpp
// IGC will force large GRF mode for all kernels
```

```
icpx -fiopenmp -fopenmp-targets=spir64
-ftarget-register-alloc-mode=pvc:auto
test.cpp
// IGC will use compiler heuristics to pick between small and large GRF
mode on a per-kernel basis
```

```
icpx -fiopenmp -fopenmp-targets=spir64
-ftarget-register-alloc-mode=pvc:small
test.cpp
// IGC will automatically use small GRF mode for all kernels
```

## SYCL – GRF Mode Selection (AOT)

Following are the various commands that can be used to specify the requisite backend option during AOT compilation for SYCL backends. Here, `test.cpp` can be any valid SYCL program:

```
icpx -fsycl -fsycl-targets=spir64_gen
-ftarget-register-alloc-mode=pvc:large
-Xsycl-target-backend "-device pvc" test.cpp
// IGC will force large GRF mode for all kernels
```

```
icpx -fsycl -fsycl-targets=spir64_gen
-ftarget-register-alloc-mode=pvc:auto
-Xsycl-target-backend "-device pvc" test.cpp
// IGC will use compiler heuristics to pick between small and large GRF
mode on a per-kernel basis
```

```
icpx -fsycl -fsycl-targets=spir64_gen
-ftarget-register-alloc-mode=pvc:small
-Xsycl-target-backend "-device pvc" test.cpp
// IGC will automatically use small GRF mode for all kernels
```

## SYCL – GRF Mode Selection (JIT)

Following are the various commands that can be used to specify the requisite backend option during JIT compilation for SYCL backends. Here, `test.cpp` can be any valid SYCL program:

```
icpx -fsycl
-ftarget-register-alloc-mode=pvc:large
test.cpp
// IGC will force large GRF mode for all kernels
```

```
icpx -fsycl
-ftarget-register-alloc-mode=pvc:auto
test.cpp
// IGC will use compiler heuristics to pick between small and large GRF
mode on a per-kernel basis
```

```
icpx -fsycl
-ftarget-register-alloc-mode=pvc:small
test.cpp
// IGC will automatically use small GRF mode for all kernels
```

## Performance Tuning Using GRF Mode Selection

This section discusses the impact of GRF mode selection on device code performance. The examples shown in this section use the OpenMP offloading model and JIT compilation flow. Two of the main features that govern GRF mode selection are the following: (1) Register pressure for kernel code (2) Number of parallel execution threads. Following is a code snippet containing an OpenMP offload region and this will be used in the forthcoming analysis.

```
#pragma omp target teams distribute thread_limit(ZDIM *NX1 *NX1)
for (int e = 0; e < nelt; e++) {
    double s_u[NX1 * NX1 * NX1];
    double s_D[NX1 * NX1];
    // SLM used for the three arrays here
    double s_ur[NX1 * NX1 * NX1];
    double s_us[NX1 * NX1 * NX1];
    double s_ut[NX1 * NX1 * NX1];

#pragma omp parallel for
    for (int inner = 0; inner < innerub; inner++) {
```

```

int k = inner / (NX1 * NX1);
int j = (inner - k * NX1 * NX1) / NX1;
int i = inner - k * NX1 * NX1 - j * NX1;
if (k == 0)
    s_D[I2(i, j)] = D[I2(i, j)];
for (; k < NX1; k += ZDIM) {
    s_u[I3(i, j, k)] = u[I4(i, j, k, e)];
}
}

#pragma omp parallel for
for (int inner = 0; inner < innerub; inner++) {
    int k = inner / (NX1 * NX1);
    int j = (inner - k * NX1 * NX1) / NX1;
    int i = inner - k * NX1 * NX1 - j * NX1;

    double r_G00, r_G01, r_G02, r_G11, r_G12, r_G22;

    for (; k < NX1; k += ZDIM) {
        double r_ur, r_us, r_ut;
        r_ur = r_us = r_ut = 0;
#define FORCE_UNROLL
#pragma unroll NX1
#endif
        for (int m = 0; m < NX1; m++) {
            r_ur += s_D[I2(i, m)] * s_u[I3(m, j, k)];
            r_us += s_D[I2(j, m)] * s_u[I3(i, m, k)];
            r_ut += s_D[I2(k, m)] * s_u[I3(i, j, m)];
        }

        const unsigned gbase = 6 * I4(i, j, k, e);
        r_G00 = g[gbase + 0];
        r_G01 = g[gbase + 1];
        r_G02 = g[gbase + 2];
        s_ur[I3(i, j, k)] = r_G00 * r_ur + r_G01 * r_us + r_G02 * r_ut;
        r_G11 = g[gbase + 3];
        r_G12 = g[gbase + 4];
        s_us[I3(i, j, k)] = r_G01 * r_ur + r_G11 * r_us + r_G12 * r_ut;
        r_G22 = g[gbase + 5];
        s_ut[I3(i, j, k)] = r_G02 * r_ur + r_G12 * r_us + r_G22 * r_ut;
    }
}

#pragma omp parallel for
for (int inner = 0; inner < innerub; inner++) {
    int k = inner / (NX1 * NX1);
    int j = (inner - k * NX1 * NX1) / NX1;
    int i = inner - k * NX1 * NX1 - j * NX1;
    for (; k < NX1; k += ZDIM) {
        double wr = 0.0;
        for (int m = 0; m < NX1; m++) {
            double s_D_i = s_D[I2(m, i)];
            double s_D_j = s_D[I2(m, j)];
            double s_D_k = s_D[I2(m, k)];
            wr += s_D_i * s_ur[I3(m, j, k)] + s_D_j * s_us[I3(i, m, k)] +
                  s_D_k * s_ut[I3(i, j, m)];
        }
        w[I4(i, j, k, e)] = wr;
    }
}

```

```

    }
}
}
```

There are two parameters that can be modified here: (1) Unroll factor of inner loop in line number 36 (2) Number of OpenMP teams specified in line number 1. The unroll factor can be used to control register pressure. Greater the unroll factor, higher will be the register pressure. Number of OpenMP teams can be used to control the number of parallel threads. In this discussion, kernel execution time on the device is used as metric for performance. Actual numbers are not provided as they may vary based on user environments and device settings. Following are some observations:

When unrolling is turned off, use of small GRF mode is found to provide better performance. This implies that the register pressure is not high enough to get any benefits out of using large GRF mode.

When unrolling is turned on, use of large GRF mode is found to provide better performance. This implies that the register pressure is high and large GRF mode is required to accommodate this pressure.

Increase in number of teams tends to result in better performance for larger (higher register pressure) workloads when using small GRF mode.

## Optimizing Register Spills

The following techniques can reduce register pressure:

- Keep live ranges of private variables as short as possible.

Though the compiler schedules instructions and optimizes the distances, in some cases moving the loading and using the same variable closer or removing certain dependencies in the source can help the compiler 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 unrolling and different times of unrolls to decide if a loop should be unrolled or how many times to unroll it.

- Prefer USM pointers.

A buffer accessor takes more space than a USM pointer. If you can choose between USM pointers and buffer accessors, choose USM pointers.

- Recompute cheap-to-compute values on-demand that otherwise would be held in registers for a long time.
- Avoid big arrays or large structures, or break an array of big structures into multiple arrays of small structures.

For example, an array of `sycl::float4`:

```
``sycl::float4 v[8];``
```

can be broken into 4 arrays of `float`:

```
``float x[8];
float y[8];
float z[8];
float w[8];``
```

All or part of the 4 arrays of `float` have a better chance to be allocated in registers than the array of `sycl::float4`.

- Break a large loop into multiple small loops to reduce the number of simultaneously live variables.
- Choose smaller sized data types if possible.

- Do not declare private variables as volatile.
- Do not take address of a private variable and later dereference the pointer
- Share registers in a sub-group.
- Use sub-group block load/store if possible.
- Use shared local memory.

The list here is not exhaustive.

The rest of this chapter shows how to apply these techniques, especially the last five, in real examples.

## Choosing Smaller Data Types

```
constexpr int BLOCK_SIZE = 256;
constexpr int NUM_BINS = 32;

std::vector<unsigned long> hist(NUM_BINS, 0);
sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);

auto e = q.submit([&] (auto &h) {
    sycl::accessor macc(mbuf, h, sycl::read_only);
    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
    h.parallel_for(
        sycl::nd_range(sycl::range{N / BLOCK_SIZE}, sycl::range{64}),
        [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
            int group = it.get_group()[0];
            int gSize = it.get_local_range()[0];
            auto sg = it.get_sub_group();
            int sgSize = sg.get_local_range()[0];
            int sgGroup = sg.get_group_id()[0];

            unsigned long
                histogram[NUM_BINS]; // histogram bins take too much storage to be
                // promoted to registers
            for (int k = 0; k < NUM_BINS; k++) {
                histogram[k] = 0;
            }
            for (int k = 0; k < BLOCK_SIZE; k++) {
                unsigned long x =
                    sg.load(macc.get_pointer() + group * gSize * BLOCK_SIZE +
                            sgGroup * sgSize * BLOCK_SIZE + sgSize * k);
                #pragma unroll
                for (int i = 0; i < 8; i++) {
                    unsigned int c = x & 0x1FU;
                    histogram[c] += 1;
                    x = x >> 8;
                }
            }
            for (int k = 0; k < NUM_BINS; k++) {
                hacc[k].fetch_add(histogram[k]);
            }
        });
});
```

This example calculates histograms with a 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 in poor performance.

With `BLOCK_SIZE` 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.

```
constexpr int BLOCK_SIZE = 256;
constexpr int NUM_BINS = 32;

std::vector<unsigned long> hist(NUM_BINS, 0);

sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);

auto e = q.submit([&] (auto &h) {
    sycl::accessor macc(mbuf, h, sycl::read_only);
    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
    h.parallel_for(
        sycl::nd_range(sycl::range{N / BLOCK_SIZE}, sycl::range{64}),
        [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
            int group = it.get_group()[0];
            int gSize = it.get_local_range()[0];
            auto sg = it.get_sub_group();
            int sgSize = sg.get_local_range()[0];
            int sgGroup = sg.get_group_id()[0];

            unsigned int histogram[NUM_BINS]; // histogram bins take less storage
                                              // with smaller data type
            for (int k = 0; k < NUM_BINS; k++) {
                histogram[k] = 0;
            }
            for (int k = 0; k < BLOCK_SIZE; k++) {
                unsigned long x =
                    sg.load(macc.get_pointer() + group * gSize * BLOCK_SIZE +
                            sgGroup * sgSize * BLOCK_SIZE + sgSize * k);
                #pragma unroll
                for (int i = 0; i < 8; i++) {
                    unsigned int c = x & 0x1FU;
                    histogram[c] += 1;
                    x = x >> 8;
                }
            }
            for (int k = 0; k < NUM_BINS; k++) {
                hacc[k].fetch_add(histogram[k]);
            }
        });
});
```

## Do Not Declare Private Variables as Volatile

Now we make a small change to the code example:

```
constexpr int BLOCK_SIZE = 256;
constexpr int NUM_BINS = 32;

std::vector<unsigned long> hist(NUM_BINS, 0);

sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);
```

```

auto e = q.submit([&](auto &h) {
    sycl::accessor macc(mbuf, h, sycl::read_only);
    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
    h.parallel_for(sycl::nd_range(sycl::range{N / BLOCK_SIZE}, sycl::range{64}),
                   [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
                    int group = it.get_group()[0];
                    int gSize = it.get_local_range()[0];
                    auto sg = it.get_sub_group();
                    int sgSize = sg.get_local_range()[0];
                    int sgGroup = sg.get_group_id()[0];

                    volatile unsigned int
                        histogram[NUM_BINS]; // volatile variables will not
                                              // be assigned to any registers

                    for (int k = 0; k < NUM_BINS; k++) {
                        histogram[k] = 0;
                    }
                    for (int k = 0; k < BLOCK_SIZE; k++) {
                        unsigned long x = sg.load(
                            macc.get_pointer() + group * gSize * BLOCK_SIZE +
                            sgGroup * sgSize * BLOCK_SIZE + sgSize * k);
#pragma unroll
                        for (int i = 0; i < 8; i++) {
                            unsigned int c = x & 0x1FU;
                            histogram[c] += 1;
                            x = x >> 8;
                        }
                    }

                    for (int k = 0; k < NUM_BINS; k++) {
                        hacc[k].fetch_add(histogram[k]);
                    }
                });
});

```

The private histogram array is qualified as a volatile array. Volatile variables are not prompted to registers because their values may change between two different load operations.

There is really no reason for the private histogram array to be volatile, because it is only accessible by the local execution thread. In fact, if a private variable really needs to be volatile, it is not private any more.

## Do Not Take Address of a Private Variable and Later Dereference the Pointer

Now we make more changes to the code example:

```

constexpr int BLOCK_SIZE = 256;
constexpr int NUM_BINS = 32;

std::vector<unsigned long> hist(NUM_BINS, 0);

sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);

auto e = q.submit([&](auto &h) {
    sycl::accessor macc(mbuf, h, sycl::read_only);
    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
    h.parallel_for(

```

```

sycl::nd_range(sycl::range{N / BLOCK_SIZE}, sycl::range{64}),
[=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
    int group = it.get_group()[0];
    int gSize = it.get_local_range()[0];
    auto sg = it.get_sub_group();
    int sgSize = sg.get_local_range()[0];
    int sgGroup = sg.get_group_id()[0];

    unsigned int histogram[NUM_BINS]; // histogram bins take less storage
                                    // with smaller data type
    for (int k = 0; k < NUM_BINS; k++) {
        histogram[k] = 0;
    }
    for (int k = 0; k < BLOCK_SIZE; k++) {
        unsigned long x =
            sg.load(macc.get_pointer() + group * gSize * BLOCK_SIZE +
                    sgGroup * sgSize * BLOCK_SIZE + sgSize * k);
        unsigned long *p = &x;

#pragma unroll
        for (int i = 0; i < 8; i++) {
            unsigned int c = (*p & 0x1FU);
            histogram[c] += 1;
            *p = (*p >> 8);
        }
    }

    for (int k = 0; k < NUM_BINS; k++) {
        hacc[k].fetch_add(histogram[k]);
    }
});
});
```

The address of private variable `x` is taken and stored in pointer `p` and later `p` is dereferenced to access `x`.

Because its address is used, the variable `x` now has to reside in memory even though there is room for it in registers.

## Sharing Registers in a Sub-group

Now we increase the histogram bins to 256:

```

constexpr int BLOCK_SIZE = 256;
constexpr int NUM_BINS = 256;

std::vector<unsigned long> hist(NUM_BINS, 0);

sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);

auto e = q.submit([&](auto &h) {
    sycl::accessor macc(mbuf, h, sycl::read_only);
    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
    h.parallel_for(
        sycl::nd_range(sycl::range{N / BLOCK_SIZE}, sycl::range{64}),
        [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
            int group = it.get_group()[0];
            int gSize = it.get_local_range()[0];
            auto sg = it.get_sub_group();
            int sqSize = sg.get_local_range()[0];

```

```

int sgGroup = sg.get_group_id()[0];

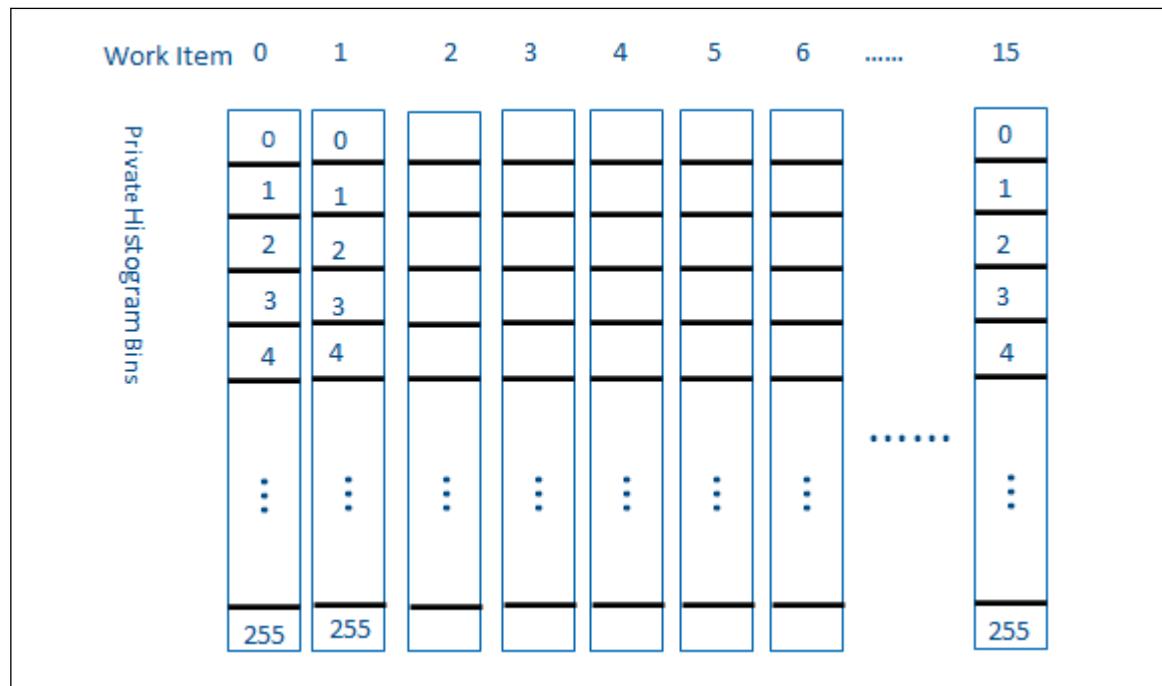
unsigned int
    histogram[NUM_BINS]; // histogram bins take too much storage to be
                          // promoted to registers
for (int k = 0; k < NUM_BINS; k++) {
    histogram[k] = 0;
}
for (int k = 0; k < BLOCK_SIZE; k++) {
    unsigned long x =
        sg.load(macc.get_pointer() + group * gSize * BLOCK_SIZE +
                sgGroup * sgSize * BLOCK_SIZE + sgSize * k);

#pragma unroll
    for (int i = 0; i < 8; i++) {
        unsigned int c = x & 0x1FU;
        histogram[c] += 1;
        x = x >> 8;
    }
}

for (int k = 0; k < NUM_BINS; k++) {
    hacc[k].fetch_add(histogram[k]);
}
});
```

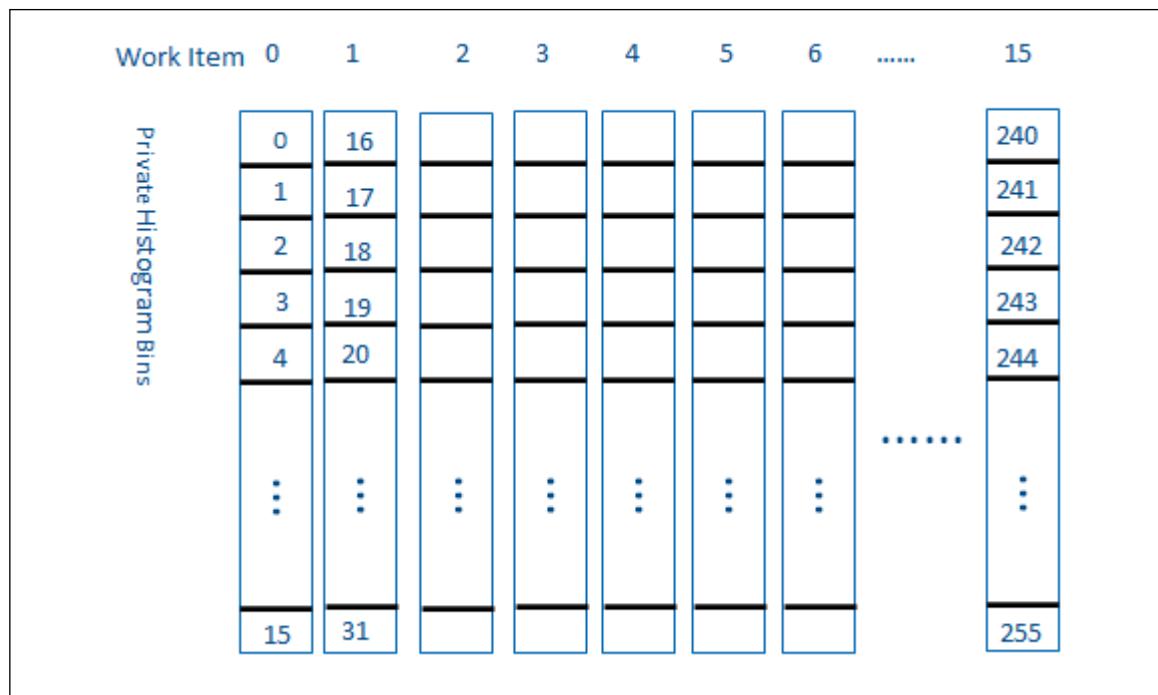
With 256 histogram bins, the performance degrades even with smaller data type unsigned integer. The storage of the private bins in each work item is too large for registers.

## **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 Vector Engine 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.

### 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.

```
constexpr int BLOCK_SIZE = 256;
constexpr int NUM_BINS = 256;

std::vector<unsigned long> hist(NUM_BINS, 0);

sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);

auto e = q.submit([&] (auto &h) {
    sycl::accessor macc(mbuf, h, sycl::read_only);
    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
    h.parallel_for(
        sycl::nd_range(sycl::range{N / BLOCK_SIZE}, sycl::range{64}),
        [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
            int group = it.get_group()[0];
            int gSize = it.get_local_range()[0];
            auto sg = it.get_sub_group();
            int sgSize = sg.get_local_range()[0];
            int sgGroup = sg.get_group_id()[0];

            unsigned int
                histogram[NUM_BINS / 16]; // histogram bins take too much storage
                                         // to be promoted to registers
            for (int k = 0; k < NUM_BINS / 16; k++) {
                histogram[k] = 0;
                if (group == sgGroup) {
                    hacc[sg.get_global_linear_id() / 16] = histogram[k];
                }
            }
        });
});
```

```

    }
    for (int k = 0; k < BLOCK_SIZE; k++) {
        unsigned long x =
            sg.load(macc.get_pointer() + group * gSize * BLOCK_SIZE +
                    sgGroup * sgSize * BLOCK_SIZE + sgSize * k);
// subgroup size is 16
#pragma unroll
        for (int j = 0; j < 16; j++) {
            unsigned long y = sycl::group_broadcast(sg, x, j);
#pragma unroll
            for (int i = 0; i < 8; i++) {
                unsigned int c = y & 0xFF;
                // (c & 0xF) is the workitem in which the bin resides
                // (c >> 4) is the bin index
                if (sg.get_local_id()[0] == (c & 0xF)) {
                    histogram[c >> 4] += 1;
                }
                y = y >> 8;
            }
        }
    }

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

```

## Using Sub-group Block Load/Store

Memory loads/stores are vectorized. Each lane of a vector load/store instruction has its own address and data. Both addresses and data take register space. For example:

```

constexpr int N = 1024 * 1024;
int *data = sycl::malloc_shared<int>(N, q);
int *data2 = sycl::malloc_shared<int>(N, q);
memset(data2, 0xFF, sizeof(int) * N);

auto e = q.submit([&](auto &h) {
    h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{32}),
                  [=](sycl::nd_item<1> it) {
                      int i = it.get_global_linear_id();
                      data[i] = data2[i];
                  });
});

```

The memory loads and stores in the statement:

```
``data[i] = data2[i];``
```

are vectorized and each vector lane has its own address. Assuming the SIMD width or the sub-group size is 16, total register space for addresses of the 16 lanes is 128 bytes. If each GRF register is 32-byte wide, 4 GRF registers are needed for the addresses.

Noticing the addresses are contiguous, we can use sub-group block load/store built-ins to save register space for addresses:

```

constexpr int N = 1024 * 1024;
int *data = sycl::malloc_shared<int>(N, q);
int *data2 = sycl::malloc_shared<int>(N, q);

```

```

memset(data2, 0xFF, sizeof(int) * N);

auto e = q.submit([&] (auto &h) {
    h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{32}),
                   [=] (sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
                       auto sg = it.get_sub_group();

                       int base =
                           (it.get_group(0) * 32 +
                            sg.get_group_id()[0] * sg.get_local_range()[0]);

                       auto load_ptr = get_multi_ptr(&(data2[base + 0]));
                       int x = sg.load(load_ptr);

                       auto store_ptr = get_multi_ptr(&(data[base + 0]));
                       sg.store(store_ptr, x);
                   });
    });

```

The statements:

```

``x = sg.load(global_ptr(&(data2[base + 0])));
sg.store(global_ptr(&(data[base + 0])), x);``

```

each loads/stores a contiguous block of memory and the compiler will compile these 2 statements into special memory block load/store instructions. And because it is a contiguous memory block, we only need the starting address of the block. So 8, instead of 128, bytes of actual register space, or at most 1 register, is used for the address for each block load/store.

## Using Shared Local Memory

If the number of histogram bins gets larger than, 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. Refer to the "Shared Local Memory" chapter and see how it is done in the histogram example there.

## Porting Code with High Register Pressure to Intel® Max GPUs

When migrating code from CUDA to SYCL, it may happen that the resulting SYCL code does not perform well on Intel® Data Center GPUs (e.g. the Intel® Data Center GPU Max 1550) despite showing good performance on NVIDIA GPUs. In some cases this is due to register spills on Intel® GPUs which do not occur on NVIDIA GPUs due to the different sizes of the general-purpose register files (GRF). Possible ways to handle these register spills without changes to the code is discussed in what follows. Further optimization techniques for reducing the overall register pressure can be found in [Optimizing Register Spills](#).

## A Simple Code Example with High Register Pressure

The example used for this demonstration is inspired by code performing automatic differentiation utilizing multivariate dual numbers. No knowledge about dual numbers is required to understand the following example. For more information on dual numbers we refer to the [Wikipedia page](#) and [this book](#).

```

#include <chrono>
#include <iostream>
#include <sycl/sycl.hpp>

#define NREGISTERS 80

#ifndef SIMD16
#define SIMD_SIZE 32

```

```

#else
#define SIMD_SIZE 16
#endif

class my_type {
public:
    my_type(double x0, double x1, double x2, double x3, double x4, double x5,
            double x6, double x7)
        : x0_(x0), x1_(x1), x2_(x2), x3_(x3), x4_(x4), x5_(x5), x6_(x6), x7_(x7) {}

    my_type(double const *const vals)
        : x0_(vals[0]), x1_(vals[1]), x2_(vals[2]), x3_(vals[3]), x4_(vals[4]),
          x5_(vals[5]), x6_(vals[6]), x7_(vals[7]) {}

    my_type operator+(const my_type &rhs) {
        return my_type(rhs.x0_ + x0_, rhs.x1_ + x1_, rhs.x2_ + x2_, rhs.x3_ + x3_,
                      rhs.x4_ + x4_, rhs.x5_ + x5_, rhs.x6_ + x6_, rhs.x7_ + x7_);
    }

    void WriteBack(double *const vals) {
        vals[0] += x0_;
        vals[1] += x1_;
        vals[2] += x2_;
        vals[3] += x3_;
        vals[4] += x4_;
        vals[5] += x5_;
        vals[6] += x6_;
        vals[7] += x7_;
    }
}

private:
    double x0_, x1_, x2_, x3_, x4_, x5_, x6_, x7_;

int main(int argc, char **argv) {
    sycl::queue Q(sycl::gpu_selector_v);
    size_t nsubgroups;
    if (argc > 1)
        nsubgroups = 10;
    else
        nsubgroups = 8;

    const size_t ARR_SIZE = nsubgroups * SIMD_SIZE * NREGISTERS;
    double *val_in = sycl::malloc_shared<double>(ARR_SIZE, Q);
    double *val_out = sycl::malloc_shared<double>(ARR_SIZE, Q);

    std::cout << "Using simd size " << SIMD_SIZE << std::endl;

    Q.parallel_for(ARR_SIZE, [=](auto it) {
        val_in[it.get_id()] = argc * it.get_id();
    }).wait();

    auto start_time = std::chrono::high_resolution_clock::now();

    for (int rep = 0; rep < 1000; rep++) {
        Q.parallel_for(sycl::nd_range<1>(nsubgroups * SIMD_SIZE, SIMD_SIZE),
                      [=](auto it) [[intel::reqd_sub_group_size(SIMD_SIZE)]] {

```

```
    const size_t id = it.get_global_linear_id();

    double const *const val_offs = val_in + id * NREGISTERS;

    my_type v0(val_offs + 0);
    my_type v1(val_offs + 8);
    my_type v2(val_offs + 16);
    my_type v3(val_offs + 24);
    my_type v4(val_offs + 32);
    my_type v5(val_offs + 40);
    my_type v6(val_offs + 48);
    my_type v7(val_offs + 56);
    my_type v8(val_offs + 64);
    my_type v9(val_offs + 72);

    my_type p0 = v0 + v5;
    my_type p1 = v1 + v6;
    my_type p2 = v2 + v7;
    my_type p3 = v3 + v8;
    my_type p4 = v4 + v9;

    p0 = p0 + p1 + v5;
    p2 = p2 + p3 + v6;
    p4 = p4 + p1 + v7;
    p3 = p3 + p2 + v8;
    p2 = p0 + p4 + v9;
    p1 = p1 + v0;

    double *const vals_out_offs = val_out + id * NREGISTERS;
    p0.WriteBack(vals_out_offs);
    p1.WriteBack(vals_out_offs + 8);
    p2.WriteBack(vals_out_offs + 16);
    p3.WriteBack(vals_out_offs + 24);
    p4.WriteBack(vals_out_offs + 32);
    v0.WriteBack(vals_out_offs + 40);
    v1.WriteBack(vals_out_offs + 48);
    v2.WriteBack(vals_out_offs + 56);
    v3.WriteBack(vals_out_offs + 64);
    v4.WriteBack(vals_out_offs + 72);
}

.wait();
}

auto end_time = std::chrono::high_resolution_clock::now();

std::cout << "Took "
      << std::chrono::duration_cast<std::chrono::microseconds>(end_time -
                                                               start_time)
      .count()
      << " microseconds" << std::endl;

sycl::free(val_in, Q);
sycl::free(val_out, Q);

return 0;
}
```

The code shows a class `my_type`, which represents a dimension 7 dual number and which wraps 8 doubles and includes overloads of arithmetic operators. The main function of the program initializes some data and then launches a specific GPU kernel 1000 times (for loop in line 70, kernel launch in line 72). These repeated launches are purely for the purpose of getting more accurate performance numbers and do not influence the register spills under consideration. The kernel uses the data to initialize several `my_type` dual numbers and then performs several arithmetic operations with them in such a way that they produce high register pressure.

More precisely, the SYCL code compiled for an NVIDIA A100 GPU with [oneAPI for NVIDIA GPUs](#) and the following command

```
clang++ -O2 -Xcuda-ptxas -v -fsycl -fsycl-targets=nvidia_gpu_sm_80 --cuda-path=/opt/hpc_software/compilers/nvidia/cuda-12.0 main.cpp -o exec_cuda.out
```

shows that 168 registers are allocated, and none are spilled with the following output

```
ptxas info      : Used 168 registers, 380 bytes cmem[0]
```

Compiling the code for PVC, on the other hand, using ahead-of-time (AOT) compilation (cf. [Ahead-Of-Time Compilation](#)) with oneAPI 2023.2.0 with the following command

```
icpx -fsycl -fsycl-targets=spir64_gen -Xsycl-target-backend "-device pvc" main.cpp -o exec_jit.out
```

results in register spills as indicated by a compiler warning such as the following

```
warning: kernel _ZTSZ4mainEULT_E0_ compiled SIMD32 allocated 128 regs and spilled around 396
```

The execution time of the code on an A100 80GB GPU and a single stack of the Intel® Data Center GPU Max 1550 is shown in the following table.

#### **Execution time on a single stack of Intel® Max 1550 vs NVIDIA A100 80GB**

Hardware	Time (ms), lower is better	Register Spills
A100 80GB	21.5	No
1T Max 1550	36.1	Yes

It is evident that the execution time on the Intel® GPU is significantly longer than on the NVIDIA GPU. Based on the information above, it is reasonable to expect that the difference is related to the register allocations. To understand the underlying cause, the following section points out the differences in the register files.

#### **Registers on an Intel® Data Center GPU Max 1550**

The difference in the register allocation is due to the different sizes of the general-purpose register files (GRF). More precisely, a NVIDIA A100 GPU provides up to 256 32-bit wide registers per work-item whereas the Intel® Data Center GPU Max 1550 provides by default 64 32-bit wide registers (assuming the same number of 32 work-items per sub-group as NVIDIA, cf. [Intel® Xe GPU Architecture](#)), i.e., a quarter of the NVIDIA GPU. There are two possibilities to adjust the available number of registers: the large GRF mode (cf. [Small Register Mode vs. Large Register Mode](#)) and a reduced sub-group size (cf. [Sub-Groups and SIMD Vectorization](#)). The impact of these two options on the above example are discussed in the following section.

#### **GRF on Intel® GPU Max 1550 with and without large GRF and different SIMD widths.**

Hardware	SIMD width	GRF mode	Registers per work-item
A100 80GB	32	N.A	256 x 32-bit
1T Max 1550	32	default	64 x 32-bit
1T Max 1550	32	large GRF	128 x 32-bit

Hardware	SIMD width	GRF mode	Registers per work-item
1T Max 1550	16	default	128 x 32-bit
1T Max 1550	16	large GRF	256 x 32-bit

### Increasing the Number of Registers per Work-item

The above example requires 168 registers. Based on the discussion in the previous section, large GRF mode and a reduced SIMD width of 16 together should therefore avoid register spills. In what follows we check the three remaining combinations of the GRF mode and the SIMD width and we will see that, indeed, large GRF mode and SIMD 16 is required to avoid all register spills.

First, compiling the code again with large GRF, using

```
icpx -fsycl -fsycl-targets=spir64_gen -Xsycl-target-backend "-device pvc -options -ze-opt-large-register-file" main.cpp -o exec_largegrf.out
```

still shows the following warning about register spills, although significantly less than before.

```
warning: kernel _ZTSZ4mainEULT_E0_ compiled SIMD32 allocated 256 regs and spilled around 206
```

This is expected considering that only 128 32-bit wide registers are now available for the 168 required registers.

The second option to reduce the register spills is to reduce the SIMD width from 32 to 16 through a small change in the code. This can be done in this case by compiling the code with the following command

```
icpx -DSIMD16 -fsycl -fsycl-targets=spir64_gen -Xsycl-target-backend "-device pvc" main.cpp -o exec_simd16.out
```

Using the default GRF mode with SIMD width 16 again shows a warning about register spills.

```
warning: kernel _ZTSZ4mainEULT_E0_ compiled SIMD16 allocated 128 regs and spilled around 112
```

Using both, a reduced SIMD width of 16 and large GRF mode, removes all register spills and no warning is given by the compiler, which is in line with the initial expectations.

The execution times for the four configurations are shown in the following table.

### Execution times on a single stack of Intel® Data Center GPU Max 1550 with the different settings.

GRF mode	SIMD width	Register Spills	Time (ms), lower is better
default	32	Yes	36.1
large GRF	32	Yes	32.2
default	16	Yes	30.3
large GRF	16	No	27.7

Clearly, the best performance is achieved in the case where no register spills happen. It is interesting to note that SIMD width 16 is preferable to large GRF mode. This is not surprising considering that with large GRF mode each XVE has 4 threads.

### Special Considerations When Porting to SIMD 16

There are certain cases where the above methods may not be suitable or require special attention. On the one hand, as indicated above, with large GRF mode, 4 instead of 8 threads are available on each XVE. This may result in worse performance for certain cases.

On the other hand, switching from SIMD 32 to SIMD 16 may require special attention since it may lead to wrong results. An example is shown in the following code.

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

#define NREGISTERS 80

#ifndef SIMD16
#define SIMD_SIZE 32
#else
#define SIMD_SIZE 16
#endif

class my_type {
public:
    my_type(double x0, double x1, double x2, double x3, double x4, double x5,
            double x6, double x7)
        : x0_(x0), x1_(x1), x2_(x2), x3_(x3), x4_(x4), x5_(x5), x6_(x6), x7_(x7) {}

    my_type(double const *const vals)
        : x0_(vals[0]), x1_(vals[1]), x2_(vals[2]), x3_(vals[3]), x4_(vals[4]),
          x5_(vals[5]), x6_(vals[6]), x7_(vals[7]) {}

    my_type operator+(const my_type &rhs) {
        return my_type(rhs.x0_ + x0_, rhs.x1_ + x1_, rhs.x2_ + x2_, rhs.x3_ + x3_,
                      rhs.x4_ + x4_, rhs.x5_ + x5_, rhs.x6_ + x6_, rhs.x7_ + x7_);
    }

    void WriteBack(double *const vals) {
        vals[0] += x0_;
        vals[1] += x1_;
        vals[2] += x2_;
        vals[3] += x3_;
        vals[4] += x4_;
        vals[5] += x5_;
        vals[6] += x6_;
        vals[7] += x7_;
    }

    void Load(double const *const vals) {
        x0_ = vals[0];
        x1_ = vals[1];
        x2_ = vals[2];
        x3_ = vals[3];
        x4_ = vals[4];
        x5_ = vals[5];
        x6_ = vals[6];
        x7_ = vals[7];
    }

private:
    double x0_, x1_, x2_, x3_, x4_, x5_, x6_, x7_;
};

int main(int argc, char **argv) {
    sycl::queue Q(sycl::gpu_selector_v);
```

```

size_t nsubgroups;
if (argc > 1)
    nsubgroups = 10;
else
    nsubgroups = 8;

const size_t ARR_SIZE = nsubgroups * SIMD_SIZE * NREGISTERS;
double *val_in = sycl::malloc_shared<double>(ARR_SIZE, Q);
double *val_out = sycl::malloc_shared<double>(ARR_SIZE, Q);

std::cout << "Using simd size " << SIMD_SIZE << std::endl;

Q.parallel_for(ARR_SIZE, [=](auto it) {
    val_in[it.get_id()] = argc * it.get_id();
}).wait();

auto start_time = std::chrono::high_resolution_clock::now();

for (int rep = 0; rep < 1000; rep++) {
    Q.submit([&](auto &h) {
        sycl::accessor<double, 1, sycl::access::mode::read_write,
                    sycl::access::target::local>
        slm(sycl::range(32 * 8), h);

        h.parallel_for(sycl::nd_range<1>(nsubgroups * SIMD_SIZE, 32),
                      [=](auto it) [[intel::reqd_sub_group_size(SIMD_SIZE)]] {
            const int id = it.get_global_linear_id();
            const int local_id = it.get_local_id(0);
            for (int i = 0; i < 8; i++) {
                slm[i + local_id * 8] = 0.0;
            }
        });
    #ifdef SIMD16
        it.barrier(sycl::access::fence_space::local_space);
    #endif
        double const *const val_offs = val_in + id * NREGISTERS;

        my_type v0(val_offs + 0);
        my_type v1(val_offs + 8);
        my_type v2(val_offs + 16);
        my_type v3(val_offs + 24);
        my_type v4(val_offs + 32);
        my_type v5(val_offs + 40);
        my_type v6(val_offs + 48);
        my_type v7(val_offs + 56);
        my_type v8(val_offs + 64);
        my_type v9(val_offs + 72);

        my_type p0 = v0 + v5;
        v0.WriteBack(&slm[local_id * 8]);
    #ifdef SIMD16
        it.barrier(sycl::access::fence_space::local_space);
    #endif
        v0.Load(&slm[32 * 8 - 8 - local_id * 8]);
        my_type p1 = v1 + v6;
        my_type p2 = v2 + v7;
        my_type p3 = v3 + v8;
        my_type p4 = v4 + v9;
    });
}

```

```

    p0 = p0 + p1 + v5;
    p2 = p2 + p3 + v6;
    p4 = p4 + p1 + v7;
    p3 = p3 + p2 + v8;
    p2 = p0 + p4 + v9;

    p1 = p1 + v0;

    double *const vals_out_offs = val_out + id * NREGISTERS;
    p0.WriteBack(vals_out_offs);
    p1.WriteBack(vals_out_offs + 8);
    p2.WriteBack(vals_out_offs + 16);
    p3.WriteBack(vals_out_offs + 24);
    p4.WriteBack(vals_out_offs + 32);
    v0.WriteBack(vals_out_offs + 40);
    v1.WriteBack(vals_out_offs + 48);
    v2.WriteBack(vals_out_offs + 56);
    v3.WriteBack(vals_out_offs + 64);
    v4.WriteBack(vals_out_offs + 72);
});

}).wait();
}

auto end_time = std::chrono::high_resolution_clock::now();

std::cout << "Took "
<< std::chrono::duration_cast<std::chrono::microseconds>(end_time -
start_time)
.count()
<< " microseconds" << std::endl;

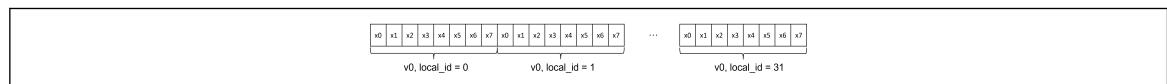
sycl::free(val_in, Q);
sycl::free(val_out, Q);

return 0;
}

```

It is similar to the first example above but introduces a shared local memory array (SLM, cf. [Shared Local Memory](#)) which is utilized to transpose the values stored in `v0` within each work-group (which coincides with the sub-groups when using SIMD 32), as illustrated in the following figure and table.

**The figure illustrates how the SLM is used in the second code example. The `v0` value of each work-item in the work-group is stored in SLM. Each `v0` consists of eight double values `x0 - x7`.**



**After storing the `v0` values in SLM, they are read in reverse order.**

work-item local id store to SLM	work-item local id load from SLM
0	31
1	30
2	29
...	...

work-item local id store to SLM	work-item local id load from SLM
29	2
30	1
31	0

In the present case, the switch from SIMD 32 to SIMD 16 produces in general wrong results. The reason is that the synchronization between 32 work-items accessing the SLM is not guaranteed in the SIMD 16 case. This is not an issue in the SIMD 32 case since the work-items are in the same sub-group, which are processed together (cf. [Sub-Groups and SIMD Vectorization](#)). Thus, in the SIMD 16 case it may happen that a value in SLM is read and used (line 120) before it is actually written (line 116). To solve this issue, a SLM barrier needs to be added, as shown in line 118, to ensure that all values are written before any are read. Note that these barriers may negatively impact the computing time. The computing times for this example in the four configurations, are shown in the following table.

#### Execution times of the SLM example on a single stack of Intel® Data Center GPU Max 1550 with the different settings.

GRF mode	SIMD width	Register Spills	Time (ms), lower is better
default	32	Yes	38.0
large GRF	32	Yes	33.7
default	16	Yes	31.3
large GRF	16	No	29.7

To summarize, when porting high-register-pressure code from NVIDIA GPUs to Intel® GPUs, the code may show sub-optimal performance due to register spills induced by the different GRF sizes. In such a case, there are simple ways to increase the GRF size to better align the performance on the different devices with minimal code adjustments.

## 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 a vector engine (VE) 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 X<sup>e</sup>-core of Intel GPUs has its own SLM. Access to the SLM is limited to the VEs in the X<sup>e</sup>-core or work-items in the same work-group scheduled to execute on the VEs of the same X<sup>e</sup>-core. It is local to a X<sup>e</sup>-core (or work-group) and shared by VEs in the same X<sup>e</sup>-core (or work-items in the same work-group), so it is called SLM. Because it is on-chip in each X<sup>e</sup>-core, 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 a significant performance difference.

## 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.:

```
std::cout << "Local Memory Size: "
<< q.get_device().get_info<sycl::info::device::local_mem_size>()
<< 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.

## 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:

```
constexpr int N = 32;
int *data = sycl::malloc_shared<int>(N, q);

auto e = q.submit([&] (auto &h) {
    sycl::local_accessor<int, 1> slm(sycl::range(32 * 64), h);
    h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{32}),
                  [=] (sycl::nd_item<1> it) {
                      int i = it.get_global_linear_id();
                      int j = it.get_local_linear_id();

                      slm[j * 16] = 0;
                      it.barrier(sycl::access::fence_space::local_space);

                      for (int m = 0; m < 1024 * 1024; m++) {
                          slm[j * 16] += i * m;
                          it.barrier(sycl::access::fence_space::local_space);
                      }

                      data[i] = slm[j * 16];
                  });
});
```

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.

```
constexpr int N = 32;
int *data = sycl::malloc_shared<int>(N, q);

auto e = q.submit([&] (auto &h) {
    sycl::local_accessor<int, 1> slm(sycl::range(32 * 64), h);
```

```

    h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{32}),
                  [=](sycl::nd_item<1> it) {
                      int i = it.get_global_linear_id();
                      int j = it.get_local_linear_id();

                      slm[j] = 0;
                      it.barrier(sycl::access::fence_space::local_space);

                      for (int m = 0; m < 1024 * 1024; m++) {
                          slm[j] += i * m;
                          it.barrier(sycl::access::fence_space::local_space);
                      }

                      data[i] = slm[j];
                  });
    });
}

```

## Data Sharing and Work-group Barriers

Let us consider the histogram with 256 bins example from the [Optimizing Register Spills](#) once again.

```

constexpr int BLOCK_SIZE = 256;
constexpr int NUM_BINS = 256;

std::vector<unsigned long> hist(NUM_BINS, 0);

sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);

auto e = q.submit([&] (auto &h) {
    sycl::accessor macc(mbuf, h, sycl::read_only);
    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
    h.parallel_for(
        sycl::nd_range(sycl::range{N / BLOCK_SIZE}, sycl::range{64}),
        [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
            int group = it.get_group()[0];
            int gSize = it.get_local_range()[0];
            auto sg = it.get_sub_group();
            int sgSize = sg.get_local_range()[0];
            int sgGroup = sg.get_group_id()[0];

            unsigned int
                histogram[NUM_BINS / 16]; // histogram bins take too much storage
                                         // to be promoted to registers
            for (int k = 0; k < NUM_BINS / 16; k++) {
                histogram[k] = 0;
            }
            for (int k = 0; k < BLOCK_SIZE; k++) {
                unsigned long x =
                    sg.load(macc.get_pointer() + group * gSize * BLOCK_SIZE +
                            sgGroup * sgSize * BLOCK_SIZE + sgSize * k);
// subgroup size is 16
#pragma unroll
                for (int j = 0; j < 16; j++) {
                    unsigned long y = sycl::group_broadcast(sg, x, j);
#pragma unroll
                    for (int i = 0; i < 8; i++) {
                        unsigned int c = y & 0xFF;

```

```

        // (c & 0xF) is the workitem in which the bin resides
        // (c >> 4) is the bin index
        if (sg.get_local_id()[0] == (c & 0xF)) {
            histogram[c >> 4] += 1;
        }
        y = y >> 8;
    }
}

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

```

This example has been optimized to use the 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 you 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.

```

constexpr int NUM_BINS = 1024;
constexpr int BLOCK_SIZE = 256;

std::vector<unsigned long> hist(NUM_BINS, 0);
sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);

auto e = q.submit([&](auto &h) {
    sycl::accessor macc(mbuf, h, sycl::read_only);
    sycl::accessor hacc(hbuf, h, sycl::read_write);
    sycl::local_accessor<unsigned int, 1> local_histogram(sycl::range(NUM_BINS),
                                                          h);
    h.parallel_for(
        sycl::nd_range(sycl::range{N / BLOCK_SIZE}, sycl::range{64}),
        [=](sycl::nd_item<1> it) {
            int group = it.get_group()[0];
            int gSize = it.get_local_range()[0];
            auto sg = it.get_sub_group();
            int sgSize = sg.get_local_range()[0];
            int sgGroup = sg.get_group_id()[0];

            int factor = NUM_BINS / gSize;
            int local_id = it.get_local_id()[0];
            if ((factor <= 1) && (local_id < NUM_BINS)) {
                sycl::atomic_ref<unsigned int, sycl::memory_order::relaxed,
                               sycl::memory_scope::device,
                               sycl::access::address_space::local_space>
                    local_bin(local_histogram[local_id]);
                local_bin.store(0);
            } else {
                for (int k = 0; k < factor; k++) {
                    sycl::atomic_ref<unsigned int, sycl::memory_order::relaxed,
                                   sycl::memory_scope::device,
                                   sycl::access::address_space::local_space>
                        local_bin(local_histogram[gSize * k + local_id]);
                    local_bin.store(0);
                }
            }
        }
    );
});

```

```

    }
    it.barrier(sycl::access::fence_space::local_space);

    for (int k = 0; k < BLOCK_SIZE; k++) {
        unsigned long x =
            sg.load(macc.get_pointer() + group * gSize * BLOCK_SIZE +
                    sgGroup * sgSize * BLOCK_SIZE + sgSize * k);

#pragma unroll
        for (std::uint8_t shift : {0, 16, 32, 48}) {
            constexpr unsigned long mask = 0x3FFU;
            sycl::atomic_ref<unsigned int, sycl::memory_order::relaxed,
                sycl::memory_scope::device,
                sycl::access::address_space::local_space>
                local_bin(local_histogram[(x >> shift) & mask]);
            local_bin += 1;
        }
    }
    it.barrier(sycl::access::fence_space::local_space);

    if ((factor <= 1) && (local_id < NUM_BINS)) {
        sycl::atomic_ref<unsigned int, sycl::memory_order::relaxed,
            sycl::memory_scope::device,
            sycl::access::address_space::local_space>
            local_bin(local_histogram[local_id]);
        sycl::atomic_ref<unsigned long, sycl::memory_order::relaxed,
            sycl::memory_scope::device,
            sycl::access::address_space::global_space>
            global_bin(hacc[local_id]);
        global_bin += local_bin.load();
    } else {
        for (int k = 0; k < factor; k++) {
            sycl::atomic_ref<unsigned int, sycl::memory_order::relaxed,
                sycl::memory_scope::device,
                sycl::access::address_space::local_space>
                local_bin(local_histogram[gSize * k + local_id]);
            sycl::atomic_ref<unsigned long, sycl::memory_order::relaxed,
                sycl::memory_scope::device,
                sycl::access::address_space::global_space>
                global_bin(hacc[gSize * k + local_id]);
            global_bin += local_bin.load();
        }
    }
};

});
```

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 24-38 in the above example). You 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 39 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.

You can find an example of an SLM version of a 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 think about further optimizations that can be done.

## Using SLM as Cache

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

Consider the following 1-D convolution example:

```

sycl::buffer<int> ibuf(input.data(), N);
sycl::buffer<int> obuf(output.data(), N);
sycl::buffer<int> kbuf(kernel.data(), M);

auto e = q.submit([&] (auto &h) {
    sycl::accessor iacc(ibuf, h, sycl::read_only);
    sycl::accessor oacc(obuf, h);
    sycl::accessor kacc(kbuf, h, sycl::read_only);

    h.parallel_for(sycl::nd_range<1>(sycl::range{N}, sycl::range{256}),
                  [=](sycl::nd_item<1> it) {
                      int i = it.get_global_linear_id();
                      int group = it.get_group()[0];
                      int gSize = it.get_local_range()[0];

                      int t = 0;
                      int _M = static_cast<int>(M);
                      int _N = static_cast<int>(N);

                      if ((group == 0) || (group == _N / gSize - 1)) {
                          if (i < _M / 2) {
                              for (int j = _M / 2 - i, k = 0; j < _M; ++j, ++k) {
                                  t += iacc[k] * kacc[j];
                              }
                          } else {
                              if (i + _M / 2 >= _N) {
                                  for (int j = 0, k = i - _M / 2;
                                       j < _M / 2 + _N - i; ++j, ++k) {
                                      t += iacc[k] * kacc[j];
                                  }
                              } else {
                                  for (int j = 0, k = i - _M / 2; j < _M; ++j, ++k) {
                                      t += iacc[k] * kacc[j];
                                  }
                              }
                          }
                      } else {
                          for (int j = 0, k = i - _M / 2; j < _M; ++j, ++k) {
                              t += iacc[k] * kacc[j];
                          }
                      }
                  });
}

```

```

        oacc[i] = t;
    });
});

```

The example convolves an integer array of  $8192 \times 8192$  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, you 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.

```

sycl::buffer<int> ibuf(input.data(), N);
sycl::buffer<int> obuf(output.data(), N);
sycl::buffer<int> kbuf(kernel.data(), M);

auto e = q.submit([&](auto &h) {
    sycl::accessor iacc(ibuf, h, sycl::read_only);
    sycl::accessor oacc(obuf, h);
    sycl::accessor kacc(kbuf, h, sycl::read_only);
    sycl::local_accessor<int, 1> ciacc(sycl::range(256 + (M / 2) * 2), h);

    h.parallel_for(
        sycl::nd_range(sycl::range{N}, sycl::range{256}),
        [=](sycl::nd_item<1> it) {
            int i = it.get_global_linear_id();
            int group = it.get_group()[0];
            int gSize = it.get_local_range()[0];
            int local_id = it.get_local_id()[0];
            int _M = static_cast<int>(M);

            ciacc[local_id + M / 2] = iacc[i];

            if (local_id == 0) {
                if (group == 0) {
                    for (int j = 0; j < _M / 2; ++j) {
                        ciacc[j] = 0;
                    }
                } else {
                    for (int j = 0, k = i - _M / 2; j < _M / 2; ++j, ++k) {
                        ciacc[j] = iacc[k];
                    }
                }
            }
            if (local_id == gSize - 1) {
                if (group == static_cast<int>(it.get_group_range()[0]) - 1) {
                    for (int j = gSize + _M / 2; j < gSize + _M / 2 + _M / 2; ++j) {
                        ciacc[j] = 0;
                    }
                } else {
                    for (int j = gSize + _M / 2, k = i + 1;
                         j < gSize + _M / 2 + _M / 2; ++j, ++k) {
                        ciacc[j] = iacc[k];
                    }
                }
            }
        });

    it.barrier(sycl::access::fence_space::local_space);
});

```

```

        int t = 0;
        for (int j = 0, k = local_id; j < _M; ++j, ++k) {
            t += ciacc[k] * kacc[j];
        }

        oacc[i] = t;
    });
});

```

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.

## Troubleshooting SLM Errors

A PI\_ERROR\_OUT\_OF\_RESOURCES error may occur when a kernel uses more shared local memory than the amount available on the hardware. When this occurs, you will see an error message similar to this:

```

$ ./myapp
:
terminate called after throwing an instance of 'sycl::_V1::runtime_error'
what(): Native API failed. Native API returns: -5
(PI_ERROR_OUT_OF_RESOURCES) -5 (PI_ERROR_OUT_OF_RESOURCES)
Aborted (core dumped)
$
```

To see how much memory was being requested and the actual hardware limit, set debug keys:

```

export PrintDebugMessages=1
export NEOReadDebugKeys=1
```

This will change the output to:

```

$ ./myapp
:
Size of SLM (656384) larger than available (131072)
terminate called after throwing an instance of 'sycl::_V1::runtime_error'
what(): Native API failed. Native API returns: -5
(PI_ERROR_OUT_OF_RESOURCES) -5 (PI_ERROR_OUT_OF_RESOURCES)
Aborted (core dumped)
$
```

## Pointer Aliasing and the 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.

```

size_t VectorAdd(sycl::queue &q, const IntArray &a, const IntArray &b,
                  IntArray &sum, int iter) {
    sycl::range num_items{a.size()};

    sycl::buffer a_buf(a);
```

```

sycl::buffer b_buf(b);
sycl::buffer sum_buf(sum.data(), num_items);

auto start = std::chrono::steady_clock::now();
for (int i = 0; i < iter; i++) {
    auto e = q.submit([&](auto &h) {
        // Input accessors
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        // Output accessor
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

        h.parallel_for(num_items,
                      [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
    });
}
q.wait();
auto end = std::chrono::steady_clock::now();
std::cout << "Vector add completed on device - took " << (end - start).count()
      << " u-secs\n";
return ((end - start).count());
} // end VectorAdd

```

In this case, the programmer leaves all the choices about vector length and the number of work-groups to the compiler. In most cases the 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 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.

```

size_t VectorAdd2(sycl::queue &q, const IntArray &a, const IntArray &b,
                  IntArray &sum, int iter) {
    sycl::range num_items{a.size()};

    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf(sum.data(), num_items);
    // size_t num_groups =
    // q.get_device().get_info<sycl::info::device::max_compute_units>(); size_t
    // wg_size =
    // q.get_device().get_info<sycl::info::device::max_work_group_size>();
    size_t num_groups = 1;
    size_t wg_size = 16;
    auto start = std::chrono::steady_clock::now();
    for (int i = 0; i < iter; i++) {
        q.submit([&](auto &h) {
            // Input accessors
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            // Output accessor
            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

            h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                          [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(
                            16)]] [[intel::kernel_args_restrict]] {

```

```

        size_t loc_id = index.get_local_id();
        // unroll with a directive
#pragma unroll(2)
        for (size_t i = loc_id; i < mysize; i += wg_size) {
            sum_acc[i] = a_acc[i] + b_acc[i];
        }
    });
}
q.wait();
auto end = std::chrono::steady_clock::now();
std::cout << "Vector add2 completed on device - took "
    << (end - start).count() << " u-secs\n";
return ((end - start).count());
} // 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.

```

size_t VectorAdd3(sycl::queue &q, const IntArray &a, const IntArray &b,
                  IntArray &sum, int iter) {
    sycl::range num_items(a.size());

    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf(sum.data(), num_items);
    size_t num_groups = 1;
    size_t wg_size = 16;
    auto start = std::chrono::steady_clock::now();
    for (int i = 0; i < iter; i++) {
        q.submit([&](auto &h) {
            // Input accessors
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            // Output accessor
            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

            h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                           [=](sycl::nd_item<1> index)
                           [[intel::reqd_sub_group_size(16)]] {
                               // Manual unrolling
                               size_t loc_id = index.get_local_id();
                               for (size_t i = loc_id; i < mysize; i += 32) {
                                   sum_acc[i] = a_acc[i] + b_acc[i];
                                   sum_acc[i + 16] = a_acc[i + 16] + b_acc[i + 16];
                               }
                           });
        });
    }
    q.wait();
    auto end = std::chrono::steady_clock::now();
    std::cout << "Vector add3 completed on device - took "
        << (end - start).count() << " u-secs\n";
    return ((end - start).count());
} // 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.

```
size_t VectorAdd4(sycl::queue &q, const IntArray &a, const IntArray &b,
                  IntArray &sum, int iter) {
    sycl::range num_items{a.size()};

    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf(sum.data(), num_items);
    size_t num_groups = 1;
    size_t wg_size = 16;
    auto start = std::chrono::steady_clock::now();
    for (int i = 0; i < iter; i++) {
        q.submit([&](auto &h) {
            // Input accessors
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            // Output accessor
            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

            h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                           [=](sycl::nd_item<1> index)
                           [[intel::reqd_sub_group_size(16)]] {
                               // Manual unrolling
                               size_t loc_id = index.get_local_id();
                               for (size_t i = loc_id; i < mysize; i += 32) {
                                   int t1 = a_acc[i];
                                   int t2 = b_acc[i];
                                   int t3 = a_acc[i + 16];
                                   int t4 = b_acc[i + 16];
                                   sum_acc[i] = t1 + t2;
                                   sum_acc[i + 16] = t3 + t4;
                               }
                           });
        });
    }
    q.wait();
    auto end = std::chrono::steady_clock::now();
    std::cout << "Vector add4 completed on device - took "
           << (end - start).count() << " u-secs\n";
    return ((end - start).count());
} // end VectorAdd4
```

The following kernel has a `restrict` directive, which provides a 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.

```
size_t VectorAdd5(sycl::queue &q, const IntArray &a, const IntArray &b,
                  IntArray &sum, int iter) {
    sycl::range num_items{a.size()};

    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf(sum.data(), num_items);
    size_t num_groups = 1;
    size_t wg_size = 16;
    auto start = std::chrono::steady_clock::now();
```

```

for (int i = 0; i < iter; i++) {
    q.submit([&](auto &h) {
        // Input accessors
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        // Output accessor
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

        h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                      [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(
                        16)]] [[intel::kernel_args_restrict]] {
                        // compiler needs to hoist the loads
                        size_t loc_id = index.get_local_id();
                        for (size_t i = loc_id; i < mysize; i += 32) {
                            sum_acc[i] = a_acc[i] + b_acc[i];
                            sum_acc[i + 16] = a_acc[i + 16] + b_acc[i + 16];
                        }
                    });
    });
}
q.wait();
auto end = std::chrono::steady_clock::now();
std::cout << "Vector add5 completed on device - took "
       << (end - start).count() << " u-secs\n";
return ((end - start).count());
} // end VectorAdd5

```

## 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      SYCL 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 preceding synchronization events can be used to achieve the same functionality, but with significant differences in efficiency and performance.

- [Atomic Operations](#)
- [Local Barriers vs Global Atomics](#)

## Atomic Operations

Atomics allow multiple work-items for any cross work-item communication via memory. SYCL 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 factors affect the performance and legality of atomic operations

- Data types
- Local vs global address space
- Host, shared and device allocated USM

## Data Types in Atomic Operations

The following kernel shows the implementation of a reduction operation in SYCL 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 compared to other techniques used for reduction, such as blocking.

```
q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

    h.parallel_for(data_size, [=](auto index) {
        size_t glob_id = index[0];
        auto v = sycl::atomic_ref<int, sycl::memory_order::relaxed,
              sycl::memory_scope::device,
              sycl::access::address_space::global_space>(
            sum_acc[0]);
        v.fetch_add(buf_acc[glob_id]);
    });
});
```

If the data type of the vector is a float or a double as shown in the kernel below, the performance on certain accelerators is impaired due to lack of hardware support for float or double atomics. The following two kernels demonstrate how the time to execute an atomic add can vary drastically based on whether native atomics are supported.

```
// 
int VectorInt(sycl::queue &q, int iter) {
    VectorAllocator<int> alloc;
    AlignedVector<int> a(array_size, alloc);
    AlignedVector<int> b(array_size, alloc);

    InitializeArray<int>(a);
    InitializeArray<int>(b);
    sycl::range num_items{a.size()};
    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    auto start = std::chrono::steady_clock::now();
    for (int i = 0; i < iter; i++) {
        q.submit([&](sycl::handler &h) {
            // InputGt accessors
            sycl::accessor a_acc(a_buf, h, sycl::read_write);
            sycl::accessor b_acc(a_buf, h, sycl::read_only);

            h.parallel_for(num_items, [=](auto i) {
                auto v = sycl::atomic_ref<int, sycl::memory_order::relaxed,
                      sycl::memory_scope::device,
                      sycl::access::address_space::global_space>(
                    a_acc[0]);
                v += b_acc[i];
            });
        });
    }}
```

```

    }
    q.wait();
    auto end = std::chrono::steady_clock::now();
    std::cout << "Vector int completed on device - took " << (end - start).count()
        << " u-secs\n";
    return ((end - start).count());
}

```

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.

```

// 
int VectorDouble(sycl::queue &q, int iter) {
    VectorAllocator<double> alloc;
    AlignedVector<double> a(array_size, alloc);
    AlignedVector<double> b(array_size, alloc);

    InitializeArray<double>(a);
    InitializeArray<double>(b);
    sycl::range num_items{a.size()};
    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);

    auto start = std::chrono::steady_clock::now();
    for (int i = 0; i < iter; i++) {
        q.submit([&](sycl::handler &h) {
            // InputGt accessors
            sycl::accessor a_acc(a_buf, h, sycl::read_write);
            sycl::accessor b_acc(a_buf, h, sycl::read_only);

            h.parallel_for(num_items, [=](auto i) {
                auto v = sycl::atomic_ref<double, sycl::memory_order::relaxed,
                    sycl::memory_scope::device,
                    sycl::access::address_space::global_space>(
                        a_acc[0]);
                v += b_acc[i];
            });
        });
    }
    q.wait();
    auto end = std::chrono::steady_clock::now();
    std::cout << "Vector Double completed on device - took "
        << (end - start).count() << " u-secs\n";
    return ((end - start).count());
}

```

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.

### VTune Dynamic Instruction

Task		Data Tra...	GPU Instructions Executed by Instruction Type			
Time	Instance Count	Size	Control Flow	Send	Int32 & SP Float	Others
73ms	10	0 B	66,783,238	66,865,158	163,840	133,620
51ms	10	0 B	64,047,824	32,085,352	64,252,624	133,620
21ms	20	0 B	0	81,920	389,120	133,620
19ms	4	640 KB				

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

Here is the assembly code for our VectorInt kernel.

### VTune Atomic Integer

Addr...	So...	Assembly	GPU Instructions Executed
<b>0</b>		<b>Block 1:</b>	
0	53	(W) mov (8 M0) r3.0<1>:ud r0.0<1;1,0>:ud	20,
0x10	53	(W) or (1 M0) cr0.0<1>:ud cr0.0<0;1,0>:ud 0x4C0:uw {Swi}	20,
0x20		(W) mul (1 M0) r10.0<1>:d r11.0<0;1,0>:d r3.1<0;1,0>:d	20,
0x28		(W) mov (16 M0) r11.0<1>:d 0:w	20,
0x38		(W) shl (1 M0) r10.2<1>:d r9.6<0;1,0>:d 2:w	20,
0x48		(W) mov (16 M16) r4.0<1>:d 0:w	20,
0x58		(W) shl (1 M0) r10.1<1>:d r9.0<0;1,0>:d 2:w	20,
0x68		add (16 M0) r8.0<1>:d r10.0<0;1,0>:d r1.0<16;16,1>:uw	20,
0x78		add (16 M16) r4.0<1>:d r10.0<0;1,0>:d r2.0<16;16,1>:uw	20,
0x88		(W) add (1 M0) r10.0<1>:d r10.2<0;1,0>:d r10.5<0;1,0>:d	20,
0x98		(W) mov (8 M0) r112.0<1>:ud r3.0<8;8,1>:ud {Compacted}	20,
0xa0		(W) add (1 M0) r14.0<1>:d r10.1<0;1,0>:d r10.4<0;1,0>:d	20,
0xb0		add (16 M0) r8.0<1>:d r8.0<8;8,1>:d r7.0<0;1,0>:d {Compacted}	20,
0xb8		add (16 M16) r4.0<1>:d r4.0<8;8,1>:d r7.0<0;1,0>:d	20,
0xc8		shl (16 M0) r8.0<1>:d r8.0<8;8,1>:d 2:w	20,
0xd8		shl (16 M16) r4.0<1>:d r4.0<8;8,1>:d 2:w	20,
0xe8		add (16 M0) r8.0<1>:d r10.0<0;1,0>:d r8.0<8;8,1>:d {Compacted}	20,
0xf0		add (16 M16) r4.0<1>:d r10.0<0;1,0>:d r4.0<8;8,1>:d	20,
0x100		send (16 M0) r6:w r8 0xC 0x04205E01 [Data Cache Data P]	20,
0x110		send (16 M16) r4:w r4 0xC 0x04205E01 [Data Cache Data P]	20,
0x120		mov (16 M0) r11.0<1>:d r6.0<8;8,1>:d {Compacted}	20,
0x128		(W) add (16 M0) r4.0<1>:d r11.0<8;8,1>:d r4.0<8;8,1>:d	20,
0x130		(W) add (8 M0) r2.0<1>:d r4.0<8;8,1>:d r5.0<8;8,1>:d {Compacted}	20,
0x138		(W) add (4 M0) r2.0<1>:d r2.0<4;4,1>:d r2.4<4;4,1>:d {Compacted}	20,
0x140		(W) add (2 M0) r2.0<1>:d r2.0<2;2,1>:d r2.2<2;2,1>:d	20,
0x150		(W) add (1 M0) r2.0<1>:d r2.0<0;1,0>:d r2.1<0;1,0>:d {Compacted}	20,
0x158		(W) sends (1 M0) null:ud r14 r2 0x4C 0x020009700 [Data Cache D	20,
0x168		(W) send (8 M0) null r112 0x27 0x02000010 {EOT} [Thread]	20,

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

### Vtune Atomic Double

Addr...	Sour...	Assembly	GPU Instructions Executed
0x50		(W) shl (1 M0) r6.0<1>:q r6.0<0;1,0>:q 3:w	20,4
0x60		mov (8 M8) r16.0<1>:q 0:w	20,4
0x70		add (16 M0) r8.0<1>:d r8.0<0;1,0>:d r1.0<16;16,1>:uw	20,4
0x80		(W) add (1 M0) r6.2<1>:d r6.2<0;1,0>:d r7.4<0;1,0>:d	20,4
0x90		(W) add (1 M0) r26.0<1>:q r6.0<0;1,0>:q r5.0<0;1,0>:q	20,4
0xa0		add (16 M0) r8.0<1>:d r8.0<8;8,1>:d r4.0<0;1,0>:d {Compacted}	20,4
0xa8		mov (8 M0) r10.0<1>:uq r26.0<0;1,0>:uq	20,4
0xb8		mov (8 M8) r12.0<1>:uq r26.0<0;1,0>:uq	20,4
0xc8		shl (16 M0) r8.0<1>:d r8.0<8;8,1>:d 3:w	20,4
0xd8		add (16 M0) r8.0<1>:d r6.2<0;1,0>:d r8.0<8;8,1>:d {Compacted}	20,4
0xe0		send (16 M0) r4:w r8 0xC 0x04405C01 [Data Cache Data P]	20,4
0xf0		sends (8 M0) r8:uq r10 r14 0x8C 0x0424B2FF [Data Cache	20,4
0x100		sends (8 M8) r10:uq r12 r16 0x8C 0x0424B2FF [Data Cache	20,4
0x110		mov (8 M0) r23.0<2>:d r4.0<8;8,1>:d	20,4
0x120		mov (8 M8) r21.0<2>:d r5.0<8;8,1>:d	20,4
0x130		mov (8 M0) r23.1<2>:d r6.0<8;8,1>:d	20,4
0x140		mov (8 M8) r21.1<2>:d r7.0<8;8,1>:d	20,4
0x150		mov (8 M0) r19.0<1>:uq r8.0<4;4,1>:uq	20,4
0x160		mov (8 M8) r17.0<1>:uq r10.0<4;4,1>:uq	20,4
<b>0x170</b>		<b>Block 2:</b>	
0x170		add (8 M0) r11.0<1>:df r23.0<4;4,1>:df r19.0<4;4,1>:df	33,391,6
0x180		mov (8 M0) r5.0<1>:uq r26.0<0;1,0>:uq	33,391,6
0x190		mov (8 M0) r9.0<1>:q r19.0<4;4,1>:q	33,391,6
0x1a0		add (8 M8) r15.0<1>:df r21.0<4;4,1>:df r17.0<4;4,1>:df	33,391,6
0x1b0		mov (8 M8) r7.0<1>:uq r26.0<0;1,0>:uq	33,391,6
0x1c0		mov (8 M8) r13.0<1>:q r17.0<4;4,1>:q	33,391,6
0x1d0		sends (8 M0) r3:uq r5 r9 0x10C 0x0424BEFF [Data Cache]	33,391,6
0x1e0		sends (8 M8) r5:uq r7 r13 0x10C 0x0424BEFF [Data Cache]	33,391,6
0x1f0		mov (8 M0) r7.0<1>:uq r3.0<4;4,1>:uq	33,391,6
0x200		mov (8 M8) r3.0<1>:uq r5.0<4;4,1>:uq	33,391,6
0x210		cmp (8 M0) (eq)f0.0 null<1>:q r7.0<4;4,1>:q r19.0<4;4,1>:q	33,391,6
0x220		cmp (8 M8) (eq)f0.0 null<1>:q r3.0<4;4,1>:q r17.0<4;4,1>:q	33,391,6
0x230		<u>(f0.0) break (16 M0) bb_4 bb_4</u>	33,391,6
<b>0x240</b>		<b>Block 3:</b>	
0x240		mov (8 M0) r19.0<1>:q r7.0<4;4,1>:q	33,371,1
0x250		mov (8 M8) r17.0<1>:q r3.0<4;4,1>:q	33,371,1
<b>0x260</b>		<b>Block 4:</b>	
0x260		<u>while (16 M0) bb_2</u>	33,391,6
<b>0x270</b>		<b>Block 5:</b>	
0x270	80	(W) mov (8 M0) r112.0<1>:ud r2.0<8;8,1>:ud {Compacted}	20,4
0x278		(W) s VTune Profiler session timed out. All open results were closed and a new session was created.	

The Intel® Advisor tool has a recommendation pane that provides insights on how to improve the performance of GPU kernels.

## Advisor Recommendation Pane

Compute Task	Elapsed Time	Performance Issues	GPU Compute Performance			Work Size
			FLOAT Operations		INT Operations	
[Outside any task]	2.337s		0.000 GOp	0.000 GOp/s	AI 0.000	Global
zeCommandListAppendMemoryCopy	0.001s		0.000 GOp	0.000 GOp/s	AI 0.000	Local
comp_flux	0.545s	GRF spilling present 1 more detected	26.466 GOp	48.576 GOp/s	AI 0.924	192 x 506
			6.149 GOp	11.287 GOp/s	AI 0.215	192 x 1

One of the recommendations that Intel® Advisor provides is “Inefficient atomics present”. When atomics are not natively supported in hardware, they are emulated. This can be detected and Intel® Advisor gives advice on possible solutions.

## Advisor Inefficient Atomics

Compute Task	Elapsed Time	Performance Issues	GPU Compute Performance			Work Size
			FLOAT Operations		INT Operations	
[Outside any task]	2.337s		0.000 GOp	0.000 GOp/s	AI 0.000	Global
zeCommandListAppendMemoryCopy	0.001s		0.000 GOp	0.000 GOp/s	AI 0.000	Local
comp_flux	0.545s	GRF spilling present 1 more detected	26.466 GOp	48.576 GOp/s	AI 0.924	192 x 506
			6.149 GOp	11.287 GOp/s	AI 0.215	192 x 1

## Atomic Operations in Global and Local Address Spaces

The standard C++ memory model assumes that applications execute on a single device with a single address space. Neither of these assumptions holds for SYCL 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.

```
#include <CL/sycl.hpp>
#include <iostream>
int main() {
    constexpr int N = 256 * 256;
    constexpr int M = 512;
    int total = 0;
    int *a = static_cast<int *>(malloc(sizeof(int) * N));
    for (int i = 0; i < N; i++)
        a[i] = 1;
    sycl::queue q({sycl::property::queue::enable_profiling()});
    sycl::buffer<int> buf(&total, 1);
    sycl::buffer<int> bufa(a, N);
    auto e = q.submit([&](sycl::handler &h) {
        sycl::accessor acc(buf, h);
        sycl::accessor acc_a(bufa, h, sycl::read_only);
        h.parallel_for(sycl::nd_range<1>(N, M), [=](auto it) {
            auto i = it.get_global_id();
            sycl::atomic_ref<int, sycl::memory_order_relaxed,
                sycl::memory_scope_device,
                sycl::access::address_space::global_space>
                atomic_op(acc[0]);
            atomic_op += acc_a[i];
        });
    });
    sycl::host_accessor h_a(buf);
    std::cout << "Reduction Sum : " << h_a[0] << "\n";
    std::cout
        << "Kernel Execution Time of Global Atomics Ref: "
        << e.get_profiling_info<sycl::info::event_profiling::command_end>() -
        e.get_profiling_info<sycl::info::event_profiling::command_start>()
        << "\n";
    return 0;
}
```

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

```
#include <CL/sycl.hpp>
#include <iostream>
int main() {
    constexpr int N = 256 * 256;
    constexpr int M = 512;
    constexpr int NUM_WG = N / M;
    int total = 0;
    int *a = static_cast<int *>(malloc(sizeof(int) * N));
    for (int i = 0; i < N; i++)
        a[i] = 1;
    sycl::queue q({sycl::property::queue::enable_profiling()});
    sycl::buffer<int> global(&total, 1);
    sycl::buffer<int> bufa(a, N);
```

```

auto e1 = q.submit([&](sycl::handler &h) {
    sycl::accessor b(global, h);
    sycl::accessor acc_a(bufa, h, sycl::read_only);
    auto acc = sycl::local_accessor<int, 1>(NUM_WG, h);
    h.parallel_for(sycl::nd_range<1>(N, M), [=](auto it) {
        auto i = it.get_global_id(0);
        auto group_id = it.get_group(0);
        sycl::atomic_ref<int, sycl::memory_order_relaxed,
                      sycl::memory_scope_device,
                      sycl::access::address_space::local_space>
            atomic_op(acc[group_id]);
        sycl::atomic_ref<int, sycl::memory_order_relaxed,
                      sycl::memory_scope_device,
                      sycl::access::address_space::global_space>
            atomic_op_global(b[0]);
        atomic_op += acc_a[i];
        it.barrier(sycl::access::fence_space::local_space);
        if (it.get_local_id() == 0)
            atomic_op_global += acc[group_id];
    });
});
sycl::host_accessor h_global(global);
std::cout << "Reduction Sum : " << h_global[0] << "\n";
int total_time =
    (e1.get_profiling_info<sycl::info::event_profiling::command_end>() -
     e1.get_profiling_info<sycl::info::event_profiling::command_start>());
std::cout << "Kernel Execution Time of Local Atomics : " << total_time
     << "\n";
return 0;
}

```

## Atomic Operations on USM Data

On discrete GPU,

- Atomic operations on host allocated USM (`sycl::malloc_host`) are not supported.
- Concurrent accesses from host and device to shared USM location (`sycl::malloc_shared`) are not supported.

We recommend using device allocated USM (`sycl::malloc_device`) memory for atomics and device algorithms with atomic operations.

## Memory Scope of Atomic Operations

Memory scope of atomic operations allows you to avoid data races.

To learn more about the `sycl::memory_scope` parameter see the [Memory scope](#) section of the SYCL specification.

To learn more about the memory model, see the [Memory Model and Atomics chapter](#) of the DPC++ book.

## 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.

```
q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
    h.parallel_for(data_size, [=](auto index) {
        int glob_id = index[0];
        if (glob_id == 0) {
            int sum = 0;
            for (int i = 0; i < N; i++)
                sum += buf_acc[i];
            sum_acc[0] = sum;
        }
    });
});
```

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.

```
q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

    h.parallel_for(data_size, [=](auto index) {
        size_t glob_id = index[0];
        auto v = sycl::atomic_ref<int, sycl::memory_order::relaxed,
              sycl::memory_scope::device,
              sycl::access::address_space::global_space>(
            sum_acc[0]);
        v.fetch_add(buf_acc[glob_id]);
    });
});
```

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.

```
Timer timer;
q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
    sycl::local_accessor<int, 1> scratch(work_group_size, h);
    h.parallel_for(sycl::nd_range<1>{work_group_size, work_group_size},
                  [=](sycl::nd_item<1> item) {
                      size_t loc_id = item.get_local_id();
                      int sum = 0;
                      for (int i = loc_id; i < data_size; i += num_work_items)
                          sum += buf_acc[i];
                      scratch[loc_id] = sum;
                      for (int i = work_group_size / 2; i > 0; i >>= 1) {
                          item.barrier(sycl::access::fence_space::local_space);
                          if (loc_id < i)
                              scratch[loc_id] += scratch[loc_id + i];
                      }
                  });
});
```

```

        if (loc_id == 0)
            sum_acc[0] = scratch[0];
    });
});

```

The performance of these three kernels varies quite a bit among various platforms, and developers need to pick the technique that suits their application and hardware.

## Considerations for Selecting Work-Group Size

In SYCL you 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 the same work-group may have access to hardware resources like shared local memory and hardware synchronization capabilities that will allow them to run and communicate more efficiently than work-items across work-groups. So in general you should 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 lets you 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.

```

void reduction(sycl::queue &q, std::vector<int> &data, std::vector<int> &flush,
               int iter, int work_group_size) {
    const size_t data_size = data.size();
    const size_t flush_size = flush.size();
    int sum = 0;

    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
    // int vec_size =
    // q.get_device().get_info<sycl::info::device::native_vector_width_int>();
    int num_work_items = data_size / work_group_size;
    sycl::buffer<int> buf(data.data(), data_size, props);
    sycl::buffer<int> flush_buf(flush.data(), flush_size, props);
    sycl::buffer<int> sum_buf(&sum, 1, props);

    init_data(q, buf, data_size);

    double elapsed = 0;
    for (int i = 0; i < iter; i++) {
        q.submit([&](auto &h) {
            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

            h.parallel_for(1, [=](auto index) { sum_acc[index] = 0; });

        });
        // flush the cache
        q.submit([&](auto &h) {
            sycl::accessor flush_acc(flush_buf, h, sycl::write_only, sycl::no_init);
            h.parallel_for(flush_size, [=](auto index) { flush_acc[index] = 1; });

        });

        Timer timer;
        // reductionMapToHWVector main begin
        q.submit([&](auto &h) {
            sycl::accessor buf_acc(buf, h, sycl::read_only);
            sycl::local_accessor<int, 1> scratch(work_group_size, h);
            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

```

```

    h.parallel_for(
        sycl::nd_range<1>(num_work_items, work_group_size),
        [=](sycl::nd_item<1> item) [[intel::reqd_sub_group_size(16)]] {
            auto v =
                sycl::atomic_ref<int, sycl::memory_order::relaxed,
                sycl::memory_scope::device,
                sycl::access::address_space::global_space>(
                    sum_acc[0]);
            int sum = 0;
            int glob_id = item.get_global_id();
            int loc_id = item.get_local_id();
            for (unsigned int i = glob_id; i < data_size; i += num_work_items)
                sum += buf_acc[i];
            scratch[loc_id] = sum;

            for (int i = work_group_size / 2; i > 0; i >>= 1) {
                item.barrier(sycl::access::fence_space::local_space);
                if (loc_id < i)
                    scratch[loc_id] += scratch[loc_id + i];
            }

            if (loc_id == 0)
                v.fetch_add(scratch[0]);
        });
    q.wait();
    elapsed += timer.Elapsed();
    sycl::host_accessor h_acc(sum_buf);
    sum = h_acc[0];
}
elapsed = elapsed / iter;
std::string msg = "with work-groups=" + std::to_string(work_group_size);
check_result(elapsed, msg, sum);
} // reduction end

```

In the code below, the above kernel is called with two different values: **2\*vec-size** and the 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.

```

int vec_size = 16;
int work_group_size = vec_size;
reduction(q, data, extra, 16, work_group_size);
work_group_size =
    q.get_device().get_info<sycl::info::device::max_work_group_size>();
reduction(q, data, extra, 16, work_group_size);

```

In situations where there are no barriers nor 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.

```

void vec_copy(sycl::queue &q, std::vector<int> &src, std::vector<int> &dst,
              std::vector<int> &flush, int iter, int work_group_size) {
    const size_t data_size = src.size();
    const size_t flush_size = flush.size();

    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
    int num_work_items = data_size;
    double elapsed = 0;
{
    sycl::buffer<int> src_buf(src.data(), data_size, props);

```

```

sycl::buffer<int> dst_buf(dst.data(), data_size, props);
sycl::buffer<int> flush_buf(flush.data(), flush_size, props);

for (int i = 0; i < iter; i++) {
    // flush the cache
    q.submit([&](auto &h) {
        sycl::accessor flush_acc(flush_buf, h, sycl::write_only, sycl::no_init);
        h.parallel_for(flush_size, [=](auto index) { flush_acc[index] = 1; });
    });

    Timer timer;
    q.submit([&](auto &h) {
        sycl::accessor src_acc(src_buf, h, sycl::read_only);
        sycl::accessor dst_acc(dst_buf, h, sycl::write_only, sycl::no_init);

        h.parallel_for(sycl::nd_range<1>(num_work_items, work_group_size),
                      [=](sycl::nd_item<1> item)
                          [[intel::reqd_sub_group_size(16)]] {
                            int glob_id = item.get_global_id();
                            dst_acc[glob_id] = src_acc[glob_id];
                        });
    });
    q.wait();
    elapsed += timer.Elapsed();
}
}

elapsed = elapsed / iter;
std::string msg = "with work-group-size=" + std::to_string(work_group_size);
check_result(elapsed, msg, dst);
} // 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 the scheduling and resourcing point of view when there are no barriers nor shared memory in the work-groups.

```

int vec_size = 16;
int work_group_size = vec_size;
vec_copy(q, src, dst, extra, 16, work_group_size);
work_group_size = 2 * vec_size;
vec_copy(q, src, dst, extra, 16, work_group_size);
work_group_size = 4 * vec_size;
vec_copy(q, src, dst, extra, 16, work_group_size);
work_group_size = 8 * vec_size;
vec_copy(q, src, dst, extra, 16, work_group_size);
work_group_size = 16 * vec_size;
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 you may see a big performance difference when the number of sub-groups is 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 two sub-groups at a time.

## Tuning Kernels with Local and Global Work-group Sizes in OpenMP Offload Mode

The approach of tuning kernel performance on accelerator devices as explained above for SYCL, is also applicable for implementations via OpenMP in offload mode. It is possible to customize an application kernel along with the use of OpenMP directives to make use of appropriate work-group sizes. However, this may require significant modifications to the code. The OpenMP implementation provides an option to custom tune kernels with the use of environment variables. The local and global work-group sizes for kernels in an app can be customized with the the use of two environment variables – **OMP\_THREAD\_LIMIT** and **OMP\_NUM\_TEAMS** help in setting up the local work-group size (**LWS**) and global work-group size (**GWS**) as shown below:

```
LWS = OMP_THREAD_LIMIT
GWS = OMP_THREAD_LIMIT * OMP_NUM_TEAMS
```

With the help of following reduction kernel example, we show the use of **LWS** and **GWS** in tuning kernel performance on accelerator device.

```
int N = 2048;

double* A = make_array(N, 0.8);
double* B = make_array(N, 0.65);
double* C = make_array(N*N, 2.5);
if ((A == NULL) || (B == NULL) || (C == NULL))
    exit(1);

int i, j;
double val = 0.0;

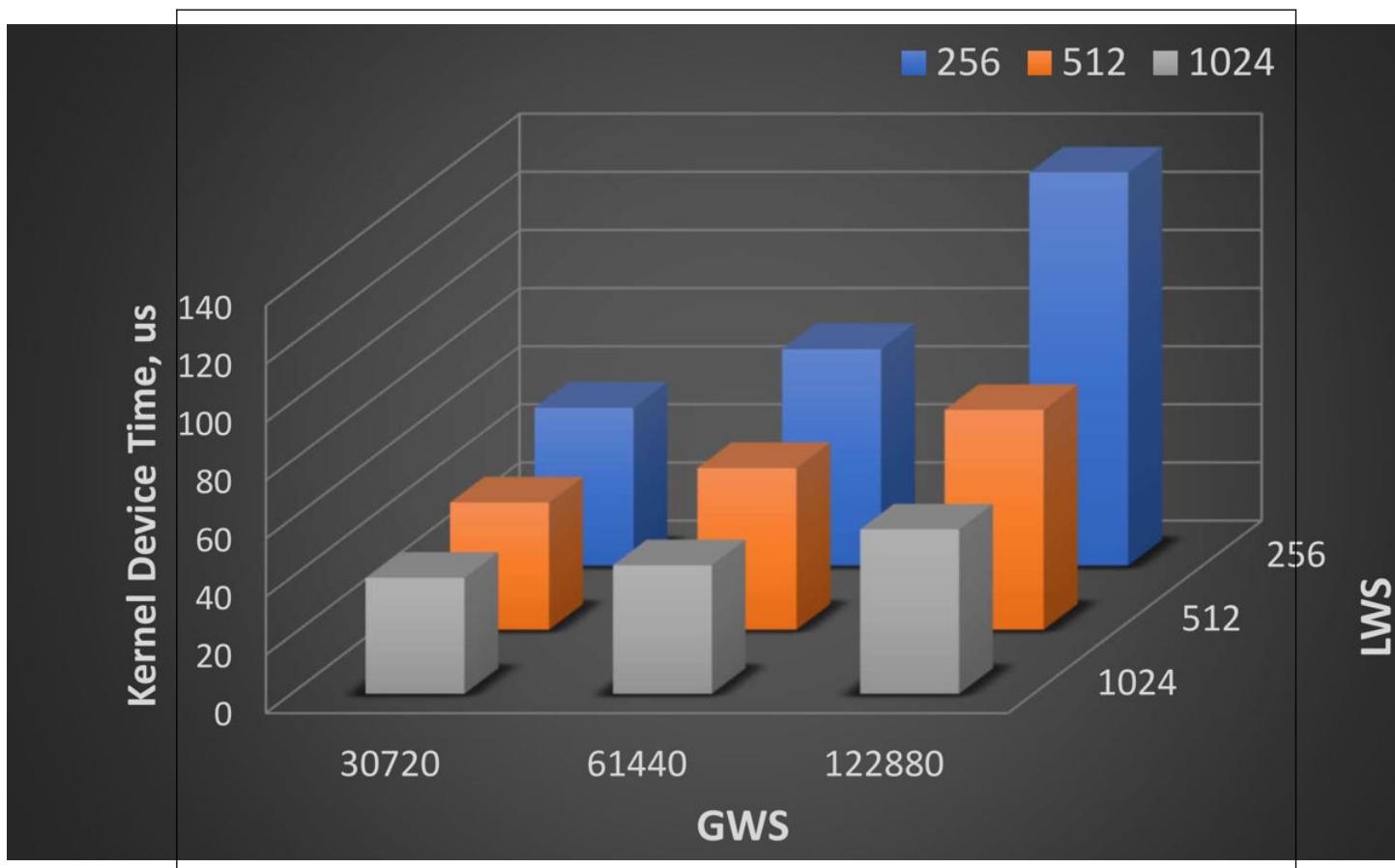
#pragma omp target map(to:A[0:N],B[0:N],C[0:N*N]) map(tofrom:val)
{

#pragma omp teams distribute parallel for collapse(2) reduction(+ : val)
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            val += C[i * N + j] * A[i] * B[j];
        }
    }
}

printf("val = %f10.3\n", val);

free(A);
free(B);
free(C);
```

e.g. by choosing **OMP\_THREAD\_LIMIT = 1024** and **OMP\_NUM\_TEAMS = 120**, the **LWS** and **GWS** parameters are set to **1024** and **122880**, respectively.



The figure above shows that the best performance for this kernel comes with **LWS = 1024** and **GWS = 30720** which corresponds to **OMP\_THREAD\_LIMIT = 1024** and **OMP\_NUM\_TEAMS = 30**. These environment variables will set the **LWS** and **GWS** values to a fixed numbers for all kernels offloaded via OpenMP. However, these environment variables will not affect the **LWS** and **GWS** used by highly tuned library kernels like OneMKL.

## Prefetch

Access to global memory is one of the most common reasons of execution stall. When application stops and waits for data to reach local registers, it is called memory latency. This is especially costly when access misses cache and must reach HBM memory. GPU tries to hide memory latencies by reordering instructions to execute (out-of-order execution) and switching threads in Vector Engine. If these techniques are not enough and application keeps stalling, yet memory bandwidth is not fully utilized, application developer might use prefetch function to provide hints when to load data into local cache before expected use. When correctly used, next access to memory will successfully hit cache, shortening distance for data to reach registers, lowering memory latency. Prefetch works on Intel® Data Center GPU Max Series and later products.

Prefetch is an asynchronous operation. When submitting prefetch, application does not wait for data to reach cache and keeps running. This means prefetch must be submitted way ahead of expected access, so there are enough instructions in between to hide the transfer of data from HBM memory to local cache.

One common use case for prefetch are memory accesses inside loop. In typical use case, compiler tries to apply loop unrolling optimization. This technique gives scheduler more instructions to work with, intertwining loop iterations and hiding memory latencies in the process. If this technique can't be applied (unknown loop boundaries; high register pressure in larger kernels), prefetch can be used to lower memory latencies by caching data for future iterations.

Intel® VTune™ Profiler can be used to search for chances to use prefetch. If memory bandwidth is not fully utilized and number of scoreboard stalls is high (execution waits for data to load), there is a good chance that correctly inserted prefetch can speed up execution.

Examples in this section create artificial scenario to simulate case where GPU is unable to hide latencies. First, we run only one work-group, submitting hardware threads only to one Vector Engine. Then, we use pragma to disable loop unrolling. This limits number of available instructions to reschedule in order to hide memory latency.

In first example, kernel reduces array `values` and saves result to `results`. `values` uses indirect indexing, with indices loaded from array `indices`. Because indices are unknown at compilation time, compiler can't optimize access to array `values`. Since loop unrolling is disabled, GPU can't schedule other work between accessing `indices` and `values` and execution stalls until access to `indices` finishes.

```
auto e = q.submit([&] (auto &h) {
    h.parallel_for(
        sycl::nd_range(sycl::range{GLOBAL_WORK_SIZE},
                      sycl::range{GLOBAL_WORK_SIZE}),
        [=] (sycl::nd_item<1> it) [[intel::reqd_sub_group_size(SIMD_SIZE)]] {
            const int i = it.get_global_linear_id();
            const int lane = it.get_sub_group().get_local_id()[0];
            const int subgroup = it.get_sub_group().get_group_id()[0];

            // Index starting position
            int *indexCurrent = indices + lane + subgroup * ITEMS_PER_SUBGROUP;

            float dx = 0.0f;
#pragma unroll(0)
            for (int j = 0; j < ITEMS_PER_LANE; ++j) {
                // Load index for indirect addressing
                int index = *indexCurrent;
                // Waits for load to finish, high latency
                float v = values[index];
                for (int k = 0; k < 64; ++k)
                    dx += sqrtf(v + k);

                indexCurrent += SIMD_SIZE;
            }

            result[i] = dx;
        });
    });
});
```

In second example, before accessing current value from `indices`, kernel calls `sycl::global_ptr::prefetch` to submit prefetch request of next index, potentially speeding up access on next iteration. Data is not prefetched for first iterations, as there is not enough instructions to hide latency.

```
auto e = q.submit([&] (auto &h) {
    h.parallel_for(
        sycl::nd_range(sycl::range{GLOBAL_WORK_SIZE},
                      sycl::range{GLOBAL_WORK_SIZE}),
        [=] (sycl::nd_item<1> it) [[intel::reqd_sub_group_size(SIMD_SIZE)]] {
            const int i = it.get_global_linear_id();
            const int lane = it.get_sub_group().get_local_id()[0];
            const int subgroup = it.get_sub_group().get_group_id()[0];

            // Index starting position
            int *indexCurrent = indices + lane + subgroup * ITEMS_PER_SUBGROUP;
            int *indexNext = indexCurrent + SIMD_SIZE;

            float dx = 0.0f;
#pragma unroll(0)
            for (int j = 0; j < ITEMS_PER_LANE; ++j) {
                // Prefetch next index to cache
```

```

        sycl::global_ptr<int>(indexNext).prefetch(1);
        // Load index, might be cached
        int index = *indexCurrent;
        // Latency might be reduced if index was cached
        float v = values[index];
        for (int k = 0; k < 64; ++k)
            dx += sqrtf(v + k);

        indexCurrent = indexNext;
        indexNext += SIMD_SIZE;
    }

    result[i] = dx;
});
});
}
);

```

`sycl::global_ptr::prefetch` takes one argument: number of continuous elements to prefetch. Prefetch is available only for global address space. At the moment this function has multiple limitations that should be taken into consideration:

1. Compiler caches at max 32 bytes total per work item for each prefetch call (even for larger data types).
2. Compiler prefers for argument (number of continuous elements to prefetch) to be constant at compilation time.
3. There is no control over to what cache data will be fetched. Current implementation loads data only to L3 cache.

Lack of cache control heavily limits scenarios where `sycl::global_ptr::prefetch` can give meaningful performance gains. When running second example, user might even see performance loss. Prefetch gives best results if data is loaded to L1 cache. At the moment on Intel® Data Center GPU Max Series this is possible only with dedicated functions. First, include these declarations in your code:

```

#include <CL/sycl.hpp>

enum LSC_LDCC {
    LSC_LDCC_DEFAULT,
    LSC_LDCC_L1UC_L3UC, // 1 // Override to L1 uncached and L3 uncached
    LSC_LDCC_L1UC_L3C, // 2 // Override to L1 uncached and L3 cached
    LSC_LDCC_L1C_L3UC, // 3 // Override to L1 cached and L3 uncached
    LSC_LDCC_L1C_L3C, // 4 // Override to L1 cached and L3 cached
    LSC_LDCC_L1S_L3UC, // 5 // Override to L1 streaming load and L3 uncached
    LSC_LDCC_L1S_L3C, // 6 // Override to L1 streaming load and L3 cached
    LSC_LDCC_L1IAR_L3C, // 7 // Override to L1 invalidate-after-read, and L3
                           // cached
};

extern "C" {
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_uchar(const __attribute__((opencl_global))
                                         uint8_t *base,
                                         int immElemOff, enum LSC_LDCC cacheOpt);

SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_ushort(const __attribute__((opencl_global))
                                         uint16_t *base,
                                         int immElemOff, enum LSC_LDCC cacheOpt);

SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_uint(const __attribute__((opencl_global))
                                         uint32_t *base,
                                         int immElemOff, enum LSC_LDCC cacheOpt);

SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_uint2(const __attribute__((opencl_global)))

```

```

        uint32_t *base,
        int immElemOff, enum LSC_LDCC cacheOpt);
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_uint3(const __attribute__((opencl_global))
        uint32_t *base,
        int immElemOff, enum LSC_LDCC cacheOpt);
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_uint4(const __attribute__((opencl_global))
        uint32_t *base,
        int immElemOff, enum LSC_LDCC cacheOpt);
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_uint8(const __attribute__((opencl_global))
        uint32_t *base,
        int immElemOff, enum LSC_LDCC cacheOpt);
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_ulong(const __attribute__((opencl_global))
        uint64_t *base,
        int immElemOff, enum LSC_LDCC cacheOpt);
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_ulong2(const __attribute__((opencl_global))
        uint64_t *base,
        int immElemOff, enum LSC_LDCC cacheOpt);
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_ulong3(const __attribute__((opencl_global))
        uint64_t *base,
        int immElemOff, enum LSC_LDCC cacheOpt);
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_ulong4(const __attribute__((opencl_global))
        uint64_t *base,
        int immElemOff, enum LSC_LDCC cacheOpt);
SYCL_EXTERNAL void
__builtin_IB_lsc_prefetch_global_ulong8(const __attribute__((opencl_global))
        uint64_t *base,
        int immElemOff, enum LSC_LDCC cacheOpt);
}

```

Argument `cacheOpt` selects target cache. `sycl::global_ptr::prefetch` maps to `LSC_LDCC_L1UC_L3C`. In most scenarios value `LSC_LDCC_L1C_L3C` is expected to give the best results. Argument `immElemOff` can be used to offset base pointer, but in most cases base pointer is set to correct address to fetch, and `immElemOff` is set to 0.

Prefetch to L1 does not support safe out-of-bounds access. When using this type of prefetch, application must take full ownership of bounds checking, otherwise any out-of-bounds access will end with undefined behavior.

Third example shows L1 prefetch in action, including bounds checking:

```

auto e = q.submit([&] (auto &h) {
    h.parallel_for(
        sycl::nd_range(sycl::range,
                      sycl::range),
        [=] (sycl::nd_item<1> it) [[intel::reqd_sub_group_size(SIMD_SIZE)]] {
            const int i = it.get_global_linear_id();
            const int lane = it.get_sub_group().get_local_id()[0];
            const int subgroup = it.get_sub_group().get_group_id()[0];

            // Index starting position
            int *indexCurrent = indices + lane + subgroup * ITEMS_PER_SUBGROUP;
            int *indexNext = indexCurrent + SIMD_SIZE;

```

```

        float dx = 0.0f;
#pragma unroll(0)
        for (int j = 0; j < ITEMS_PER_LANE; ++j) {
            // Prefetch next index to cache
#ifndef __SYCL_DEVICE_ONLY__
            if (j < ITEMS_PER_LANE - 1)
                __builtin_IB_lsc_prefetch_global_uint(
                    (const __attribute__((opencl_global)) uint32_t *)indexNext, 0,
                    LSC_LDCC_L1C_L3C);
#endif
        }
        // Load index, might be cached
        int index = *indexCurrent;
        // Latency might be reduced if index was cached
        float v = values[index];
        for (int k = 0; k < 64; ++k)
            dx += sqrtf(v + k);

        indexCurrent = indexNext;
        indexNext += SIMD_SIZE;
    }

    result[i] = dx;
});
});

```

When manually fetching data to cache, remember that memory fences might flush cache, so it might be required to repeat prefetch to refill it back.

Presented examples give only an idea how to use prefetch and don't represent real-life use cases. In short kernels like these normal optimization methods like loop unrolling are enough to hide memory latencies. Prefetch hints are expected to give performance gains in larger kernels, where ordinary optimization methods applied by compiler might not be sufficient.

## 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, and minimum. A serial summation reduction is shown below:

```

for (int it = 0; it < iter; it++) {
    sum = 0;
    for (size_t i = 0; i < data_size; ++i) {
        sum += data[i];
    }
}

```

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 update this variable using an atomic operation:

```

q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

    h.parallel_for(data_size, [=](auto index) {
        size_t glob_id = index[0];
        auto v = sycl::atomic_ref<int, sycl::memory_order::relaxed,
              sycl::memory_scope::device,

```

```

        sycl::access::address_space::global_space>(
    sum_acc[0]);
    v.fetch_add(buf_acc[glob_id]);
);
}

```

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:

```

q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
    h.parallel_for(num_processing_elements, [=](auto index) {
        size_t glob_id = index[0];
        size_t start = glob_id * BATCH;
        size_t end = (glob_id + 1) * BATCH;
        if (end > N)
            end = N;
        int sum = 0;
        for (size_t i = start; i < end; ++i)
            sum += buf_acc[i];
        accum_acc[glob_id] = sum;
    });
});
}

```

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:

```

q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
    h.parallel_for(num_work_items, [=](auto index) {
        size_t glob_id = index[0];
        int sum = 0;
        for (size_t i = glob_id; i < data_size; i += num_work_items)
            sum += buf_acc[i];
        accum_acc[glob_id] = sum;
    });
});
}

```

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 Vector Engine (VE) or Execution Unit (EU) on the machine. So the above code where the number of threads is equal to the number of VEs does not utilize all the thread contexts. Even in the case of CPUs that 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:

#### Number of thread contexts available by device

	<b>VEs</b>	<b>Threads per VE   Total threads</b>
KBL	24	7   $24 \times 7 = 168$
TGL	96	7   $96 \times 7 = 672$

The code below shows a 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, you can scale the number of work-items you create for splitting the work among them.

```
q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
    h.parallel_for(num_work_items, [=](auto index) {
        size_t glob_id = index[0];
        int sum = 0;
        for (size_t i = glob_id; i < data_size; i += num_work_items)
            sum += buf_acc[i];
        accum_acc[glob_id] = sum;
    });
});
```

One popular way of doing a 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, because a large number of kernels are launched. Also, the kernel as written below is not very efficient because the number of threads doing actual work reduces exponentially each time through the loop.

```
q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
    sycl::local_accessor<int, 1> scratch(work_group_size, h);

    h.parallel_for(sycl::nd_range<1>(num_work_items, work_group_size),
                  [=](sycl::nd_item<1> item) {
                      size_t global_id = item.get_global_id(0);
                      int local_id = item.get_local_id(0);
                      int group_id = item.get_group(0);

                      if (global_id < data_size)
                          scratch[local_id] = buf_acc[global_id];
                      else
                          scratch[local_id] = 0;

                      // Do a tree reduction on items in work-group
                      for (int i = work_group_size / 2; i > 0; i >>= 1) {
                          item.barrier(sycl::access::fence_space::local_space);
                          if (local_id < i)
                              scratch[local_id] += scratch[local_id + i];
                      }

                      if (local_id == 0)
                          accum_acc[group_id] = scratch[0];
                  });
});
```

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 improve 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.

```

q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor accum_acc(accum1_buf, h, sycl::write_only, sycl::no_init);
    sycl::local_accessor<int, 1> scratch(work_group_size, h);

    h.parallel_for(sycl::nd_range<1>(num_work_items1, work_group_size),
                  [=](sycl::nd_item<1> item) {
                      size_t global_id = item.get_global_id(0);
                      int local_id = item.get_local_id(0);
                      int group_id = item.get_group(0);

                      if (global_id < data_size)
                          scratch[local_id] = buf_acc[global_id];
                      else
                          scratch[local_id] = 0;

                      // Do a tree reduction on items in work-group
                      for (int i = work_group_size / 2; i > 0; i >>= 1) {
                          item.barrier(sycl::access::fence_space::local_space);
                          if (local_id < i)
                              scratch[local_id] += scratch[local_id + i];
                      }

                      if (local_id == 0)
                          accum_acc[group_id] = scratch[0];
                  });
};

q.submit([&](auto &h) {
    sycl::accessor buf_acc(accum1_buf, h, sycl::read_only);
    sycl::accessor accum_acc(accum2_buf, h, sycl::write_only, sycl::no_init);
    sycl::local_accessor<int, 1> scratch(work_group_size, h);

    h.parallel_for(sycl::nd_range<1>(num_work_items2, work_group_size),
                  [=](sycl::nd_item<1> item) {
                      size_t global_id = item.get_global_id(0);
                      int local_id = item.get_local_id(0);
                      int group_id = item.get_group(0);

                      if (global_id < static_cast<size_t>(num_work_items2))
                          scratch[local_id] = buf_acc[global_id];
                      else
                          scratch[local_id] = 0;

                      // Do a tree reduction on items in work-group
                      for (int i = work_group_size / 2; i > 0; i >>= 1) {
                          item.barrier(sycl::access::fence_space::local_space);
                          if (local_id < i)
                              scratch[local_id] += scratch[local_id + i];
                      }

                      if (local_id == 0)
                          accum_acc[group_id] = scratch[0];
                  });
});
}

```

SYCL also supports built-in reduction operations, and you should use it where it is suitable because its implementation is fine tuned to the underlying architecture. The following kernel shows how to use the built-in reduction operator in the compiler.

```
q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    auto sumr = sycl::reduction(sum_buf, h, sycl::plus<>());
    h.parallel_for(sycl::nd_range<1>{data_size, 256}, sumr,
        [=](sycl::nd_item<1> item, auto &sumr_arg) {
            int glob_id = item.get_global_id(0);
            sumr_arg += buf_acc[glob_id];
        });
});
```

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.

```
q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
    sycl::local_accessor<int, 1> scratch(work_group_size, h);
    h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
        [=](sycl::nd_item<1> item) {
            size_t glob_id = item.get_global_id(0);
            size_t group_id = item.get_group(0);
            size_t loc_id = item.get_local_id(0);
            int offset = ((glob_id >> log2workitems_per_block)
                << log2elements_per_block) +
                (glob_id & mask);
            int sum = 0;
            for (int i = 0; i < elements_per_work_item; ++i)
                sum +=
                    buf_acc[(i << log2workitems_per_block) + offset];
            scratch[loc_id] = sum;
            // Serial Reduction
            item.barrier(sycl::access::fence_space::local_space);
            if (loc_id == 0) {
                int sum = 0;
                for (int i = 0; i < work_group_size; ++i)
                    sum += scratch[i];
                accum_acc[group_id] = sum;
            }
        });
});
```

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.

```
q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
    sycl::local_accessor<int, 1> scratch(work_group_size, h);
    h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
        [=](sycl::nd_item<1> item) {
            size_t glob_id = item.get_global_id(0);
            size_t group_id = item.get_group(0);
            size_t loc_id = item.get_local_id(0);
            int offset = ((glob_id >> log2workitems_per_block)
```

```

            << log2elements_per_block) +
            (glob_id & mask);
        int sum = 0;
        for (int i = 0; i < elements_per_work_item; ++i)
            sum +=
                buf_acc[(i << log2workitems_per_block) + offset];
        scratch[loc_id] = sum;
        // tree reduction
        item.barrier(sycl::access::fence_space::local_space);
        for (int i = work_group_size / 2; i > 0; i >>= 1) {
            item.barrier(sycl::access::fence_space::local_space);
            if (loc_id < static_cast<size_t>(i))
                scratch[loc_id] += scratch[loc_id + i];
        }
        if (loc_id == 0)
            accum_acc[group_id] = scratch[0];
    });
}
);

```

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 on which it was tested.

```

q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    auto sumr = sycl::reduction(sum_buf, h, sycl::plus<>());
    h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size}, sumr,
                    [=](sycl::nd_item<1> item, auto &sumr_arg) {
                        size_t glob_id = item.get_global_id(0);
                        int offset = ((glob_id >> log2workitems_per_block)
                                      << log2elements_per_block) +
                                      (glob_id & mask);
                        int sum = 0;
                        for (int i = 0; i < elements_per_work_item; ++i)
                            sum +=
                                buf_acc[(i << log2workitems_per_block) + offset];
                        sumr_arg += sum;
                    });
    });

```

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 lets you perform tree reduction through efficient barriers.

```

q.submit([&](auto &h) {
    const sycl::accessor buf_acc(buf, h);
    sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
    sycl::local_accessor<sycl::vec<int, 8>, 11> scratch(work_group_size, h);
    h.parallel_for(
        sycl::nd_range<1>{num_work_items, work_group_size},
        [=](sycl::nd_item<1> item) [[intel::reqd_sub_group_size(16)]] {
            size_t group_id = item.get_group(0);
            size_t loc_id = item.get_local_id(0);
            sycl::sub_group sg = item.get_sub_group();
            sycl::vec<int, 8> sum{0, 0, 0, 0, 0, 0, 0, 0};
            using global_ptr =
                sycl::multi_ptr<int, sycl::access::address_space::global_space>;
            int base = (group_id * work_group_size +
                        sg.get_group_id()[0] * sg.get_local_range()[0]) *
                        elements_per_work_item;

```

```

        for (int i = 0; i < elements_per_work_item / 8; ++i)
            sum += sg.load<8>(global_ptr(&buf_acc[base + i * 128]));
        scratch[loc_id] = sum;
        for (int i = work_group_size / 2; i > 0; i >>= 1) {
            item.barrier(sycl::access::fence_space::local_space);
            if (loc_id < static_cast<size_t>(i))
                scratch[loc_id] += scratch[loc_id + i];
        }
        if (loc_id == 0)
            accum_acc[group_id] = scratch[0];
    });
}
);

```

Different implementations of reduction operation are provided and discussed here, which may have different performance characteristics depending on the architecture of the accelerator. Another important thing to note is that the time it takes to bring 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.

## Kernel Launch

In SYCL, work is performed by enqueueing kernels into queues targeting specific devices. These kernels are submitted by the host to the device, 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 Intel® VTune™ Profiler or [unitrace](#) provides 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. This will involve the kernel submission time on the host, 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).

```

void emptyKernel1(sycl::queue &q) {
    Timer timer;
    for (int i = 0; i < iters; ++i)
        q.parallel_for(1, [=](auto) {
            /* NOP */
        }).wait();
}

```

```

    std::cout << " emptyKernel1: Elapsed time: " << timer.Elapsed() / iters
          << " sec\n";
} // 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.

```

void emptyKernel2(sycl::queue &q) {
    Timer timer;
    for (int i = 0; i < iters; ++i)
        q.parallel_for(1, [=](auto) {
            /* NOP */
        });
    std::cout << " emptyKernel2: Elapsed time: " << timer.Elapsed() / iters
          << " sec\n";
}

```

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

One way to measure the actual kernel execution time on the device is to use the SYCL built-in profiling API. The following code demonstrates usage of the SYCL 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. Profiling tools can provide this information.

```

#include <CL/sycl.hpp>

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_;
};

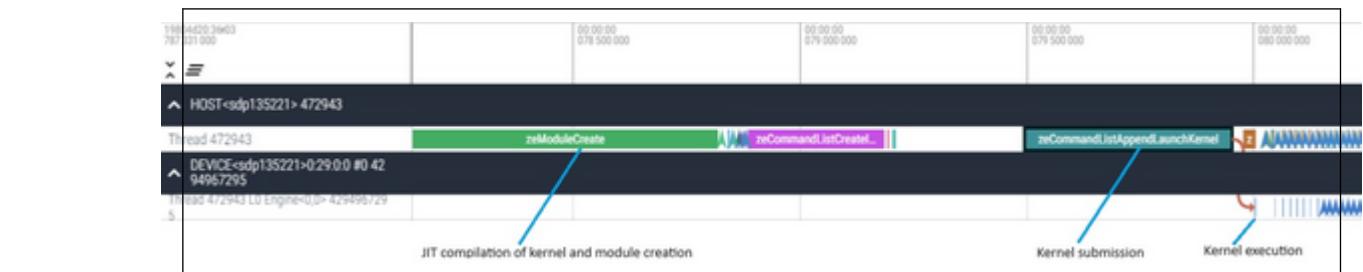
int main() {
    Timer timer;
    sycl::queue q{sycl::property::queue::enable_profiling()};
    auto evt = q.parallel_for(1000, [=](auto) {
        /* kernel statements here */
    });
    double t1 = timer.Elapsed();
    evt.wait();
    double t2 = timer.Elapsed();
    auto startK =
        evt.get_profiling_info<sycl::info::event_profiling::command_start>();
    auto endK =
        evt.get_profiling_info<sycl::info::event_profiling::command_end>();
    std::cout << "Kernel submission time: " << t1 << "secs\n";
    std::cout << "Kernel submission + execution time: " << t2 << "secs\n";
    std::cout << "Kernel execution time: "
          << ((double)(endK - startK)) / 1000000.0 << "secs\n";

    return 0;
}

```

The following picture shows the timeline of the execution for the above example. This picture is generated from running **unitrace** to generate a trace file and using a browser 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 the 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 `zeModuleCreate` in the figure below.

### Timeline of Kernel Execution



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.

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

SYCL has two kinds of queues that a programmer can create and use to 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 made at queue construction time through the property `sycl::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.

```
int multi_queue(sycl::queue &q, const IntArray &a, const IntArray &b) {
    IntArray s1, s2, s3;

    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf1(s1);
    sycl::buffer sum_buf2(s2);
    sycl::buffer sum_buf3(s3);

    size_t num_groups = 1;
    size_t wg_size = 256;
    auto start = std::chrono::steady_clock::now();
    for (int i = 0; i < iter; i++) {
        q.submit([&] (sycl::handler &h) {
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            sycl::accessor sum_acc(sum_buf1, h, sycl::write_only, sycl::no_init);

            h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                           [=](sycl::nd_item<1> index) {
```

```

        size_t loc_id = index.get_local_id();
        sum_acc[loc_id] = 0;
        for (int j = 0; j < 1000; j++) {
            for (size_t i = loc_id; i < array_size; i += wg_size) {
                sum_acc[loc_id] += a_acc[i] + b_acc[i];
            }
        });
    });
q.submit([&](sycl::handler &h) {
    sycl::accessor a_acc(a_buf, h, sycl::read_only);
    sycl::accessor b_acc(b_buf, h, sycl::read_only);
    sycl::accessor sum_acc(sum_buf2, h, sycl::write_only, sycl::no_init);

    h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                  [=](sycl::nd_item<1> index) {
                      size_t loc_id = index.get_local_id();
                      sum_acc[loc_id] = 0;
                      for (int j = 0; j < 1000; j++) {
                          for (size_t i = loc_id; i < array_size; i += wg_size) {
                              sum_acc[loc_id] += a_acc[i] + b_acc[i];
                          }
                      });
    });
    q.submit([&](sycl::handler &h) {
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        sycl::accessor sum_acc(sum_buf3, h, sycl::write_only, sycl::no_init);

        h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                      [=](sycl::nd_item<1> index) {
                          size_t loc_id = index.get_local_id();
                          sum_acc[loc_id] = 0;
                          for (int j = 0; j < 1000; j++) {
                              for (size_t i = loc_id; i < array_size; i += wg_size) {
                                  sum_acc[loc_id] += a_acc[i] + b_acc[i];
                              }
                          });
        });
    });
}
q.wait();
auto end = std::chrono::steady_clock::now();
std::cout << "multi_queue completed on device - took "
          << (end - start).count() << " u-secs\n";
// check results
return ((end - start).count());
} // 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.

```

sycl::property_list q_prop{sycl::property::queue::in_order()};
std::cout << "In order queue: Jitting+Execution time\n";
sycl::queue q1(sycl::default_selector_v, q_prop);
multi_queue(q1, a, b);
usleep(500 * 1000);
std::cout << "In order queue: Execution time\n";
multi_queue(q1, a, b);
}

```

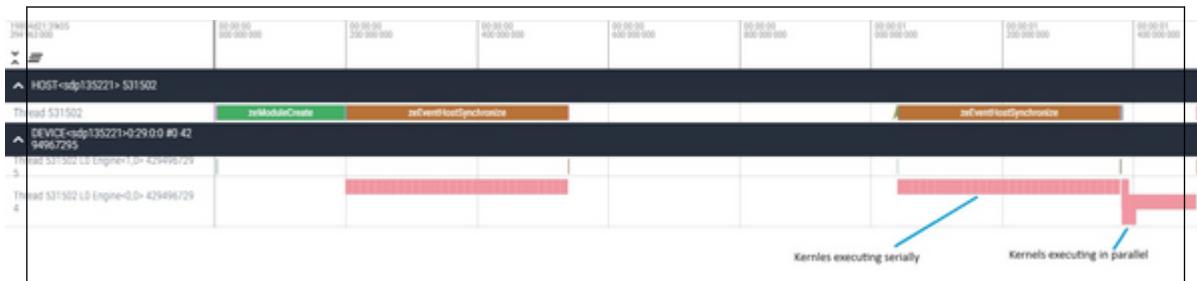
When the queue is out-of-order, the overall execution time is much lower, indicating that the machine is able to execute different kernels from the queue at the same time. The creation of the queue and the invocation of the kernel is shown below.

```
sycl::queue q2(sycl::default_selector_v);
std::cout << "Out of order queue: Jitting+Execution time\n";
multi_queue(q2, a, b);
usleep(500 * 1000);
std::cout << "Out of order queue: Execution time\n";
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 [unitrace](#) tool). 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 all the time. How many kernels are executed in parallel is affected by multiple factors such as the availability of hardware resources, the time gap between kernel submissions, etc.

### 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.

### 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 30 independent kernels that use the 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.

```
int VectorAdd(sycl::queue &q1, sycl::queue &q2, sycl::queue &q3,
              const IntArray &a, const IntArray &b) {

    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer<int> *sum_buf[3 * iter];
    for (size_t i = 0; i < (3 * iter); i++)
        sum_buf[i] = new sycl::buffer<int>(256);

    size_t num_groups = 1;
```

```
size_t wg_size = 256;
auto start = std::chrono::steady_clock::now();
for (int i = 0; i < iter; i++) {
    q1.submit([&](auto &h) {
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        auto sum_acc = sum_buf[3 * i]->get_access<sycl::access::mode::write>(h);

        h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                      [=](sycl::nd_item<1> index) {
                          size_t loc_id = index.get_local_id();
                          sum_acc[loc_id] = 0;
                          for (size_t i = loc_id; i < array_size; i += wg_size) {
                              sum_acc[loc_id] += a_acc[i] + b_acc[i];
                          }
                      });
    });
    q2.submit([&](auto &h) {
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        auto sum_acc =
            sum_buf[3 * i + 1]->get_access<sycl::access::mode::write>(h);

        h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                      [=](sycl::nd_item<1> index) {
                          size_t loc_id = index.get_local_id();
                          sum_acc[loc_id] = 0;
                          for (size_t i = loc_id; i < array_size; i += wg_size) {
                              sum_acc[loc_id] += a_acc[i] + b_acc[i];
                          }
                      });
    });
    q3.submit([&](auto &h) {
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        auto sum_acc =
            sum_buf[3 * i + 2]->get_access<sycl::access::mode::write>(h);

        h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                      [=](sycl::nd_item<1> index) {
                          size_t loc_id = index.get_local_id();
                          sum_acc[loc_id] = 0;
                          for (size_t i = loc_id; i < array_size; i += wg_size) {
                              sum_acc[loc_id] += a_acc[i] + b_acc[i];
                          }
                      });
    });
}
q1.wait();
q2.wait();
q3.wait();
auto end = std::chrono::steady_clock::now();
std::cout << "Vector add completed on device - took " << (end - start).count()
      << " u-secs\n";
// check results
for (size_t i = 0; i < (3 * iter); i++)
```

```

    delete sum_buf[i];
    return ((end - start).count());
} // end VectorAdd

```

Submitting the kernels to the same queue gives the best performance because all the kernels are able to just transfer the needed inputs once at the beginning and do all their computations.

```
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.

```

sycl::queue q1(sycl::default_selector_v);
sycl::queue q2(q1.get_context(), sycl::default_selector_v);
sycl::queue q3(q1.get_context(), sycl::default_selector_v);
VectorAdd(q1, q2, q3, a, b);

```

If the kernels are submitted to three different queues that 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.

```

sycl::queue q4(sycl::default_selector_v);
sycl::queue q5(sycl::default_selector_v);
sycl::queue q6(sycl::default_selector_v);
VectorAdd(q4, q5, q6, a, b);

```

If for some reason you 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 to 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 so the runtime will be different.

```

int VectorAdd(sycl::queue &q1, sycl::queue &q2, sycl::queue &q3,
              const IntArray &a, const IntArray &b) {

    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer<int> *sum_buf[3 * iter];
    for (size_t i = 0; i < (3 * iter); i++)
        sum_buf[i] = new sycl::buffer<int>(256);

    size_t num_groups = 1;
    size_t wg_size = 256;
    auto start = std::chrono::steady_clock::now();
    for (int i = 0; i < iter; i++) {
        q1.submit([&](auto &h) {
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            auto sum_acc = sum_buf[3 * i]->get_access<sycl::access::mode::write>(h);

            h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                           [=](sycl::nd_item<1> index) {
                               size_t loc_id = index.get_local_id();
                               sum_acc[loc_id] = 0;
                               for (int j = 0; j < 1000; j++)

```

```

        for (size_t i = loc_id; i < array_size; i += wg_size) {
            sum_acc[loc_id] += a_acc[i] + b_acc[i];
        }
    });
});

q2.submit([&](auto &h) {
    sycl::accessor a_acc(a_buf, h, sycl::read_only);
    sycl::accessor b_acc(b_buf, h, sycl::read_only);
    auto sum_acc =
        sum_buf[3 * i + 1]->get_access<sycl::access::mode::write>(h);

    h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                  [=](sycl::nd_item<1> index) {
                      size_t loc_id = index.get_local_id();
                      sum_acc[loc_id] = 0;
                      for (int j = 0; j < 1000; j++)
                          for (size_t i = loc_id; i < array_size; i += wg_size) {
                              sum_acc[loc_id] += a_acc[i] + b_acc[i];
                          }
                  });
});

q3.submit([&](auto &h) {
    sycl::accessor a_acc(a_buf, h, sycl::read_only);
    sycl::accessor b_acc(b_buf, h, sycl::read_only);
    auto sum_acc =
        sum_buf[3 * i + 2]->get_access<sycl::access::mode::write>(h);

    h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                  [=](sycl::nd_item<1> index) {
                      size_t loc_id = index.get_local_id();
                      sum_acc[loc_id] = 0;
                      for (int j = 0; j < 1000; j++)
                          for (size_t i = loc_id; i < array_size; i += wg_size) {
                              sum_acc[loc_id] += a_acc[i] + b_acc[i];
                          }
                  });
});

q1.wait();
q2.wait();
q3.wait();
auto end = std::chrono::steady_clock::now();
std::cout << "Vector add completed on device - took " << (end - start).count()
      << " u-secs\n";
// check results
for (size_t i = 0; i < (3 * iter); i++)
    delete sum_buf[i];
return ((end - start).count());
} // end VectorAdd
}

```

## Avoiding Redundant Queue Constructions

To execute kernels on a device, the user must create a queue, which references an associated context, platform, and device. These 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.

```
int reductionMultipleQMultipleC(std::vector<int> &data, int iter) {
    const size_t data_size = data.size();
    int sum = 0;

    int work_group_size = 512;
    int num_work_groups = 1;
    int num_work_items = work_group_size;

    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};

    sycl::buffer<int> buf(data.data(), data_size, props);
    sycl::buffer<int> sum_buf(&sum, 1, props);

    sycl::queue q1{sycl::default_selector_v, exception_handler};
    // initialize data on the device
    q1.submit([&](auto &h) {
        sycl::accessor buf_acc(buf, h, sycl::write_only, sycl::no_init);
        h.parallel_for(data_size, [=](auto index) { buf_acc[index] = 1; });
    });

    double elapsed = 0;
    for (int i = 0; i < iter; i++) {
        sycl::queue q2{sycl::default_selector_v, exception_handler};
        if (i == 0)
            std::cout << q2.get_device().get_info<sycl::info::device::name>() << "\n";
        // reductionMultipleQMultipleC main begin
        Timer timer;
        q2.submit([&](auto &h) {
            sycl::accessor buf_acc(buf, h, sycl::read_only);
            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
            sycl::local_accessor<int, 1> scratch(work_group_size, h);
            h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
                           [=](sycl::nd_item<1> item) {
                               size_t loc_id = item.get_local_id(0);
                               int sum = 0;
                               for (int i = loc_id; i < data_size; i += num_work_items)
                                   sum += buf_acc[i];
                               scratch[loc_id] = sum;
                               for (int i = work_group_size / 2; i > 0; i >>= 1) {
                                   item.barrier(sycl::access::fence_space::local_space);
                                   if (loc_id < i)
                                       scratch[loc_id] += scratch[loc_id + i];
                               }
                               if (loc_id == 0)
                                   sum_acc[0] = scratch[0];
                           });
        });
        // reductionMultipleQMultipleC main end
        q2.wait();
        sycl::host_accessor h_acc(sum_buf);
        sum = h_acc[0];
        elapsed += timer.Elapsed();
    }
}
```

```

elapsed = elapsed / iter;
if (sum == sum_expected)
    std::cout << "SUCCESS: Time reductionMultipleQMultipleC    = " << elapsed
    << "s"
    << " sum = " << sum << "\n";
else
    std::cout << "ERROR: reductionMultipleQMultipleC Expected " << sum_expected
    << " but got " << sum << "\n";
return sum;
} // end reductionMultipleQMultipleC

```

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

```

int reductionSingleQ(std::vector<int> &data, int iter) {
    const size_t data_size = data.size();
    int sum = 0;

    int work_group_size = 512;
    int num_work_groups = 1;
    int num_work_items = work_group_size;

    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};

    sycl::buffer<int> buf(data.data(), data_size, props);
    sycl::buffer<int> sum_buf(&sum, 1, props);
    sycl::queue q{sycl::default_selector_v, exception_handler};
    std::cout << q.get_device().get_info<sycl::info::device::name>() << "\n";

    // initialize data on the device
    q.submit([&](auto &h) {
        sycl::accessor buf_acc(buf, h, sycl::write_only, sycl::no_init);
        h.parallel_for(data_size, [=](auto index) { buf_acc[index] = 1; });
    });

    double elapsed = 0;
    for (int i = 0; i < iter; i++) {
        // reductionIntBarrier main begin
        Timer timer;
        q.submit([&](auto &h) {
            sycl::accessor buf_acc(buf, h, sycl::read_only);
            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
            sycl::local_accessor<int, 1> scratch(work_group_size, h);
            h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
                           [=](sycl::nd_item<1> item) {
                               size_t loc_id = item.get_local_id(0);
                               int sum = 0;
                               for (int i = loc_id; i < data_size; i += num_work_items)
                                   sum += buf_acc[i];
                               scratch[loc_id] = sum;
                               for (int i = work_group_size / 2; i > 0; i >>= 1) {
                                   item.barrier(sycl::access::fence_space::local_space);
                                   if (loc_id < i)
                                       scratch[loc_id] += scratch[loc_id + i];
                               }
                               if (loc_id == 0)
                                   sum_acc[0] = scratch[0];
                           });
        });
    });
}

```

```

// reductionSingleQ main end
q.wait();
sycl::host_accessor h_acc(sum_buf);
sum = h_acc[0];
elapsed += timer.Elapsed();
}
elapsed = elapsed / iter;
if (sum == sum_expected)
    std::cout << "SUCCESS: Time reductionSingleQ    = " << elapsed << "s"
        << " sum = " << sum << "\n";
else
    std::cout << "ERROR: reductionSingleQ Expected " << sum_expected
        << " but got " << sum << "\n";
return sum;
} // end reductionSingleQ

```

In case you need to create multiple queues, 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.

```

int reductionMultipleQSingleC(std::vector<int> &data, int iter) {
    const size_t data_size = data.size();
    int sum = 0;

    int work_group_size = 512;
    int num_work_groups = 1;
    int num_work_items = work_group_size;

    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};

    sycl::buffer<int> buf(data.data(), data_size, props);
    sycl::buffer<int> sum_buf(&sum, 1, props);

    sycl::queue q1{sycl::default_selector_v, exception_handler};
    // initialize data on the device
    q1.submit([&](auto &h) {
        sycl::accessor buf_acc(buf, h, sycl::write_only, sycl::no_init);
        h.parallel_for(data_size, [=](auto index) { buf_acc[index] = 1; });
    });

    double elapsed = 0;
    for (int i = 0; i < iter; i++) {
        sycl::queue q2{q1.get_context(), sycl::default_selector_v,
                      exception_handler};
        if (i == 0)
            std::cout << q2.get_device().get_info<sycl::info::device::name>() << "\n";
        // reductionMultipleQSingleC main begin
        Timer timer;
        q2.submit([&](auto &h) {
            sycl::accessor buf_acc(buf, h, sycl::read_only);
            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
            sycl::local_accessor<int, 1> scratch(work_group_size, h);
            h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
                          [=](sycl::nd_item<1> item) {
                size_t loc_id = item.get_local_id(0);
                int sum = 0;
                for (int i = loc_id; i < data_size; i += num_work_items)
                    sum += buf_acc[i];
            });
        });
    }
}

```

```

        scratch[loc_id] = sum;
        for (int i = work_group_size / 2; i > 0; i >>= 1) {
            item.barrier(sycl::access::fence_space::local_space);
            if (loc_id < i)
                scratch[loc_id] += scratch[loc_id + i];
        }
        if (loc_id == 0)
            sum_acc[0] = scratch[0];
    });
}

// reductionMultipleQSingleC main end
q2.wait();
sycl::host_accessor h_acc(sum_buf);
sum = h_acc[0];
elapsed += timer.Elapsed();
}
elapsed = elapsed / iter;
if (sum == sum_expected)
    std::cout << "SUCCESS: Time reductionMultipleQSingleContext = " << elapsed
        << "s"
        << " sum = " << sum << "\n";
else
    std::cout << "ERROR: reductionMultipleQSingleContext Expected "
        << sum_expected << " but got " << sum << "\n";
return sum;
} // end reductionMultipleQSingleC

```

## Programming Intel® XMX Using SYCL Joint Matrix Extension

Joint matrix is a new SYCL extension for matrix hardware programming. This unifies targets like Intel® Advanced Matrix Extensions (Intel® AMX) for CPUs, Intel® Xe Matrix Extensions (Intel® XMX) for GPUs, and NVIDIA\* Tensor Cores. In general, frameworks like TensorFlow and libraries like oneDNN are the answer for many types of AI users and applications. However, for users who want to build their own neural networks applications, these libraries and frameworks become high-level because users cannot do custom optimizations, and heavyweight because the size of the library is large. Moreover, new operations are introduced and changing in machine learning domain for which frameworks and libraries do not provide timely and performing solutions. For such cases, joint matrix has a lower-level of abstraction than the frameworks to provide performance, productivity, and fusion capabilities but at the same time offers portability by using one code to target different matrix hardware.

The detailed specification of this extension can be found [here](#)

The joint matrix extensions consists of a new type `joint_matrix`, explicit memory operations `joint_matrix_load` and `joint_matrix_store`, `joint_matrix_fill` for matrix initialization, `joint_matrix_mad` for the actual multiply and add operations, and `joint_matrix_apply` for element wise operations.

In the code below, the kernel makes use of the joint matrix interface by declaring 3 `joint_matrix` matrices: `tA`, `tB`, `tC` and computes the operation `tC += tA * tB`. DPAS (Dot Product and Accumulate Systolic) is the name of the elementary operations done in Intel® XMX. For more examples that use `joint_matrix`, please refer to the [Intel LLVM-test-suite repo](#).

In order for this example to run successfully on Intel® XMX, The GPU must have Intel® XMX hardware. There is no emulation or fall back strategy in the joint matrix implementation.

```

size_t NDRangeM = M / TM;
size_t NDRangeN = N / TN;
sycl::buffer<bfloat16, 2> bufA(A.get_data(), sycl::range<2>(M, K));
sycl::buffer<bfloat16, 2> bufB(B.get_data(), sycl::range<2>(K, N));
sycl::buffer<float, 2> bufC((float *)C.get_data(), sycl::range<2>(M, N));

```

```

sycl::queue q;
q.submit([&](sycl::handler &cgh) {
    sycl::accessor accC(bufC, cgh, sycl::read_write);
    sycl::accessor accA(bufA, cgh, sycl::read_only);
    sycl::accessor accB(bufB, cgh, sycl::read_only);
    cgh.parallel_for(
        sycl::nd_range<2>({NDRangeM, NDRangeN * SG_SZ}, {1, 1 * SG_SZ}),
        [=](sycl::nd_item<2> spmd_item) [[intel::reqd_sub_group_size(SG_SZ)]]

    {
        // The joint matrix API has to be accessed by all the workitems in a
        // subgroup these functions will be called once by the subgroup no
        // code divergence between the workitems
        const auto global_idx = spmd_item.get_global_id(0);
        const auto global_idy = spmd_item.get_global_id(1);
        const auto sg_startx = global_idx - spmd_item.get_local_id(0);
        const auto sg_starty = global_idy - spmd_item.get_local_id(1);

        sycl::sub_group sg = spmd_item.get_sub_group();
        sycl::ext::oneapi::experimental::matrix::joint_matrix<
            sycl::sub_group, bfloat16, use::a, TM, TK, layout::row_major>
            sub_a;
        // For B, we assume B has been already VNNIed.
        sycl::ext::oneapi::experimental::matrix::joint_matrix<
            sycl::sub_group, bfloat16, use::b, TK, TN,
            layout::ext_intel_packed>
            sub_b;
        sycl::ext::oneapi::experimental::matrix::joint_matrix<
            sycl::sub_group, float, use::accumulator, TM, TN>
            sub_c;

        joint_matrix_fill(sg, sub_c, 1.0);
        for (int k = 0; k < K / TK; k += 1) {
            joint_matrix_load(
                sg, sub_a,
                accA.template get_multi_ptr<sycl::access::decorated::no>() +
                (sg_startx * TM) * K + k * TK,
                K);
            joint_matrix_load(
                sg, sub_b,
                accB.template get_multi_ptr<sycl::access::decorated::no>() +
                (k * TK / 2) * (N * 2) + sg_starty / SG_SZ * TN * 2,
                N * 2);
            joint_matrix_mad(sg, sub_c, sub_a, sub_b, sub_c);
        }
        joint_matrix_apply(sg, sub_c, [=](float &x) { x *= ALPHA; });
        joint_matrix_store(
            sg, sub_c,
            accC.template get_multi_ptr<sycl::access::decorated::no>() +
            (sg_startx * TM) * N + sg_starty / SG_SZ * TN,
            N, layout::row_major);
    }); // parallel for
}).wait();

```

In order to get optimal performance on Intel® XMX, the GEMM kernel has to be written in a way that keeps feeding Intel® XMX with data it needs to perform the maximum multiply and adds operations/cycle. Some of these tuning techniques are the following:

- One sub-group can perform multiple DPAS operations.
- Blocking for cache locality should be made on the three ``*i*``, *j*, and *k* dimensions. Blocking on the first two dimensions is included in the global range of the kernel. The *k*-level cache blocking is done within the kernel body. e.g. by choosing block factors of 256x256x32, global range results in:

```
range<2> global{M / 256, N / 256 * SG_SIZE};
```

Then, by choosing one sub - group to perform 64x32x32 elements, local range results in:

```
range<2> local{256 / 64, 256 / 32 * SG_SIZE};
```

- Large register file per thread gives the best results for the GEMM kernel. This can be explicitly specified in the compilation command as follows:

```
icpx -fsycl -fsycl-targets=spir64_gen -Xsycl-target-backend "-device pvc -options -ze-opt-large-register-file" joint-matrix.cpp
```

## Doing I/O in the Kernel

Print statement is the most fundamental capability needed for looking at the results of a program. In accelerators, printing is surprisingly hard and also fairly expensive in terms of overhead.

SYCL\* provides some capabilities to help make this task similar to standard I/O C/C++ programs, but there are some quirks you need to understand because of the way accelerators work. File I/O is not possible from SYCL\* kernels.

SYCL\* provides the *stream* class to let you 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 built-in data types (such as *int*, *char* and *float*) as well as some common classes (such as *sycl::nd\_range* and *sycl::group*).

Here is an example usage of a SYCL *stream* instance:

```
void out1() {
    constexpr int N = 16;
    sycl::queue q;
    q.submit([&](auto &cgh) {
        sycl::stream str(8192, 1024, cgh);
        cgh.parallel_for(N, [=](sycl::item<1> it) {
            int id = it[0];
            /* Send the identifier to a stream to be printed on the console */
            str << "ID=" << id << sycl::endl;
        });
    }).wait();
} // end out1
```

The use of *sycl::endl* is analogous to the use of the C++ STL *std::endl* 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
ID=11
ID=12
ID=13
ID=14
ID=15
```

Care must be taken in choosing the appropriate *BufferSize* and *MaxStatementSize* parameters. Insufficient sizes may cause statements to either not be printed, or to be printed with less information than expected. Consider the following kernel:

```
void out2() {
    sycl::queue q;
    q.submit([&](auto &cgh) {
        sycl::stream str(8192, 4, cgh);
        cgh.parallel_for(1, [=](sycl::item<1>) {
            str << "ABC" << sycl::endl;      // Print statement 1
            str << "ABCDEFG" << sycl::endl; // Print statement 2
        });
    }).wait();
} // end out2
```

Compiling and running this 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:

```
void out3() {
    sycl::queue q;
    q.submit([&](auto &cgh) {
        sycl::stream str(8192, 10, cgh);
        cgh.parallel_for(1, [=](sycl::item<1>) {
            str << "ABC" << sycl::endl;      // Print statement 1
            str << "ABCDEFG" << sycl::endl; // Print statement 2
        });
    }).wait();
} // 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 you should expect statements from different work items to be interleaved. Consider the following kernel:

```
void out4() {
    sycl::queue q;
    q.submit([&](auto &cgh) {
        sycl::stream str(8192, 1024, cgh);
        cgh.parallel_for(sycl::nd_range<1>(32, 4), [=](sycl::nd_item<1> it) {
            int id = it.get_global_id();
            str << "ID=" << id << sycl::endl;
        });
    }).wait();
} // 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, depending on the order the 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, no statement will be printed. To illustrate this, consider the following kernel, which raises an exception:

```
void out5() {
    int *m = NULL;
    sycl::queue q;
    q.submit([&](auto &cgh) {
        sycl::stream str(8192, 1024, cgh);
        cgh.parallel_for(sycl::nd_range<1>(32, 4), [=](sycl::nd_item<1> it) {
            int id = it.get_global_id();
            str << "ID=" << id << sycl::endl;
            if (id == 31)
```

```

        *m = id;
    });
}).wait();
} // end out5

```

Compiling and executing the above code generates a segmentation fault due to the write to a null pointer.

```
Segmentation fault (core dumped)
```

None of the print statements are actually printed to the console. Instead, you will see an error message about a segmentation fault. This is unlike traditional C/C++ streams.

## Using Libraries for GPU Offload

Several libraries are available with oneAPI toolkits that can simplify the programming process by providing specialized APIs for use in optimized applications. This section provides steps on using the libraries, including code samples, for application accelerations. Detailed information about each library, including the available APIs, is available in the main documentation for the specific library.

- [Using Performance Libraries](#)
- [Using Standard Library Functions in SYCL Kernels](#)
- [Efficiently Implementing Fourier Correlation Using oneAPI Math Kernel Library \(oneMKL\)](#)
- [Boost Matrix Multiplication Performance with Intel® Xe Matrix Extensions](#)

## Using Performance Libraries

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:

$$C_{m,n} = A_{m,k} \times B_{k,n} = \sum_{k=0}^K A_{m,k} \times B_{k,n}$$

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

```

// Multiply matrices A and B
for (m = 0; m < M; m++) {
    for (n = 0; n < N; n++) {
        C[m][n] = 0.0;
        for (k = 0; k < K; k++) {
            C[m][n] += A[m][k] * B[k][n];
        }
    }
} // 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 SYCL 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):

```

// Initialize SYCL queue
sycl::queue Q(sycl::default_selector_v);
auto sycl_device = Q.get_device();
auto sycl_context = Q.get_context();

```

```

std::cout << "Running on: "
    << Q.get_device().get_info<sycl::info::device::name>() << std::endl;

// Allocate matrices A, B, and C in USM
auto A = sycl::malloc_shared<float *>(M, sycl_device, sycl_context);
for (m = 0; m < M; m++)
    A[m] = sycl::malloc_shared<float>(K, sycl_device, sycl_context);

auto B = sycl::malloc_shared<float *>(K, sycl_device, sycl_context);
for (k = 0; k < K; k++)
    B[k] = sycl::malloc_shared<float>(N, sycl_device, sycl_context);

auto C = sycl::malloc_shared<float *>(M, sycl_device, sycl_context);
for (m = 0; m < M; m++)
    C[m] = sycl::malloc_shared<float>(N, sycl_device, sycl_context);

// Initialize matrices A, B, and C

```

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

```

// Offload matrix multiplication kernel
Q.parallel_for(sycl::range<2>{M, N}, [=](sycl::id<2> id) {
    unsigned int m = id[0];
    unsigned int n = id[1];

    float sum = 0.0;
    for (unsigned int k = 0; k < K; k++)
        sum += A[m][k] * B[k][n];

    C[m][n] = sum;
}).wait(); // End matrix multiplication

```

However, simply offloading such code to an accelerator is unlikely to restore performance. In fact, performance could get worse. Badly written code is still badly written whether it runs on the host or a device.

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:

```

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

```

The library function is more versatile than the naive implementations and is expected to give better performance. For example, the library function can transpose one or both matrices before multiplication, if necessary. This 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
- Intel VPL for video processing

Check whether your required operation is already available in a oneAPI library before creating your own implementation of it.

## Using Standard Library Functions in SYCL Kernels

Some, but not all, standard C++ functions can be called inside SYCL 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 SYCL kernel. The following program generates a sequence of random numbers using the `rand()` function:

```
#include <CL/sycl.hpp>
#include <iostream>
#include <random>

constexpr int N = 5;

extern SYCL_EXTERNAL int rand(void);

int main(void) {
#if defined CPU
    sycl::queue Q(sycl::cpu_selector_v);
#elif defined GPU
    sycl::queue Q(sycl::gpu_selector_v);
#else
    sycl::queue Q(sycl::default_selector_v);
#endif

    std::cout << "Running on: "
        << Q.get_device().get_info<sycl::info::device::name>() << std::endl;

    // Attempt to use rand() inside a DPC++ kernel
    auto test1 = sycl::malloc_shared<float>(N, Q.get_device(), Q.get_context());

    srand((unsigned)time(NULL));
    Q.parallel_for(N, [=](auto idx) {
        test1[idx] = (float)rand() / (float)RAND_MAX;
    }).wait();

    // Show the random number sequence
    for (int i = 0; i < N; i++)
        std::cout << test1[i] << std::endl;

    // Cleanup
    sycl::free(test1, Q.get_context());
}
```

The program can be compiled to execute the SYCL kernel on the CPU (i.e., `cpu_selector`), or GPU (i.e., `gpu_selector`) devices. It compiles without errors on the two devices, and runs correctly on the CPU, but fails when run on the GPU:

```
$ icpx -fsycl -DCPU -std=c++17 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

$ icpx -fsycl -DGPU -std=c++17 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 SYCL 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](#)):

```

#include <CL/sycl.hpp>
#include <iostream>
#include <oneapi/dpl/random>
#include <oneapi/mkl/rng.hpp>

int main(int argc, char **argv) {
    unsigned int N = (argc == 1) ? 20 : std::stoi(argv[1]);
    if (N < 20)
        N = 20;

    // Generate sequences of random numbers between [0.0, 1.0] using oneDPL and
    // oneMKL
    sycl::queue Q(sycl::gpu_selector_v);
    std::cout << "Running on: "
           << Q.get_device().get_info<sycl::info::device::name>() << std::endl;

    auto test1 = sycl::malloc_shared<float>(N, Q.get_device(), Q.get_context());
    auto test2 = sycl::malloc_shared<float>(N, Q.get_device(), Q.get_context());

    std::uint32_t seed = (unsigned)time(NULL); // Get RNG seed value

    // oneDPL random number generator on GPU device
    clock_t start_time = clock(); // Start timer

    Q.parallel_for(N, [=](auto idx) {
        oneapi::dpl::minstd_rand rng_engine(seed, idx); // Initialize RNG engine
        oneapi::dpl::uniform_real_distribution<float>
            rng_distribution; // Set RNG distribution
        test1[idx] = rng_distribution(rng_engine); // Generate RNG sequence
    }).wait();

    clock_t end_time = clock(); // Stop timer
    std::cout << "oneDPL took " << float(end_time - start_time) / CLOCKS_PER_SEC

```

```

    << " seconds to generate " << N
    << " uniformly distributed random numbers." << std::endl;

// oneMKL random number generator on GPU device
start_time = clock(); // Start timer

oneapi::mkl::rng::mcg31ml engine(
    Q, seed); // Initialize RNG engine, set RNG distribution
oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
    rng_distribution(0.0, 1.0);
oneapi::mkl::rng::generate(rng_distribution, engine, N, test2)
    .wait(); // Generate RNG sequence

end_time = clock(); // Stop timer
std::cout << "oneMKL took " << float(end_time - start_time) / CLOCKS_PER_SEC
    << " seconds to generate " << N
    << " uniformly distributed random numbers." << std::endl;

// Show first ten random numbers from each method
std::cout << std::endl
    << "oneDPL"
    << "\t"
    << "oneMKL" << std::endl;
for (int i = 0; i < 10; i++)
    std::cout << test1[i] << " " << test2[i] << std::endl;

// Show last ten random numbers from each method
std::cout << "..." << std::endl;
for (size_t i = N - 10; i < N; i++)
    std::cout << test1[i] << " " << test2[i] << std::endl;

// Cleanup
sycl::free(test1, Q.get_context());
sycl::free(test2, Q.get_context());
}

```

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 SYCL kernel. In the oneMKL code, the `mkl::rng` functions handle the device offload implicitly.

## **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 0 0 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:

$$\text{corr}_{\alpha} = \sum_{i=0}^{N-1} \text{sig1}_i \times \text{sig2}_{i+\alpha}$$

where

$N$   
is the number of elements in the signal vectors and

$\alpha$   
is the shift of

$\text{sig2}$

relative to

$\text{sig1}$

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 the following example, two artificial signals will be created on the device, transformed in-place, and 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:

```
// Create buffers for signal data. This will only be used on the device.
sycl::buffer<float> sig1_buf{N + 2};
sycl::buffer<float> sig2_buf{N + 2};

// Declare container to hold the correlation result (computed on the device,
// used on the host)
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:

```
// Open new scope to trigger update of correlation result
{
    sycl::buffer<float> corr_buf(corr);

    // Initialize the input signals with artificial data
    std::uint32_t seed = (unsigned)time(NULL); // Get RNG seed value
    oneapi::mkl::rng::mcg31m1 engine(Q, seed); // Initialize RNG engine
                                                // Set RNG distribution
    oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
        rng_distribution(-0.00005, 0.00005);

    oneapi::mkl::rng::generate(rng_distribution, engine, N, sig1_buf); // Noise
    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:

```
Q.submit([&](sycl::handler &h) {
    sycl::accessor sig1_acc{sig1_buf, h, sycl::write_only};
    sycl::accessor sig2_acc{sig2_buf, h, sycl::write_only};
    h.single_task<>([=]()
    {
        sig1_acc[N - N / 4 - 1] = 1.0;
        sig1_acc[N - N / 4] = 1.0;
        sig1_acc[N - N / 4 + 1] = 1.0; // Signal
        sig2_acc[N / 4 - 1] = 1.0;
        sig2_acc[N / 4] = 1.0;
        sig2_acc[N / 4 + 1] = 1.0;
    });
}); // End signal initialization
```

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

```
// Initialize FFT descriptor
oneapi::mkl::dft::descriptor<oneapi::mkl::dft::precision::SINGLE,
                           oneapi::mkl::dft::domain::REAL>
    transform_plan(N);
transform_plan.commit(Q);

// Perform forward transforms on real arrays
oneapi::mkl::dft::compute_forward(transform_plan, sig1_buf);
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};
    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:

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

The IDFT step completes the computation:

```
// Perform backward transform on complex correlation array
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:

```
#include <CL/sycl.hpp>
#include <iostream>
#include <mkl.h>
#include <oneapi/mkl/dfti.hpp>
#include <oneapi/mkl/rng.hpp>
#include <oneapi/mkl/vm.hpp>

int main(int argc, char **argv) {
    unsigned int N = (argc == 1) ? 32 : std::stoi(argv[1]);
    if ((N % 2) != 0)
        N++;
    if (N < 32)
        N = 32;
```

```
// Initialize SYCL queue
sycl::queue Q(sycl::default_selector_v);
std::cout << "Running on: "
    << Q.get_device().get_info<sycl::info::device::name>() << std::endl;

// Create buffers for signal data. This will only be used on the device.
sycl::buffer<float> sig1_buf{N + 2};
sycl::buffer<float> sig2_buf{N + 2};

// Declare container to hold the correlation result (computed on the device,
// used on the host)
std::vector<float> corr(N + 2);

// Open new scope to trigger update of correlation result
{
    sycl::buffer<float> corr_buf(corr);

    // Initialize the input signals with artificial data
    std::uint32_t seed = (unsigned)time(NULL); // Get RNG seed value
    oneapi::mkl::rng::mcg31m1 engine(Q, seed); // Initialize RNG engine
                                                // Set RNG distribution
    oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
        rng_distribution(-0.00005, 0.00005);

    oneapi::mkl::rng::generate(rng_distribution, engine, N, sig1_buf); // Noise
    oneapi::mkl::rng::generate(rng_distribution, engine, N, sig2_buf);

    Q.submit([&](sycl::handler &h) {
        sycl::accessor sig1_acc{sig1_buf, h, sycl::write_only};
        sycl::accessor sig2_acc{sig2_buf, h, sycl::write_only};
        h.single_task<>([=] () {
            sig1_acc[N - N / 4 - 1] = 1.0;
            sig1_acc[N - N / 4] = 1.0;
            sig1_acc[N - N / 4 + 1] = 1.0; // Signal
            sig2_acc[N / 4 - 1] = 1.0;
            sig2_acc[N / 4] = 1.0;
            sig2_acc[N / 4 + 1] = 1.0;
        });
    }); // End signal initialization

    clock_t start_time = clock(); // Start timer

    // Initialize FFT descriptor
    oneapi::mkl::dft::descriptor<oneapi::mkl::dft::precision::SINGLE,
                                oneapi::mkl::dft::domain::REAL>
        transform_plan(N);
    transform_plan.commit(Q);

    // Perform forward transforms on real arrays
    oneapi::mkl::dft::compute_forward(transform_plan, sig1_buf);
    oneapi::mkl::dft::compute_forward(transform_plan, sig2_buf);

    // Compute: DFT(sig1) * CONJG(DFT(sig2))
    auto sig1_buf_cplx =
        sig1_buf.template reinterpret<std::complex<float>, 1>((N + 2) / 2);
    auto sig2_buf_cplx =
        sig2_buf.template reinterpret<std::complex<float>, 1>((N + 2) / 2);
    auto corr_buf_cplx =
```

```

corr_buf.template reinterpret<std::complex<float>, 1>((N + 2) / 2);
oneapi::mkl::vm::mulbyconj(Q, N / 2, sig1_buf_cplx, sig2_buf_cplx,
                           corr_buf_cplx);

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

clock_t end_time = clock(); // Stop timer
std::cout << "The 1D correlation (N = " << N << ") took "
       << float(end_time - start_time) / CLOCKS_PER_SEC << " seconds."
       << std::endl;

} // Buffer holding correlation result is now out of scope, forcing update of
  // host container

// Find the shift that gives maximum correlation value
float max_corr = 0.0;
int shift = 0;
for (unsigned int idx = 0; idx < N; idx++) {
    if (corr[idx] > max_corr) {
        max_corr = corr[idx];
        shift = idx;
    }
}
int _N = static_cast<int>(N);
shift =
    (shift > _N / 2) ? shift - _N : shift; // Treat the signals as circularly
                                              // shifted versions of each other.
std::cout << "Shift the second signal " << shift
       << " elements relative to the first signal to get a maximum, "
       "normalized correlation score of "
       << max_corr / N << "." << std::endl;
}

```

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:

```

// Initialize signal and correlation arrays
auto sig1 = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
auto sig2 = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
auto corr = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);

// Initialize input signals with artificial data
std::uint32_t seed = (unsigned)time(NULL); // Get RNG seed value
oneapi::mkl::rng::mcg31ml engine(Q, seed); // Initialize RNG engine
                                            // Set RNG distribution
oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
    rng_distribution(-0.00005, 0.00005);

// Warning: These statements run on the device.
auto evt1 =
    oneapi::mkl::rng::generate(rng_distribution, engine, N, sig1); // Noise
auto evt2 = oneapi::mkl::rng::generate(rng_distribution, engine, N, sig2);
evt1.wait();
evt2.wait();

// Warning: These statements run on the host, so sig1 and sig2 will have to be
// updated on the device.

```

```

sig1[N - N / 4 - 1] = 1.0;
sig1[N - N / 4] = 1.0;
sig1[N - N / 4 + 1] = 1.0; // Signal
sig2[N / 4 - 1] = 1.0;
sig2[N / 4] = 1.0;
sig2[N / 4 + 1] = 1.0;

```

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

```

// Perform forward transforms on real arrays
evt1 = oneapi::mkl::dft::compute_forward(transform_plan, sig1);
evt2 = oneapi::mkl::dft::compute_forward(transform_plan, sig2);

// Compute: DFT(sig1) * CONJG(DFT(sig2))
oneapi::mkl::vm::mulbyconj(
    Q, N / 2, reinterpret_cast<std::complex<float> *>(sig1),
    reinterpret_cast<std::complex<float> *>(sig2),
    reinterpret_cast<std::complex<float> *>(corr), {evt1, evt2})
    .wait();

// Perform backward transform on complex correlation array
oneapi::mkl::dft::compute_backward(transform_plan, corr).wait();

```

It is also necessary to free the allocated memory:

```

sycl::free(sig1, sycl_context);
sycl::free(sig2, sycl_context);
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 SYCL 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 SYCL 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:

```

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

```

The explicit buffering and USM implementations now have equivalent performance, indicating that the SYCL 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:

```

#include <CL/sycl.hpp>
#include <iostream>
#include <mkl.h>
#include <oneapi/mkl/dfti.hpp>
#include <oneapi/mkl/rng.hpp>
#include <oneapi/mkl/vm.hpp>

```

```
int main(int argc, char **argv) {
    unsigned int N = (argc == 1) ? 32 : std::stoi(argv[1]);
    if ((N % 2) != 0)
        N++;
    if (N < 32)
        N = 32;

    // Initialize SYCL queue
    sycl::queue Q(sycl::default_selector_v);
    auto sycl_device = Q.get_device();
    auto sycl_context = Q.get_context();
    std::cout << "Running on: "
        << Q.get_device().get_info<sycl::info::device::name>() << std::endl;

    // Initialize signal and correlation arrays
    auto sig1 = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
    auto sig2 = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
    auto corr = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);

    // Initialize input signals with artificial data
    std::uint32_t seed = (unsigned)time(NULL); // Get RNG seed value
    oneapi::mkl::rng::mcg31ml engine(Q, seed); // Initialize RNG engine
                                                // Set RNG distribution
    oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
        rng_distribution(-0.00005, 0.00005);

    auto evt1 =
        oneapi::mkl::rng::generate(rng_distribution, engine, N, sig1); // Noise
    auto evt2 = oneapi::mkl::rng::generate(rng_distribution, engine, N, sig2);
    evt1.wait();
    evt2.wait();

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

    clock_t start_time = clock(); // Start timer

    // Initialize FFT descriptor
    oneapi::mkl::dft::descriptor<oneapi::mkl::dft::precision::SINGLE,
                                oneapi::mkl::dft::domain::REAL>
        transform_plan(N);
    transform_plan.commit(Q);

    // Perform forward transforms on real arrays
    evt1 = oneapi::mkl::dft::compute_forward(transform_plan, sig1);
    evt2 = oneapi::mkl::dft::compute_forward(transform_plan, sig2);

    // Compute: DFT(sig1) * CONJG(DFT(sig2))
    oneapi::mkl::vm::mulbyconj(
        Q, N / 2, reinterpret_cast<std::complex<float> *>(sig1),
        reinterpret_cast<std::complex<float> *>(sig2),
```

```

    reinterpret_cast<std::complex<float> *>(corr), {evt1, evt2})
    .wait();

    // Perform backward transform on complex correlation array
    oneapi::mkl::dft::compute_backward(transform_plan, corr).wait();

    clock_t end_time = clock(); // Stop timer
    std::cout << "The 1D correlation (N = " << N << ") took "
        << float(end_time - start_time) / CLOCKS_PER_SEC << " seconds."
        << std::endl;

    // Find the shift that gives maximum correlation value
    float max_corr = 0.0;
    int shift = 0;
    for (unsigned int idx = 0; idx < N; idx++) {
        if (corr[idx] > max_corr) {
            max_corr = corr[idx];
            shift = idx;
        }
    }
    int _N = static_cast<int>(N);
    shift =
        (shift > _N / 2) ? shift - _N : shift; // Treat the signals as circularly
                                                // shifted versions of each other.
    std::cout << "Shift the second signal " << shift
        << " elements relative to the first signal to get a maximum, "
        "normalized correlation score of "
        << max_corr / N << "." << std::endl;

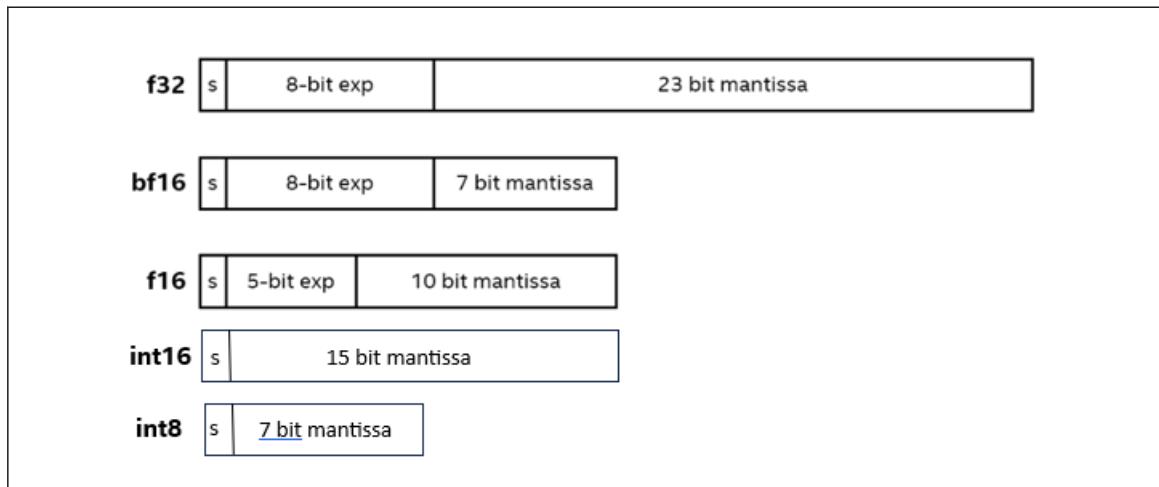
    // Cleanup
    sycl::free(sig1, sycl_context);
    sycl::free(sig2, sycl_context);
    sycl::free(corr, sycl_context);
}

```

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 SYCL. 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.

## Boost Matrix Multiplication Performance with Intel® Xe Matrix Extensions

The increasing popularity of Artificial Intelligence (AI) in today's world demands the introduction of low precision data types and hardware support for these data types to boost application performance. Low precision models are faster in computation and have smaller memory footprints. For the same reason low precision data types are getting highly used for both training and inference in AI / machine learning (ML) even though `float32` is the default data type. To optimize and support these low precision data types, special hardware features and instructions are required. Intel provides those in the form of Intel® Xe Matrix Extensions (Intel® XMX) in its GPUs. Some of the most used 16-bit formats and 8-bit formats are `float16` (`fp16`), `bfloat16` (`bf16`), 16-bit integer (`int16`), 8-bit integer (`int8`) etc. The figure below visualizes the differences between some of these formats.



In the above figure, **s** is the signed bit(the first digit of the binary presentation, 0 implies positive number and 1 implies negative number) and **exp** is the exponent.

## Intel® Xe Matrix Extensions

Intel® Xe Matrix Extensions (Intel® XMX) specializes in executing Dot Product Accumulate Systolic (DPAS) instructions on 2D systolic arrays. A systolic array in parallel computer architecture is a homogeneous network of tightly coupled data processing units. Each unit computes a partial result as a function of data received from its upstream neighbors, stores the result within itself and passes it downstream. Intel® XMX supports numerous data types, depending on hardware generation, such as int8, fp16, bf16, and tf32. To understand Intel® XMX inside Intel® Data Center GPU Max Series, please refer to Intel® Intel® Iris® Xe GPU Architecture section.

## Programming Intel® XMX

Users can interact with XMX at many different levels: from deep learning frameworks, dedicated libraries, custom SYCL kernels, down to low-level intrinsics. Programming and running applications using Intel® XMX requires Intel® oneAPI Base Toolkit.

### Using Intel® oneAPI Deep Neural Network Library (oneDNN)

To take the maximum advantage of the hardware, oneDNN has enabled Intel® XMX support on Intel GPUs (Intel® Xe 4th Generation Scalable processors and later) by default. To uses the data types supported by XMX and oneDNN, the applications needs to be built with GPU support enabled.

The [Matrix Multiplication Performance](#) bundled with oneDNN is a good example to learn how to use oneDNN to program Intel® XMX.

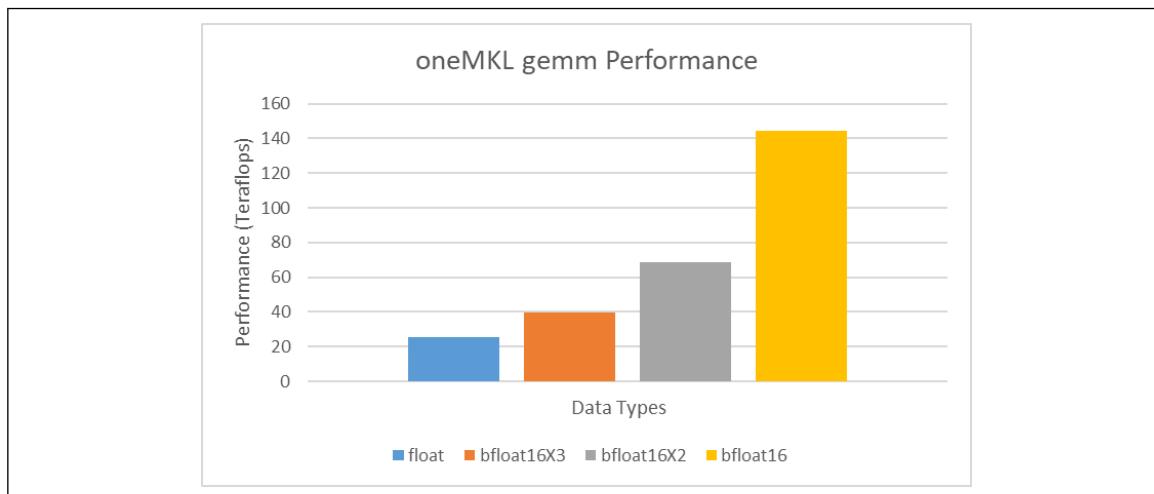
### Using Intel® oneAPI Math Kernel Library (oneMKL)

Like oneDNN, oneMKL also enables Intel® XMX by default if we use the supported data types and the code is compiled using the Intel® oneAPI DPC++ Compiler.

oneMKL supports several algorithms for accelerating single-precision `gemm` and `gemm_batch` using XMX. The `bf16x2` and `bf16x3` are 2 such algorithms using bf16 to approximate single-precision `gemm`.

Internally single-precision input data is converted into bf16 and multiplied with the systolic array. The three variants – bf16, bf16x2, and bf16x3 – allow you to make a tradeoff between accuracy and performance, with bf16 being the fastest and bf16x3 the most accurate (similar to the accuracy of standard single-precision `gemm`). The example [Matrix Multiplication](#) shows how to use these algorithms and the table below compares the performance difference.

Precision	Data type/ Algorithm	Peak (TF)	Performance relative to theoretical peak of single precision(%)
Single	fp32	26	98
Single	bf16	151	577
Single	bf16x2	74	280
Single	bf16x3	42	161



The test is performed on a Intel® Xeon® 8480+ with 512 GB DDR5-4800 + Intel® Data Center GPU Max 1550 running Ubuntu 22.04.

This table shows the Performance of bf16, bf16x2 and bf16x3 far outweigh the theoretical peak of single precision. If the accuracy tradeoff is acceptable, bf16, followed by bf16x2 and then bf16x3, is highly recommended.

## References

1. [Matrix Multiplication Performance](#)
2. [Matrix Multiplication](#)

## Host/Device Memory, Buffer and USM

Accelerators have access to a rich memory hierarchy. Utilizing the right level in the hierarchy is critical to getting the best performance.

In this section we cover topics related to declaration, movement, and access to the memory hierarchy.

The API allows sharing of memory objects across different device processes. Since each process has its own virtual address space, there is no guarantee that the same virtual address will be available when the memory object is shared in new process. There are a set of APIs that make it easier to share the memory objects.

To learn more about using the oneAPI Level Zero API for memory sharing see Inter-Process Communication in the Level Zero Specification.

- [Unified Shared Memory Allocations](#)
- [Performance Impact of USM and Buffers](#)

- [Avoiding Moving Data Back and Forth between Host and Device](#)
- [Optimizing Data Transfers](#)
- [Avoiding Declaring Buffers in a Loop](#)
- [Buffer Accessor Modes](#)

## 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 SYCL:

`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.

### 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-stack, 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 stacks 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 SYCL USM model, we added three 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 maps to:

```
extern void *omp_target_alloc_device(size_t size, int device_num)
```

## Performance Impact of USM and Buffers

SYCL offers several choices for managing memory on the device. This section discusses the performance tradeoffs, briefly introducing the concepts. For an in-depth explanation, see [Data Parallel C++](#).

As with other language features, the specification defines the behavior but not the implementation, so performance characteristics can change between software versions and devices. This guide 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 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 two 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 choices by showing the same example program written with the three models. To highlight 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.

```
void serial(int stride) {
    // Allocate and initialize data
    float *data = new float[data_size];
    init(data);

    timer it;

    for (int i = 0; i < time_steps; i++) {
        for (int j = 0; j < data_size; j++) {
            for (int k = 0; k < device_steps; k++)
                data[j] += 1.0;
        }

        for (int j = 0; j < data_size; j += stride)
            data[j] += 1.0;
    }
    put_elapsed_time(it);

    check(data);

    delete[] data;
} // serial
```

## 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_dataaccessor` 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 reading/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 data is moved back to the device, which effectively delays launching the kernel.

```
void buffer_data(int stride) {
    // Allocate buffer, initialize on host
    sycl::buffer<float> buffer_data{data_size};
    init(sycl::host_accessor(buffer_data, sycl::write_only, sycl::no_init));

    timer it;
    for (int i = 0; i < time_steps; i++) {

        // Compute on device
        q.submit([&](auto &h) {
            sycl::accessor device_data(buffer_data, h);

            auto compute = [=](auto id) {
                for (int k = 0; k < device_steps; k++)
                    device_data[id] += 1.0;
            };
            h.parallel_for(data_size, compute);
        });
    }
}
```

```

    });

    // Compute on host
    sycl::host_accessor host_data(buffer_data);
    for (int i = 0; i < data_size; i += stride)
        host_data[i] += 1.0;
}
put_elapsed_time(it);

const sycl::host_accessor h(buffer_data);
check(h);
} // 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 concern 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.

### 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.

```

void shared_usm_data(int stride) {
    float *data = sycl::malloc_shared<float>(data_size, q);
    init(data);

    timer it;

    for (int i = 0; i < time_steps; i++) {
        auto compute = [=](auto id) {
            for (int k = 0; k < device_steps; k++)
                data[id] += 1.0;
        };
        q.parallel_for(data_size, compute).wait();

        for (int k = 0; k < data_size; k += stride)
            data[k] += 1.0;
    }
}

```

```
    }
    q.wait();
    put_elapsed_time(it);

    check(data);

    sycl::free(data, q);
} // 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.

## 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.

```
void device_usm_data(int stride) {
    // Allocate and initialize host data
    float *host_data = new float[data_size];
    init(host_data);

    // Allocate device data
    float *device_data = sycl::malloc_device<float>(data_size, q);

    timer it;

    for (int i = 0; i < time_steps; i++) {
        // Copy data to device and compute
        q.memcpy(device_data, host_data, sizeof(float) * data_size);
        auto compute = [=](auto id) {
            for (int k = 0; k < device_steps; k++)
                device_data[id] += 1.0;
        };
        q.parallel_for(data_size, compute);

        // Copy data to host and compute
        q.memcpy(host_data, device_data, sizeof(float) * data_size).wait();
        for (int k = 0; k < data_size; k += stride)
            host_data[k] += 1.0;
    }
    q.wait();
    put_elapsed_time(it);

    check(host_data);
```

```

    sycl::free(device_data, q);
    delete[] host_data;
} // 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 the host, so it has neither the page faults overhead of shared allocations, nor the overloading overhead associated with buffers. Shared allocations only transfer 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.

## Avoiding 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 new vector is then transformed into a third vector by applying a function on each value and this third vector is finally 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. In machine learning, for example, models are structured as layers of computations, and output of one layer is input to the next layer.

```

double myFunc1(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b,
               AlignedVector<int> &c, AlignedVector<int> &d,
               AlignedVector<int> &res, int iter) {
    sycl::range num_items{a.size()};
    VectorAllocator<int> alloc;
    AlignedVector<int> sum(a.size(), alloc);

    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
    sycl::buffer a_buf(a, props);
    sycl::buffer b_buf(b, props);
    sycl::buffer c_buf(b, props);
    sycl::buffer d_buf(b, props);
    sycl::buffer res_buf(res, props);
    sycl::buffer sum_buf(sum.data(), num_items, props);

    Timer timer;
    for (int i = 0; i < iter; i++) {
        // kernel1
        q.submit([&](auto &h) {
            // Input accessors
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            // Output accessor
            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

            h.parallel_for(num_items,
                          [=](auto id) { sum_acc[id] = a_acc[id] + b_acc[id]; });
        });
    }
}

```

```

{
    sycl::host_accessor h_acc(sum_buf);
    for (int j = 0; j < a.size(); j++)
        if (h_acc[j] > 10)
            h_acc[j] = 1;
        else
            h_acc[j] = 0;
}

// kernel2
q.submit([&](auto &h) {
    // Input accessors
    sycl::accessor sum_acc(sum_buf, h, sycl::read_only);
    sycl::accessor c_acc(c_buf, h, sycl::read_only);
    sycl::accessor d_acc(d_buf, h, sycl::read_only);
    // Output accessor
    sycl::accessor res_acc(res_buf, h, sycl::write_only, sycl::no_init);

    h.parallel_for(num_items, [=](auto id) {
        res_acc[id] = sum_acc[id] * c_acc[id] + d_acc[id];
    });
});
q.wait();
}
double elapsed = timer.Elapsed() / iter;
return (elapsed);
} // end myFunc1

```

Instead of bringing the data to the host and applying the function to the data and sending it back to the device, you can create a **kernel3** to execute this function on the device, as shown in the following example. The kernel **kernel3** operates on the intermediate data in **accum\_buf** in between **kernel1** and **kernel2**, avoiding the round trip of data transfer between the device and the host.

```

double myFunc2(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b,
                AlignedVector<int> &c, AlignedVector<int> &d,
                AlignedVector<int> &res, int iter) {
    sycl::range num_items{a.size()};
    VectorAllocator<int> alloc;
    AlignedVector<int> sum(a.size(), alloc);

    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
    sycl::buffer a_buf(a, props);
    sycl::buffer b_buf(b, props);
    sycl::buffer c_buf(c, props);
    sycl::buffer d_buf(d, props);
    sycl::buffer res_buf(res, props);
    sycl::buffer sum_buf(sum.data(), num_items, props);

    Timer timer;
    for (int i = 0; i < iter; i++) {
        // kernel1
        q.submit([&](auto &h) {
            // Input accessors
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            // Output accessor
            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
        });
    }
}
```

```

    h.parallel_for(num_items,
                  [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
}

// kernel3
q.submit([&](auto &h) {
    sycl::accessor sum_acc(sum_buf, h, sycl::read_write);
    h.parallel_for(num_items, [=](auto id) {
        if (sum_acc[id] > 10)
            sum_acc[id] = 1;
        else
            sum_acc[id] = 0;
    });
});

// kernel2
q.submit([&](auto &h) {
    // Input accessors
    sycl::accessor sum_acc(sum_buf, h, sycl::read_only);
    sycl::accessor c_acc(c_buf, h, sycl::read_only);
    sycl::accessor d_acc(d_buf, h, sycl::read_only);
    // Output accessor
    sycl::accessor res_acc(res_buf, h, sycl::write_only, sycl::no_init);

    h.parallel_for(num_items, [=](auto i) {
        res_acc[i] = sum_acc[i] * c_acc[i] + d_acc[i];
    });
    q.wait();
}
double elapsed = timer.Elapsed() / iter;
return (elapsed);
} // 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. The SYCL compiler cannot do this kind of optimization. In some specific domains like machine learning, there are graph compilers that operate on the ML models and fuse the operations, which has the same impact.

```

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

    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
    sycl::buffer a_buf(a, props);
    sycl::buffer b_buf(b, props);
    sycl::buffer c_buf(c, props);
    sycl::buffer d_buf(d, props);
    sycl::buffer res_buf(res, props);
    sycl::buffer sum_buf(sum.data(), num_items, props);

    Timer timer;
    for (int i = 0; i < iter; i++) {
        // kernel1
        q.submit([&](auto &h) {

```

```

// Input accessors
sycl::accessor a_acc(a_buf, h, sycl::read_only);
sycl::accessor b_acc(b_buf, h, sycl::read_only);
// Output accessor
sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

h.parallel_for(num_items, [=](auto i) {
    int t = a_acc[i] + b_acc[i];
    if (t > 10)
        sum_acc[i] = 1;
    else
        sum_acc[i] = 0;
});
});

// kernel2
q.submit([&](auto &h) {
    // Input accessors
    sycl::accessor sum_acc(sum_buf, h, sycl::read_only);
    sycl::accessor c_acc(c_buf, h, sycl::read_only);
    sycl::accessor d_acc(d_buf, h, sycl::read_only);
    // Output accessor
    sycl::accessor res_acc(res_buf, h, sycl::write_only, sycl::no_init);

    h.parallel_for(num_items, [=](auto i) {
        res_acc[i] = sum_acc[i] * c_acc[i] + d_acc[i];
    });
});
q.wait();
}
double elapsed = timer.Elapsed() / iter;
return (elapsed);
} // 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.

```

double myFunc4(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b,
               AlignedVector<int> &c, AlignedVector<int> &d,
               AlignedVector<int> &res, int iter) {
    sycl::range num_items{a.size()};
    VectorAllocator<int> alloc;

    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
    sycl::buffer a_buf(a, props);
    sycl::buffer b_buf(b, props);
    sycl::buffer c_buf(b, props);
    sycl::buffer d_buf(b, props);
    sycl::buffer res_buf(res, props);

    Timer timer;
    for (int i = 0; i < iter; i++) {
        // kernel1
        q.submit([&](auto &h) {
            // Input accessors
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);

```

```

sycl::accessor c_acc(c_buf, h, sycl::read_only);
sycl::accessor d_acc(d_buf, h, sycl::read_only);
// Output accessor
sycl::accessor res_acc(res_buf, h, sycl::write_only, sycl::no_init);

h.parallel_for(num_items, [=](auto i) {
    int t = a_acc[i] + b_acc[i];
    if (t > 10)
        res_acc[i] = c_acc[i] + d_acc[i];
    else
        res_acc[i] = d_acc[i];
});
q.wait();
}
double elapsed = timer.Elapsed() / iter;
return (elapsed);
} // end myFunc4

```

## Optimizing Data Transfers

### Introduction

The previous section discussed minimizing data transfers between host and device. This section discusses ways to optimize data transfers when there is a need to move data between host and device.

When moving data between host and device, data transfer rate is maximized when both source and destination are in Unified Shared Memory (USM).

In SYCL, data transfers between host and device may be explicit via the use of the SYCL `memcpy` function, or implicit via the use of SYCL buffers and accessors.

### Case 1: Data transfers using buffers

In the case of SYCL buffers constructed with a pointer to pre-allocated system memory (i.e., memory allocated using `malloc` or `new`), the SYCL runtime has the capability to automatically promote system memory to host USM at buffer creation, and to release it at buffer destruction. To enable this capability, a buffer has to be created with a `hostptr`, for example:

```

int* hostptr = (int*)malloc(4*sizeof(int));
buffer<int, 1> Buffer(hostptr,4);

```

And then the environment variable `SYCL_USM_HOSTPTR_IMPORT=1` should be set at runtime.

### Case 2: Data transfers using SYCL data movement APIs

When the host data allocation is under user control, the user may use USM functions such as `malloc_host` to allocate host USM memory instead of system memory (i.e., memory allocated using `malloc` or `new`), if the allocated pointer is going to be used in a data transfer API.

If the source code where the allocation is made is not available, or source code changes are prohibited for portability reasons, then system memory can be promoted (imported) to host USM before the first data transfer, and released from host USM after all data transfers are completed. A set of APIs are provided for explicit import/release.

## Experimental SYCL Import/Release APIs

The following SYCL import and release APIs give the programmer explicit control over the range and duration of imports. The APIs are analogous to the `malloc_host` APIs:

```
void* sycl::prepare_for_device_copy
(void* ptr, size_t numBytes, sycl::context& syclContext)

void* sycl::prepare_for_device_copy
(void* ptr, size_t numBytes, sycl::queue& syclQueue)

void sycl::release_from_device_copy
(void* ptr, sycl::context& syclContext)

void sycl::release_from_device_copy
(void* ptr, sycl::queue& syclQueue)
```

The APIs are simple to use, but the onus is on the user to ensure correctness and safety with respect to the lifetime ranges.

Note that if `numBytes` is not the same as the size of the `malloc`'ed memory block (for example, if the `malloc`'ed memory is 1024 bytes, but `numBytes` is 512 or 2048), the guidance here would be to use a size for the import that matches the data transfer size since that much memory is expected to be allocated. If the true allocation is bigger, then importing less than the true allocation has no ill effects. Importing more than the true allocation is a user error (as is the transfer).

### Note:

- The import/release APIs are experimental and are subject to change.

### SYCL Example

The following example, `bench_10import_sycl.cpp`, measures the rate of data transfer between host and device. The data transfer size can be varied. The program prints device-to-host and host-to-device data transfer rate, measured in Gigabytes per second. Switches passed to the program are used to control data transfer direction, whether or not the SYCL import feature is used, and a range of transfer sizes. The switches are listed in the example source.

In the program, the import is done once at the beginning, and the release is done once at the end, using the following calls:

```
sycl::ext::oneapi::experimental::prepare_for_device_copy(
    hostp, transfer_upper_limit, dq);

sycl::ext::oneapi::experimental::release_from_device_copy(hostp, dq);
```

In between the above calls, there is a loop that repeatedly does `memcpy` involving the imported pointer `hostp`.

```
#include <math.h>
#include <stdlib.h>
#include <chrono>
#include <time.h>
#include <unistd.h>
#include <sycl/sycl.hpp>

using namespace sycl;

static const char *usage_str =
    "\n ze_bandwidth [OPTIONS]"
    "\n"
    "\n OPTIONS:"
```

```
"\n  -t, string          selectively run a particular test:\n    h2d or H2D        run only Host-to-Device tests\n    d2h or D2H        run only Device-to-Host tests\n  "\n  -q                  [default: both]\n  minimal output\n  -v                  [default: disabled]\n  enable verification\n  -i                  [default: disabled]\n  set number of iterations per transfer\n  [default: 500]\n  -s                  select only one transfer size (bytes)\n  -sb                 select beginning transfer size (bytes)\n  [default: 1]\n  -se                select ending transfer size (bytes)\n  [default: 2^28]\n  -l                  use SYCL prepare_for_device_copy/release_from_device_copy APIs\n  [default: disabled]\n  -h, --help           display help message\n"\n;\n\nstatic uint32_t sanitize_ulong(char *in) {\n    unsigned long temp = strtoul(in, NULL, 0);\n    if (ERANGE == errno) {\n        fprintf(stderr, "%s out of range of type ulong\n", in);\n    } else if (temp > UINT32_MAX) {\n        fprintf(stderr, "%ld greater than UINT32_MAX\n", temp);\n    } else {\n        return static_cast<uint32_t>(temp);\n    }\n    return 0;\n}\n\nsize_t transfer_lower_limit = 1;\nsize_t transfer_upper_limit = (1 << 28);\nbool verify = false;\nbool run_host2dev = true;\nbool run_dev2host = true;\nbool verbose = true;\nbool l0import = false;\nuint32_t ntimes = 500;\n\n// kernel latency\nint main(int argc, char **argv) {\n    for (int i = 1; i < argc; i++) {\n        if ((strcmp(argv[i], "-h") == 0) || (strcmp(argv[i], "--help") == 0)) {\n            std::cout << usage_str;\n            exit(0);\n        } else if (strcmp(argv[i], "-q") == 0) {\n            verbose = false;\n        } else if (strcmp(argv[i], "-v") == 0) {\n            verify = true;\n        } else if (strcmp(argv[i], "-l") == 0) {\n            l0import = true;\n        } else if (strcmp(argv[i], "-i") == 0) {\n            if ((i + 1) < argc) {\n                ntimes = sanitize_ulong(argv[i + 1]);\n                i++;\n            }\n        }\n    }\n}
```

```
    } else if (strcmp(argv[i], "-s") == 0) {
        if ((i + 1) < argc) {
            transfer_lower_limit = sanitize_ulong(argv[i + 1]);
            transfer_upper_limit = transfer_lower_limit;
            i++;
        }
    } else if (strcmp(argv[i], "-sb") == 0) {
        if ((i + 1) < argc) {
            transfer_lower_limit = sanitize_ulong(argv[i + 1]);
            i++;
        }
    } else if (strcmp(argv[i], "-se") == 0) {
        if ((i + 1) < argc) {
            transfer_upper_limit = sanitize_ulong(argv[i + 1]);
            i++;
        }
    } else if ((strcmp(argv[i], "-t") == 0)) {
        run_host2dev = false;
        run_dev2host = false;

        if ((i + 1) >= argc) {
            std::cout << usage_str;
            exit(-1);
        }
        if ((strcmp(argv[i + 1], "h2d") == 0) ||
            (strcmp(argv[i + 1], "H2D") == 0)) {
            run_host2dev = true;
            i++;
        } else if ((strcmp(argv[i + 1], "d2h") == 0) ||
                   (strcmp(argv[i + 1], "D2H") == 0)) {
            run_dev2host = true;
            i++;
        } else {
            std::cout << usage_str;
            exit(-1);
        }
    } else {
        std::cout << usage_str;
        exit(-1);
    }
}

queue dq;
device dev = dq.get_device();
size_t max_compute_units = dev.get_info<info::device::max_compute_units>();
auto BE = dq.get_device()
    .template get_info<sycl::info::device::opencl_c_version>()
    .empty()
? "L0"
: "OpenCL";
if (verbose)
    std::cout << "Device name " << dev.get_info<info::device::name>() << " "
    << "max_compute_units"
    << " " << max_compute_units << ", Backend " << BE << "\n";

void *hostp;
posix_memalign(&hostp, 4096, transfer_upper_limit);
memset(hostp, 1, transfer_upper_limit);
```

```
if (l0import) {
    if (verbose)
        std::cout << "Doing L0 Import\n";
    sycl::ext::oneapi::experimental::prepare_for_device_copy(
        hostp, transfer_upper_limit, dq);
}

void *destp =
    malloc_device<char>(transfer_upper_limit, dq.get_device(), dq.get_context());
dq.submit([&](handler &cgh) { cgh.memset(destp, 2, transfer_upper_limit); });
dq.wait();

if (run_host2dev) {
    if (!verbose)
        printf("SYCL USM API (%s)\n", BE);
    for (size_t s = transfer_lower_limit; s <= transfer_upper_limit; s <= 1) {
        auto start_time = std::chrono::steady_clock::now();
        for (int i = 0; i < ntimes; ++i) {
            dq.submit([&](handler &cgh) { cgh.memcpy(destp, hostp, s); });
            dq.wait();
        }
        auto end_time = std::chrono::steady_clock::now();
        std::chrono::duration<double> seconds = end_time - start_time;

        if (verbose)
            printf("HosttoDevice: %8lu bytes, %7.3f ms, %8.3g GB/s\n", s,
                   1000 * seconds.count() / ntimes,
                   1e-9 * s / (seconds.count() / ntimes));
        else
            printf("%10.6f\n", 1e-9 * s / (seconds.count() / ntimes));
    }
}

if (run_dev2host) {
    if (!verbose)
        printf("SYCL USM API (%s)\n", BE);
    for (size_t s = transfer_lower_limit; s <= transfer_upper_limit; s <= 1) {
        auto start_time = std::chrono::steady_clock::now();
        for (int i = 0; i < ntimes; ++i) {
            dq.submit([&](handler &cgh) { cgh.memcpy(hostp, destp, s); });
            dq.wait();
        }
        auto end_time = std::chrono::steady_clock::now();
        std::chrono::duration<double> seconds = end_time - start_time;

        if (verbose)
            printf("DeviceToHost: %8lu bytes, %7.3f ms, %8.3g GB/s\n", s,
                   seconds.count(), 1e-9 * s / (seconds.count() / ntimes));
        else
            printf("%10.6f\n", 1e-9 * s / (seconds.count() / ntimes));
    }
}

if (l0import)
    sycl::ext::oneapi::experimental::release_from_device_copy(hostp, dq);
```

```

    free(hostp);
    free(destp, dq.get_context());
}

```

### Compilation command:

```
$ icpx -fsycl bench_10import_sycl.cpp
```

### Example run command:

```
$ a.out -l -t h2d -s 256000000
```

The above run command specifies doing the import (-l), host-to-device (-t h2d), and transferring 256 million bytes (-s 256000000).

### Date Transfer Rate Measurements

We use the program, `bench_10import_sycl.cpp`, to measure the rate of data transfer between host and device (with and without import/release). The transfer rate for a data size of 256 million bytes when running on the particular GPU used (1-stack only) was as follows.

Run Command	Transfer Direction	Use API?	Tra (GB/sec)
	impo rt/ relea se		nsf er Ra te
a.out -t h2d -s 256000000	host to device	No	26.9
a.out -l -t h2d -s 256000000	host to device	Yes	45.4
a.out -t d2h -s 256000000	device to host	No	33.2
a.out -l -t d2h -s 256000000	device to host	Yes	48.0

### Performance Recommendations

For optimal data transfer between host and device, we recommend the following approaches:

1. Allocate data that will be the source or destination of data transfer between host and device in host USM memory. This can be done using the API `malloc_host` in SYCL. This way, no extra copies are necessary and the data transfer rate is optimal.
2. If the above option (1) is not feasible, then use explicit import/ release APIs. In SYCL, use the `prepare_for_device_copy` and `release_from_device_copy` APIs listed above.

### References

1. [sycl\\_ext\\_oneapi\\_copy\\_optimize](#)
2. [Environment Variables — oneAPI DPC++ Compiler Documentation](#)

### Avoiding 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.

```
#include <CL/sycl.hpp>
#include <stdio.h>

constexpr int N = 25;
constexpr int STEPS = 100000;

int main() {

    int AData[N];
    int BData[N];
    int CData[N];

    sycl::queue Q;

    // Create 2 buffers, each holding N integers
    sycl::buffer<int> ABuf(&AData[0], N);
    sycl::buffer<int> BBuf(&BData[0], N);

    Q.submit([&](auto &h) {
        // Create device accessors.
        // The property no_init lets the runtime know that the
        // previous contents of the buffer can be discarded.
        sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
        sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
        h.parallel_for(N, [=](auto i) {
            aA[i] = 10;
            aB[i] = 20;
        });
    });

    for (int j = 0; j < STEPS; j++) {
        sycl::buffer<int> CBuf(&CData[0], N);
        Q.submit([&](auto &h) {
            // Create device accessors.
            sycl::accessor aA(ABuf, h);
            sycl::accessor aB(BBuf, h);
            sycl::accessor aC(CBuf, h);
            h.parallel_for(N, [=](auto i) {
                aC[i] = (aA[i] < aB[i]) ? -1 : 1;
                aA[i] += aC[i];
                aB[i] -= aC[i];
            });
        });
    } // end for

    // Create host accessors.
    const sycl::host_accessor haA(ABuf);
    const sycl::host_accessor haB(BBuf);
    printf("%d %d\n", haA[N / 2], haB[N / 2]);

    return 0;
}
```

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.

```
#include <CL/sycl.hpp>
#include <stdio.h>

constexpr int N = 25;
constexpr int STEPS = 100000;

int main() {

    int AData[N];
    int BData[N];
    int CData[N];

    sycl::queue Q;

    // Create 3 buffers, each holding N integers
    sycl::buffer<int> ABuf(&AData[0], N);
    sycl::buffer<int> BBuf(&BData[0], N);
    sycl::buffer<int> CBuf(&CData[0], N);

    Q.submit([&](auto &h) {
        // Create device accessors.
        // The property no_init lets the runtime know that the
        // previous contents of the buffer can be discarded.
        sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
        sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
        h.parallel_for(N, [=](auto i) {
            aA[i] = 10;
            aB[i] = 20;
        });
    });

    for (int j = 0; j < STEPS; j++) {
        Q.submit([&](auto &h) {
            // Create device accessors.
            sycl::accessor aA(ABuf, h);
            sycl::accessor aB(BBuf, h);
            sycl::accessor aC(CBuf, h);
            h.parallel_for(N, [=](auto i) {
                aC[i] = (aA[i] < aB[i]) ? -1 : 1;
                aA[i] += aC[i];
                aB[i] -= aC[i];
            });
        });
    } // end for

    // Create host accessors.
    const sycl::host_accessor haA(ABuf);
    const sycl::host_accessor haB(BBuf);
    printf("%d %d\n", haA[N / 2], haB[N / 2]);

    return 0;
}
```

## Buffer Accessor Modes

In SYCL, 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. The 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 is 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.

The following example shows 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.

```
#include <CL/sycl.hpp>
#include <stdio.h>

constexpr int N = 100;

int main() {

    int AData[N];
    int BData[N];
    int CData[N];

    sycl::queue Q;

    // Kernel1
    {
        // Create 3 buffers, each holding N integers
        sycl::buffer<int> ABuf(&AData[0], N);
        sycl::buffer<int> BBuf(&BData[0], N);
        sycl::buffer<int> CBuf(&CData[0], N);

        Q.submit([](auto &h) {
            // Create device accessors.
            // The property no_init lets the runtime know that the
            // previous contents of the buffer can be discarded.
            sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
            sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
            sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);
        });
    }
}
```

```

    h.parallel_for(N, [=](auto i) {
        aA[i] = 11;
        aB[i] = 22;
        aC[i] = 0;
    });
});
} // end Kernel1

// Kernel2
{
    // Create 3 buffers, each holding N integers
    sycl::buffer<int> ABuf(&AData[0], N);
    sycl::buffer<int> BBuf(&BData[0], N);
    sycl::buffer<int> CBuf(&CData[0], N);

    Q.submit([&](auto &h) {
        // Create device accessors
        sycl::accessor aA(ABuf, h, sycl::read_only);
        sycl::accessor aB(BBuf, h, sycl::read_only);
        sycl::accessor aC(CBuf, h);
        h.parallel_for(N, [=](auto i) { aC[i] += aA[i] + aB[i]; });
    });
} // end Kernel2

// Buffers are destroyed and so CData is updated and can be accessed
for (int i = 0; i < N; i++) {
    printf("%d\n", CData[i]);
}

return 0;
}

```

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, because 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.

```

#include <CL/sycl.hpp>
#include <stdio.h>

constexpr int N = 100;

int main() {

    int AData[N];
    int BData[N];
    int CData[N];

    sycl::queue Q;

```

```

// Create 3 buffers, each holding N integers
sycl::buffer<int> ABuf(&AData[0], N);
sycl::buffer<int> BBuf(&BData[0], N);
sycl::buffer<int> CBuf(&CData[0], N);

// Kernel1
Q.submit([&](auto &h) {
    // Create device accessors.
    // The property no_init lets the runtime know that the
    // previous contents of the buffer can be discarded.
    sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
    sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
    sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);

    h.parallel_for(N, [=](auto i) {
        aA[i] = 11;
        aB[i] = 22;
        aC[i] = 0;
    });
});

// Kernel2
Q.submit([&](auto &h) {
    // Create device sycl::accessors
    sycl::accessor aA(ABuf, h, sycl::read_only);
    sycl::accessor aB(BBuf, h, sycl::read_only);
    sycl::accessor aC(CBuf, h);
    h.parallel_for(N, [=](auto i) { aC[i] += aA[i] + aB[i]; });
});

// The host accessor creation will ensure that a wait for kernel to finish
// is triggered and data from device to host is copied
sycl::host_accessor h_acc(CBuf);
for (int i = 0; i < N; i++) {
    printf("%d\n", h_acc[i]);
}

return 0;
}

```

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.

```

#include <CL/sycl.hpp>
#include <stdio.h>

constexpr int N = 100;

int main() {

    int AData[N];
    int BData[N];
    int CData[N];

    sycl::queue Q;
    {

```

```

// Create 3 buffers, each holding N integers
sycl::buffer<int> ABuf(&AData[0], N);
sycl::buffer<int> BBuf(&BData[0], N);
sycl::buffer<int> CBuf(&CData[0], N);

// Kernel1
Q.submit([&](auto &h) {
    // Create device accessors.
    // The property no_init lets the runtime know that the
    // previous contents of the buffer can be discarded.
    sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
    sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
    sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);

    h.parallel_for(N, [=](auto i) {
        aA[i] = 11;
        aB[i] = 22;
        aC[i] = 0;
    });
});

// Kernel2
Q.submit([&](auto &h) {
    // Create device accessors
    sycl::accessor aA(ABuf, h, sycl::read_only);
    sycl::accessor aB(BBuf, h, sycl::read_only);
    sycl::accessor aC(CBuf, h);
    h.parallel_for(N, [=](auto i) { aC[i] += aA[i] + aB[i]; });
});

}

// Since the buffers are going out of scope, they will have to be
// copied back from device to host and this will require a wait for
// all the kernels to finish and so no explicit wait is needed
for (int i = 0; i < N; i++) {
    printf("%d\n", CData[i]);
}

return 0;
}

```

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.

```

#include <CL/sycl.hpp>
#include <stdio.h>

constexpr int N = 100;
constexpr int iters = 100;

int main() {

    int AData[N];
    int BData[N];
    int CData[N];

    sycl::queue Q;
    sycl::buffer<int> CBuf(&CData[0], N);

```

```

{
    // Create 2 buffers, each holding N integers
    sycl::buffer<int> ABuf(&AData[0], N);
    sycl::buffer<int> BBuf(&BData[0], N);

    // Kernel1
    Q.submit([&](auto &h) {
        // Create device accessors.
        // The property no_init lets the runtime know that the
        // previous contents of the buffer can be discarded.
        sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
        sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
        sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);

        h.parallel_for(N, [=](auto i) {
            aA[i] = 11;
            aB[i] = 22;
            aC[i] = 0;
        });
    });
}

for (int it = 0; it < iters; it++) {
    // Kernel2
    Q.submit([&](auto &h) {
        // Create device accessors
        sycl::accessor aC(CBuf, h);
        h.parallel_for(N, [=](auto i) { aC[i] += AData[i] + BData[i]; });
    });
}

sycl::host_accessor h_acc(CBuf);
for (int i = 0; i < N; i++) {
    printf("%d\n", h_acc[i]);
}

return 0;
}

```

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.

```

#include <CL/sycl.hpp>
#include <stdio.h>

constexpr int N = 100;
constexpr int iters = 100;

int main() {

    int AData[N];
    int BData[N];
    int CData[N];

    sycl::queue Q;
    sycl::buffer<int> CBuf(&CData[0], N);

```

```
{  
    // Create 2 buffers, each holding N integers  
    sycl::buffer<int> ABuf(&AData[0], N);  
    sycl::buffer<int> BBuf(&BData[0], N);  
  
    // Kernel1  
    Q.submit([&](auto &h) {  
        // Create device accessors.  
        // The property no_init lets the runtime know that the  
        // previous contents of the buffer can be discarded.  
        sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);  
        sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);  
        sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);  
  
        h.parallel_for(N, [=](auto i) {  
            aA[i] = 11;  
            aB[i] = 22;  
            aC[i] = 0;  
        });  
    });  
}  
  
sycl::buffer<int> ABuf(&AData[0], N);  
sycl::buffer<int> BBuf(&BData[0], N);  
for (int it = 0; it < iters; it++) {  
    // Kernel2  
    Q.submit([&](auto &h) {  
        // Create device accessors  
        sycl::accessor aA(ABuf, h, sycl::read_only);  
        sycl::accessor aB(BBuf, h, sycl::read_only);  
        sycl::accessor aC(CBuf, h);  
        h.parallel_for(N, [=](auto i) { aC[i] += aA[i] + aB[i]; });  
    });  
}  
  
sycl::host_accessor h_acc(CBuf);  
for (int i = 0; i < N; i++) {  
    printf("%d\n", h_acc[i]);  
}  
  
return 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.

- [Asynchronous and Overlapping Data Transfers Between Host and Device](#)

### Asynchronous and Overlapping Data Transfers Between Host and Device

An accelerator is a 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.

## Bandwidth Between Host and Device

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.

### 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 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).

```
float ComputeParallel1(sycl::queue &q, std::vector<float> &data) {
    const size_t data_size = data.size();
    float sum = 0;
    static float *accum = 0;

    if (data_size > 0) {
        const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
        int num_EUs =
            q.get_device().get_info<sycl::info::device::max_compute_units>();
        int vec_size =
            q.get_device()
                .get_info<sycl::info::device::native_vector_width_float>();
        int num_processing_elements = num_EUs * vec_size;
        int BATCH = (N + num_processing_elements - 1) / num_processing_elements;
        sycl::buffer<float> buf(data.data(), data.size(), props);
        sycl::buffer<float> accum_buf(accum, num_processing_elements, props);

        if (!accum)
            accum = new float[num_processing_elements];

        q.submit([&](auto &h) {
            sycl::accessor buf_acc(buf, h, sycl::read_only);
            sycl::accessor accum_acc(accum_buf, h, sycl::write_only);
            for (int i = 0; i < num_processing_elements; ++i) {
                accum_acc[i] = 0;
            }
            for (int i = 0; i < data_size; ++i) {
                accum_acc[buf_acc[i]] += data[i];
            }
        });
    }
}
```

```

sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
h.parallel_for(num_processing_elements, [=](auto index) {
    size_t glob_id = index[0];
    size_t start = glob_id * BATCH;
    size_t end = (glob_id + 1) * BATCH;
    if (end > N)
        end = N;
    float sum = 0.0;
    for (size_t i = start; i < end; i++)
        sum += buf_acc[i];
    accum_acc[glob_id] = sum;
});
});
q.wait();
sycl::host_accessor h_acc(accum_buf);
for (int i = 0; i < num_processing_elements; i++)
    sum += h_acc[i];
}
return sum;
} // 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.

```

float ComputeParallel2(sycl::queue &q, std::vector<float> &data) {
    const size_t data_size = data.size();
    float sum = 0;
    static float *accum = 0;

    if (data_size > 0) {
        const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
        int num_EUs =
            q.get_device().get_info<sycl::info::device::max_compute_units>();
        int vec_size =
            q.get_device()
                .get_info<sycl::info::device::native_vector_width_float>();
        int num_processing_elements = num_EUs * vec_size;
        int BATCH = (N + num_processing_elements - 1) / num_processing_elements;
        sycl::buffer<float> buf(data.data(), data.size(), props);
        sycl::buffer<float> accum_buf(accum, num_processing_elements, props);
        sycl::buffer<float> res_buf(&sum, 1, props);
        if (!accum)
            accum = new float[num_processing_elements];

        q.submit([&](auto &h) {
            sycl::accessor buf_acc(buf, h, sycl::read_only);
            sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
            h.parallel_for(num_processing_elements, [=](auto index) {
                size_t glob_id = index[0];
                size_t start = glob_id * BATCH;
                size_t end = (glob_id + 1) * BATCH;
                if (end > N)
                    end = N;
                float sum = 0.0;
                for (size_t i = start; i < end; i++)

```

```

        sum += buf_acc[i];
        accum_acc[glob_id] = sum;
    });
});

q.submit([&](auto &h) {
    sycl::accessor accum_acc(accum, h, sycl::read_only);
    sycl::accessor res_acc(res_buf, h, sycl::write_only, sycl::no_init);
    h.parallel_for(1, [=](auto index) {
        res_acc[index] = 0;
        for (size_t i = 0; i < num_processing_elements; i++)
            res_acc[index] += accum_acc[i];
    });
});
}

// Buffers go out of scope and data gets transferred from device to host
return sum;
} // end ComputeParallel2
}

```

## 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 57), processed (line 60), and the result (line 63) 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 are copy engines that can be used to transfer data between host and device, we can see that the operations 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 Vector Engines and one by a copy engine, or both are executed by copy engines.
- 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 Vector Engines.

```

#include <CL/sycl.hpp>

#define NITERS 10
#define KERNEL_ITERS 10000
#define NUM_CHUNKS 10
#define CHUNK_SIZE 10000000

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_;
};

int main() {
    const int num_chunks = NUM_CHUNKS;
}

```

```
const int chunk_size = CHUNK_SIZE;
const int iter = NITERS;

sycl::queue q;

// Allocate and initialize host data
float *host_data[num_chunks];
for (int c = 0; c < num_chunks; c++) {
    host_data[c] = sycl::malloc_host<float>(chunk_size, q);
    float val = c;
    for (int i = 0; i < chunk_size; i++)
        host_data[c][i] = val;
}
std::cout << "Allocated host data\n";

// Allocate and initialize device memory
float *device_data[num_chunks];
for (int c = 0; c < num_chunks; c++) {
    device_data[c] = sycl::malloc_device<float>(chunk_size, q);
    float val = 1000.0;
    q.fill<float>(device_data[c], val, chunk_size);
}
q.wait();
std::cout << "Allocated device data\n";

Timer timer;
for (int it = 0; it < iter; it++) {
    for (int c = 0; c < num_chunks; c++) {
        auto add_one = [=](auto id) {
            for (int i = 0; i < KERNEL_ITERS; i++)
                device_data[c][id] += 1.0;
        };
        // Copy-in not dependent on previous event
        auto copy_in =
            q.memcpy(device_data[c], host_data[c], sizeof(float) * chunk_size);
        // Compute waits for copy_in
        auto compute = q.parallel_for(chunk_size, copy_in, add_one);
        auto cg = [=](auto &h) {
            h.depends_on(compute);
            h.memcpy(host_data[c], device_data[c], sizeof(float) * chunk_size);
        };
        // Copy out waits for compute
        auto copy_out = q.submit(cg);
    }

    q.wait();
}

auto elapsed = timer.Elapsed() / iter;
for (int c = 0; c < num_chunks; c++) {
    for (int i = 0; i < chunk_size; i++) {
        if (host_data[c][i] != (float)((c + KERNEL_ITERS * iter))) {
            std::cout << "Mismatch for chunk: " << c << " position: " << i
                << " expected: " << c + 10000 << " got: " << host_data[c][i]
                << "\n";
            break;
        }
    }
}
```

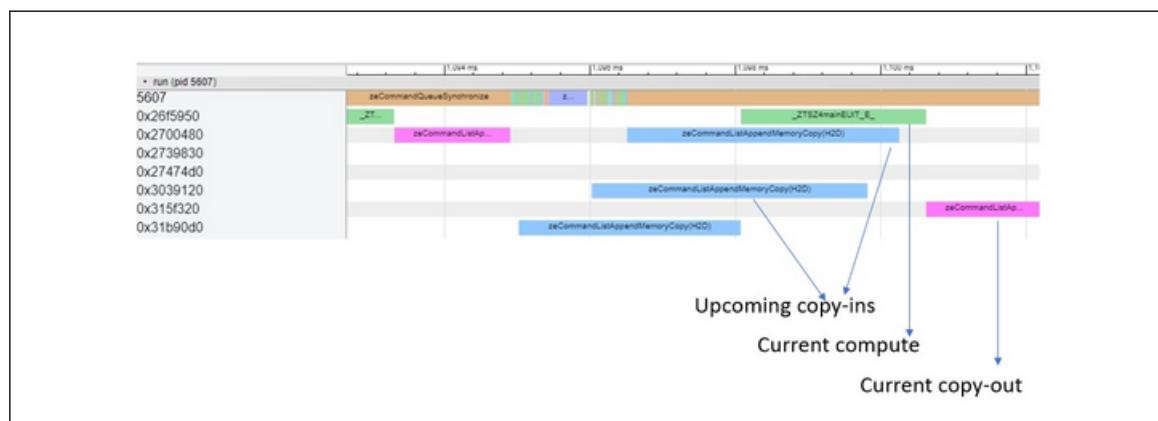
```

    }
    std::cout << "Time = " << elapsed << " usecs\n";
}

```

In the timeline picture below, which is collected using **onetrace**, we can see that copy-ins from upcoming iterations overlap with the execution of compute kernel. Also, we see multiple copy-ins executing in parallel on multiple copy engines.

### Copy Overlaps with Compute Kernel Execution



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.)

## Using Multiple Heterogeneous Devices

Most accelerators 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. SYCL 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.

### Overlapping Compute on Various Devices

SYCL 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 SYCL 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 SYCL 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.

The following example shows a simple vector add operation that works on a single GPU device.

```

size_t VectorAdd1(sycl::queue &q, const IntArray &a, const IntArray &b,
                  IntArray &sum, int iter) {
    sycl::range num_items{a.size()};

    sycl::buffer a_buf(a);

```

```

sycl::buffer b_buf(b);
sycl::buffer sum_buf(sum.data(), num_items);
auto start = std::chrono::steady_clock::now();
for (int i = 0; i < iter; i++) {

    auto e = q.submit([&] (auto &h) {
        // Input accessors
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        // Output accessor
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

        h.parallel_for(num_items,
                      [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
    });
}

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

```

In the following kernel the input vector is split into two parts and computation is done on two different accelerators (one CPU and one 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 metrics 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.

```

size_t VectorAdd2(sycl::queue &q1, sycl::queue &q2, const IntArray &a,
                  const IntArray &b, IntArray &sum, int iter) {
    sycl::range num_items{a.size() / 2};

    auto start = std::chrono::steady_clock::now();
    {
        sycl::buffer a1_buf(a.data(), num_items);
        sycl::buffer b1_buf(b.data(), num_items);
        sycl::buffer sum1_buf(sum.data(), num_items);

        sycl::buffer a2_buf(a.data() + a.size() / 2, num_items);
        sycl::buffer b2_buf(b.data() + a.size() / 2, num_items);
        sycl::buffer sum2_buf(sum.data() + a.size() / 2, num_items);
        for (int i = 0; i < iter; i++) {

            q1.submit([&] (auto &h) {
                // Input accessors
                sycl::accessor a_acc(a1_buf, h, sycl::read_only);
                sycl::accessor b_acc(b1_buf, h, sycl::read_only);
                // Output accessor
                sycl::accessor sum_acc(sum1_buf, h, sycl::write_only, sycl::no_init);

                h.parallel_for(num_items,
                              [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
            });
            // do the work on host
        }
    }
}

```

```

q2.submit([&](auto &h) {
    // Input accessors
    sycl::accessor a_acc(a2_buf, h, sycl::read_only);
    sycl::accessor b_acc(b2_buf, h, sycl::read_only);
    // Output accessor
    sycl::accessor sum_acc(sum2_buf, h, sycl::write_only, sycl::no_init);

    h.parallel_for(num_items,
                  [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
});

}

// On some platforms this explicit flush of queues is needed
// to ensure the overlap in execution between the CPU and GPU
// cl_command_queue cq = q1.get();
// clFlush(cq);
// cq=q2.get();
// clFlush(cq);
}

q1.wait();
q2.wait();
auto end = std::chrono::steady_clock::now();
std::cout << "Vector add2 completed on device - took "
      << (end - start).count() << " u-secs\n";
return ((end - start).count());
} // 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, you will have to split the work in proportion to the capability of the accelerator, instead of distributing it evenly as was done in the above example.

## Compilation

oneAPI has multiple types of compilation. The main source to the application is compiled, and 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.

- [Just-In-Time Compilation](#)
- [Ahead-Of-Time Compilation](#)
- [Specialization Constants](#)
- [Accuracy Versus Performance Tradeoffs in Floating-Point Computations](#)

### Just-In-Time Compilation

The Intel® oneAPI SYCL Compiler converts a SYCL 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.

```
#include <CL/sycl.hpp>
#include <array>
#include <chrono>
```

```
#include <iostream>

// Array type and data size for this example.
constexpr size_t array_size = (1 << 16);
typedef std::array<int, array_size> IntArray;

void VectorAdd1(sycl::queue &q, const IntArray &a, const IntArray &b,
                IntArray &sum) {
    sycl::range num_items{a.size()};

    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf(sum.data(), num_items);

    auto e = q.submit([&] (auto &h) {
        // Input accessors
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        // Output accessor
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

        h.parallel_for(num_items,
                      [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
    });
    q.wait();
}

void VectorAdd2(sycl::queue &q, const IntArray &a, const IntArray &b,
                IntArray &sum) {
    sycl::range num_items{a.size()};

    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf(sum.data(), num_items);

    auto e = q.submit([&] (auto &h) {
        // Input accessors
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        // Output accessor
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

        h.parallel_for(num_items,
                      [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
    });
    q.wait();
}

void InitializeArray(IntArray &a) {
    for (size_t i = 0; i < a.size(); i++)
        a[i] = i;
}

int main() {
    IntArray a, b, sum;

    InitializeArray(a);
    InitializeArray(b);
```

```

sycl::queue q(sycl::default_selector_v,
              sycl::property::queue::enable_profiling{});

std::cout << "Running on device: "
       << q.get_device().get_info<sycl::info::device::name>() << "\n";
std::cout << "Vector size: " << a.size() << "\n";
auto start = std::chrono::steady_clock::now();
VectorAdd1(q, a, b, sum);
auto end = std::chrono::steady_clock::now();
std::cout << "Initial Vector add1 successfully completed on device - took "
       << (end - start).count() << " nano-secs\n";

start = std::chrono::steady_clock::now();
VectorAdd1(q, a, b, sum);
end = std::chrono::steady_clock::now();
std::cout << "Second Vector add1 successfully completed on device - took "
       << (end - start).count() << " nano-secs\n";

start = std::chrono::steady_clock::now();
VectorAdd2(q, a, b, sum);
end = std::chrono::steady_clock::now();
std::cout << "Initial Vector add2 successfully completed on device - took "
       << (end - start).count() << " nano-secs\n";

start = std::chrono::steady_clock::now();
VectorAdd2(q, a, b, sum);
end = std::chrono::steady_clock::now();
std::cout << "Second Vector add2 successfully completed on device - took "
       << (end - start).count() << " nano-secs\n";
return 0;
}

```

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

```
icpx -fsycl -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.

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 JIT compilation on all the kernels in the binary at the beginning of execution, while lazy JIT compilation will enable JIT compilation 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`: performs eager JIT compilation of all kernels in the application

- *auto*: the default, the compiler will use its heuristic to select the best way of splitting device code for JIT compilation

If the above program is compiled with this option:

```
icpx -fsycl -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

```
icpx -fsycl -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

```
icpx -fsycl -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.

## Ahead-Of-Time Compilation

The overhead of JIT compilation at runtime can be avoided by Ahead-Of-Time (AOT) compilation. With AOT compilation, the binary will contain the actual assembly code of the platform that was selected at compile time 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.

When compiling in Ahead-Of-Time (AOT) mode for an Intel® GPU, one need to add an extra compiler option (`-Xs`) to indicate the specific GPU target.

### AOT Compiler Options for SYCL

Intel® Data Center GPU Max Series:

```
icpx -fsycl -fsycl-targets=spir64_gen -Xs "-device pvc" source.c
```

### AOT Compiler Options for OpenMP

Intel® Data Center GPU Max Series:

```
icx -fopenmp -fopenmp-targets=spir64_gen -Xs "-device pvc" source.c
```

### Notes

- The compiler options shown above can also be used when compiling OpenMP Fortran programs in AOT mode (using `ifx`).
- In JIT-mode, the Intel® Graphics Compiler knows the type of the hardware and will adjust automatically. The extra `-revision_id` option is only needed for AOT mode.

### Specialization Constants

SYCL 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.

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

class specialized_kernel;

// const static identifier of specialization constant
const static sycl::specialization_id<float> value_id;

// Fetch a value at runtime.
float get_value() { return 10; }

int main() {
    sycl::queue queue;

    std::vector<float> vec(1);
    {
        sycl::buffer<float> buffer(vec.data(), vec.size());
        queue.submit([&](auto &cgh) {
            sycl::accessor acc(buffer, cgh, sycl::write_only, sycl::no_init);

            // Set value of specialization constant.
            cgh.template set_specialization_constant<value_id>(get_value());

            // Runtime builds the kernel with specialization constant
            // replaced by the literal value provided in the preceding
            // call of `set_specialization_constant<value_id>`
            cgh.template single_task<specialized_kernel>(
                [=](sycl::kernel_handler kh) {
                    const float val = kh.get_specialization_constant<value_id>();
                    acc[0] = val;
                });
        });
    }
    queue.wait_and_throw();

    std::cout << vec[0] << std::endl;

    return 0;
}
```

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:

```
#include <CL/sycl.hpp>
#include <chrono>
#include <vector>

class specialized_kernel;
class literal_kernel;
```

```
// const static identifier of specialization constant
const static sycl::specialization_id<float> value_id;

// Fetch a value at runtime.
float get_value() { return 10; }

int main() {
    sycl::queue queue;

    // Get kernel ID from kernel class qualifier
    sycl::kernel_id specialized_kernel_id =
        sycl::get_kernel_id<specialized_kernel>();

    // Construct kernel bundle with only specialized_kernel in the input state
    sycl::kernel_bundle kb_src =
        sycl::get_kernel_bundle<sycl::bundle_state::input>(
            queue.get_context(), {specialized_kernel_id});
    // set specialization constant value
    kb_src.set_specialization_constant<value_id>(get_value());

    auto start = std::chrono::steady_clock::now();
    // build the kernel bundle for the set value
    sycl::kernel_bundle kb_exe = sycl::build(kb_src);
    auto end = std::chrono::steady_clock::now();
    std::cout << "specialization took - " << (end - start).count()
        << " nano-secs\n";

    std::vector<float> vec{0, 0, 0, 0, 0};
    sycl::buffer<float> buffer1(vec.data(), vec.size());
    sycl::buffer<float> buffer2(vec.data(), vec.size());
    start = std::chrono::steady_clock::now();
{
    queue.submit([&] (auto &cgh) {
        sycl::accessor acc(buffer1, cgh, sycl::write_only, sycl::no_init);

        // use the precompiled kernel bundle in the executable state
        cgh.use_kernel_bundle(kb_exe);

        cgh.template single_task<specialized_kernel>(
            [=](sycl::kernel_handler kh) {
                float v = kh.get_specialization_constant<value_id>();
                acc[0] = v;
            });
    });
    queue.wait_and_throw();
}
end = std::chrono::steady_clock::now();

{
    sycl::host_accessor host_acc(buffer1, sycl::read_only);
    std::cout << "result1 (c): " << host_acc[0] << " " << host_acc[1] << " "
        << host_acc[2] << " " << host_acc[3] << " " << host_acc[4]
        << std::endl;
}
std::cout << "execution took : " << (end - start).count() << " nano-secs\n";

start = std::chrono::steady_clock::now();
{
```

```

queue.submit([&](auto &cgh) {
    sycl::accessor acc(buffer2, cgh, sycl::write_only, sycl::no_init);
    cgh.template single_task<literal_kernel>([=]{} { acc[0] = 20; });
});
queue.wait_and_throw();
}
end = std::chrono::steady_clock::now();

{
    sycl::host_accessor host_acc(buffer2, sycl::read_only);
    std::cout << "result2 (c): " << host_acc[0] << " " << host_acc[1] << " "
        << host_acc[2] << " " << host_acc[3] << " " << host_acc[4]
        << std::endl;
}
std::cout << "execution took - " << (end - start).count() << " nano-secs\n";
}

```

Looking at 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 because this kernel will have been JIT-compiled by the runtime before actually executing it.

Below we provide some examples showing simple use cases and applications of specialization constants.

## Simple Trip Count Use Case

The following example performs a summation and uses specialization constants to set the trip count.

```

#include <CL/sycl.hpp>

class SpecializedKernel;

// Identify the specialization constant.
constexpr sycl::specialization_id<int> nx_sc;

int main(int argc, char *argv[]) {
    sycl::queue queue;

    std::cout << "Running on "
        << queue.get_device().get_info<sycl::info::device::name>() << "\n";

    std::vector<float> vec(1);
    {
        sycl::buffer<float> buf(vec.data(), vec.size());

        // Application execution stops here asking for input from user
        int Nx;
        if (argc > 1) {
            Nx = std::stoi(argv[1]);
        } else {
            Nx = 1024;
        }

        std::cout << "Nx = " << Nx << std::endl;

        queue.submit([&](sycl::handler &h) {
            sycl::accessor acc(buf, h, sycl::write_only, sycl::no_init);

            // set specialization constant with runtime variable
            h.set_specialization_constant<nx_sc>(Nx);
        });
    }
}

```

```

    h.single_task<SpecializedKernel>([=](sycl::kernel_handler kh) {
        // nx_sc value here will be input value provided at runtime and
        // can be optimized because JIT compiler now treats it as a constant.
        int runtime_const_trip_count = kh.get_specialization_constant<nx_sc>();
        int accum = 0;
        for (int i = 0; i < runtime_const_trip_count; i++) {
            accum = accum + i;
        }
        acc[0] = accum;
    });
}
std::cout << vec[0] << std::endl;
return 0;
}

```

The goal is to specialize the trip count variable `Nx` for the loop in the kernel. Since the user inputs the trip count after execution of the program starts, the host compiler does not know the value of `Nx`. The input value can be passed as a specialization constant to the JIT compiler, allowing the JIT compiler to apply some optimizations such as unrolling the loop.

Without the specialization constants feature, the variable `Nx` would need to be a constant expression for the whole program to achieve this. In this way, specialization constants can lead to more optimization and hence faster kernel code, by creating constant values from runtime variables.

In contrast, the host compiler cannot effectively optimize the example loop below where the trip count is not a constant, since it needs runtime checks for safety/legality.

```

for (int i = 0; i < Nx; i++) {
    // Optimizations are limited when Nx is not a constant.
}

```

## Modified STREAM Triad Application

In the following example (a modified STREAM Triad) we have the classic STREAM Triad with several multiply and add operations where the variable `multiplier` and the number of multiply-add operations are determined by an input variable.

Below is a snippet of the original kernel code run on the device. The runtime variable `inner_loop_size` is used to set the loop upper bound.

```

auto q0_event = q.submit([&](sycl::handler &h) {
    h.parallel_for<non_specialized_kernel>(array_size / 2, [=](auto idx) {
        // set trip count to runtime variable
        auto runtime_trip_count_const = inner_loop_size;
        auto accum = 0;
        for (size_t j = 0; j < runtime_trip_count_const; j++) {
            auto multiplier = scalar * j;
            accum = accum + A0[idx] + B0[idx] * multiplier;
        }
        C0[idx] = accum;
    });
});

q.wait();

cl_ulong exec_time_ns0 =
    q0_event

```

```

        .get_profiling_info<sycl::info::event_profiling::command_end>() -
q0_event
        .get_profiling_info<sycl::info::event_profiling::command_start>();

std::cout << "Execution time (iteration " << i
              << ") [sec]: " << (double)exec_time_ns0 * 1.0E-9 << "\n";
min_time_ns0 = std::min(min_time_ns0, exec_time_ns0);

```

In order to improve performance, we use the specialization constant feature to specialize the variable `inner_loop_size`. Below is a snippet of the kernel code run on the device - using a specialization constant.

```

auto q0_event = q.submit([&](sycl::handler &h) {
    // set specialization constant using runtime variable
    h.set_specialization_constant<trip_sc>(inner_loop_size);
    h.parallel_for<specialized_kernel>(
        array_size / 2, [=](auto idx, sycl::kernel_handler kh) {
            // set trip count to the now known specialization constant
            auto runtime_trip_count_const =
                kh.get_specialization_constant<trip_sc>();
            auto accum = 0;
            for (size_t j = 0; j < runtime_trip_count_const; j++) {
                auto multiplier = scalar * j;
                accum = accum + A0[idx] + B0[idx] * multiplier;
            }
            C0[idx] = accum;
        });
    });

q.wait();

cl_ulong exec_time_ns0 =
    q0_event
        .get_profiling_info<sycl::info::event_profiling::command_end>() -
q0_event
        .get_profiling_info<sycl::info::event_profiling::command_start>();

std::cout << "Execution time (iteration " << i
              << ") [sec]: " << (double)exec_time_ns0 * 1.0E-9 << "\n";

```

We finally compare the specialization trip count value with the following example that uses a regular constant value. Below is a snippet of the kernel code run on the device using a regular constant.

```

auto q0_event = q.submit([&](sycl::handler &h) {
    h.parallel_for<regular_constant_kernel>(array_size / 2, [=](auto idx) {
        // set trip count to known regular constant
        size_t runtime_trip_count_const = 10;
        auto accum = 0;
        for (size_t j = 0; j < runtime_trip_count_const; j++) {
            auto multiplier = scalar * j;
            accum = accum + A0[idx] + B0[idx] * multiplier;
        }
        C0[idx] = accum;
    });
});

q.wait();

cl_ulong exec_time_ns0 =
    q0_event

```

```

        .get_profiling_info<sycl::info::event_profiling::command_end>() -
q0_event
        .get_profiling_info<sycl::info::event_profiling::command_start>();

std::cout << "Execution time (iteration " << i
              << ") [sec]: " << (double)exec_time_ns0 * 1.0E-9 << "\n";
min_time_ns0 = std::min(min_time_ns0, exec_time_ns0);

```

Timings from the runs of the three different versions are displayed below. The stream size represents the size of arrays A0, B0, and C0. The inner trip count represents the value of the `runtime_trip_count_const` variable set using the specialization constant.

Displayed below are timing outputs for example runs of the different versions using a stream size of 134217728 elements (1024 MB) and an inner trip count of 10 as inputs.

Run with runtime variable: Time in sec (fastest run): 0.00161008

Run with specialization constant: Time in sec (fastest run): 0.00156256

Run with constant value: Time in sec (fastest run): 0.00155104

The results, as expected, show that using the specialization constant improves the performance of the computation on the device from the execution time seen with the runtime variable to one that more closely matches the execution time seen with the constant value. Furthermore, analysis of the generated code shows the specialized version of the application unrolls the main loop due to its added capability to specialize the loop trip count & JIT compile it as a known constant. We witness about `inner_loop_size` times (thus 10 times in this example) as many floating-point add instructions in the main loop of the program using specialization constants as compared to the one using a runtime variable.

## Accuracy Versus Performance Tradeoffs in Floating-Point Computations

Programmers of floating-point applications typically aim for the following two objectives:

- Accuracy: Produce results that are “close” to the result of the exact calculation.
- Performance: Produce an application that runs as fast as possible.

The two objectives usually conflict. However, good programming practices and judicious use of compiler options allow you to control the tradeoffs.

For more information, see the article: [Consistency of Floating-Point Results using the Intel Compiler or Why doesn't my application always give the same answer?](#), by Dr. Martyn J. Corden and David Kreitzer (2018)

In this section, we present some mechanisms (compiler options and source-level changes) that allow the programmer to control the semantics (and hence the accuracy and performance) of floating-point computations on the host as well as on the device. We describe compiler options for OpenMP and SYCL programs, and describe source-level changes in SYCL.

### OpenMP

In OpenMP, the following `-fp-model` options may be used to control the semantics of floating-point computations on the host as well as on the device.

**1. `-fp-model=precise`**

This option tells the compiler to strictly adhere to value-safe optimizations when implementing floating-point calculations. It disables optimizations (such as re-association, multiply by reciprocal, and zero folding) that can change the result of floating-point calculations. The increased accuracy that comes with `-fp-model=precise` may result in lower performance.

**1. `-fp-model=fast`**

This option is the default for both host and device compilations at -O2 and above. The option tells the compiler to use more aggressive optimizations when implementing floating-point calculations. These optimizations increase speed but may affect the accuracy or reproducibility of floating-point computations.

In C/C++, the `-fp-model=fast` option is equivalent to the `-ffast-math` option. With this option (at -O2 and above), all 7 fast-math flags (`nnan`, `ninf`, `nsz`, `arcp`, `contract`, `afn`, `reassoc`) are set by the C/C++ front-end. (See <https://llvm.org/docs/LangRef.html#fast-math-flags> for a description of the fast-math flags in LLVM.)

In Fortran, on the other hand, the language rules dictate that we cannot set the `nnan` flag (No NaNs) by default. So the `-fp-model=fast` option (at -O2 and above) only sets the other 6 fast-math flags (`ninf`, `nsz`, `arcp`, `contract`, `afn`, `reassoc`). To set all 7 fast-math flags in Fortran, use the `-ffast-math` option.

### **1. `-Xopenmp-target-backend "-options -cl-fast-relaxed-math"`**

The `-fp-model=fast` (or `-ffast-math`) option does not enable native math instructions on the Intel GPU (Intel® Data Center GPU Max Series). You need to compile with `-Xopenmp-target-backend "-options -cl-fast-relaxed-math"` to get native math instructions on the GPU. Native math instructions give even lower accuracy than what is allowed with `-fp-model=fast`.

`-Xopenmp-target-backend "-options -cl-fast-relaxed-math"` passes the `-cl-fast-relaxed-math` option to the backend in the compilation tool chain for the device. `-cl-fast-relaxed-math` relaxes the precision of commonly used math functions on the device. It offers a quick way to get performance gains for kernels with many math library function calls, as long as the accuracy requirements are satisfied by what is provided through native math instructions.

The `-cl-fast-relaxed-math` option affects the compilation of the entire program and does not permit fine control of the resulting numeric accuracy.

Note that Intel GPUs support native for single precision (float, real) only.

#### **Notes (OpenMP):**

- When `-fp-model` is specified on the compilation command line (outside of the `-fopenmp-targets=spir64="..."` set of options), the `-fp-model` option applies to both the host and the device compilations. For example, the following compilation command specifies `-fp-model=precise` for both the host and the device compilations:

```
icx/icpx/ifx -O2 -fiopenmp -fopenmp-targets=spir64 -fp-model=precise
```

You can specify different ```-fp-model``` keywords for the host and the device compilations as shown below.

- To specify `-fp-model=fast` for the host compilation, and `-fp-model=precise` for the device compilation:

```
icx/icpx/ifx -O2 -fiopenmp -fopenmp-targets=spir64="-fp-model=precise" -fp-model=fast
```

Or:

```
icx/icpx/ifx -O2 -fiopenmp -fopenmp-targets=spir64="-fp-model=precise"  
(No need to specify ``-fp-model=fast`` since it is the default at -O2 or higher.)
```

- To specify `-fp-model=precise` for the host compilation, and `-fp-model=fast` for the device compilation:

```
icx/icpx/ifx -O3 -fiopenmp -fopenmp-targets=spir64="-fp-model=fast" -fp-model=precise
```

- To specify `-fp-model=fast` for the host compilation, and `relaxed-math` for the device compilation:

```
icx/icpx/ifx -O2 -fiopenmp -fopenmp-targets=spir64 -Xopenmp-target-backend "-options -cl-fast-relaxed-math" -fp-model=fast
```

Or:

```
icx/ifx -O2 -fiopenmp -fopenmp-targets=spir64 -Xopenmp-target-backend "-options -cl-fast-relaxed-math"
(No need to explicitly specify ``-fp-model=fast`` since it is the default at -O2 or higher.)
```

- You can combine the fast, precise, and relaxed-math options with `-fimf-precision=low` (medium or high) option to fine-tune precision on the host side. The `-fimf-precision` option is not supported on the device side currently.

The following table shows a summary of how to set the various options (precise, fast-math, relaxed math) in OpenMP to (a) both the host and the device (second column); (b) to the host only (third column); and (c) to the device only (fourth column).

### OpenMP - Summary of Options

Floating Point Semantics	Apply to Host and Device Compilations	Apply to Host Compilation Only	Apply to Device Compilation Only
Precise	<code>-fp-model=precise</code>	<code>-fp-model=precise, and specify ``-fiopenmp -fopenmp-targets=spir64=-fp-model=fast``</code>	<code>-fiopenmp -fopenmp-targets=spir64="-fp-model=precise"</code>
Fast-math	<code>-fp-model=fast (default)</code>	<code>-fp-model=fast, and specify -fiopenmp -fopenmp-targets=spir64="-fp-model=precise"</code>	<code>-fp-model=precise, and specify -fiopenmp -fopenmp-targets=spir64="-fp-model=fast"</code>
Relaxed-math (native instructions)	Applies to device only	Applies to device only	<code>-Xopenmp-target-backend "-options -cl-fast-relaxed-math"</code>

### SYCL

In SYCL, as in OpenMP, the `-fp-model=fast` and `-fp-model=precise` options may be used for both host and device compilations.

In SYCL, `-fp-model=fast` option is equivalent to the `-ffast-math` option, and is the default for both host and device compilations at `-O2` and above.

To specify relaxed-math for device compilation, use the compiler option `-Xsycl-target-backend "-options -cl-fast-relaxed-math"`. You need to compile with this option to get native math instructions on the GPU.

Note that SYCL (and Intel GPUs) support native for single precision (float, real) only.

### Notes (SYCL):

- When `-fp-model` is specified on the compilation command line (outside of any `-Xsycl-target` option), the `-fp-model` option applies to both the host and the device compilations. For example, the following compilation command specifies `-fp-model=precise` for both the host and the device compilations:

```
icx/icpx/ifx -fsycl -fp-model=precise
```

You can specify different ```-fp-model``` keywords for the host and device compilations as shown below.

- To specify `-fp-model=fast` for the host compilation, and `-fp-model=precise` for the device compilation:

```
icx/icpx -fsycl -Xsycl-target-frontend "-fp-model=precise" -fp-model=fast
```

Or:

```
icx/icpx -fsycl -Xsycl-target-frontend "-fp-model=precise"
(No need to specify ``-fp-model=fast`` since it is the default at -O2 or higher.)
```

- To specify `-fp-model=precise` for the host compilation, and `-fp-model=fast` for the device compilation:

```
icx/icpx -fsycl -Xsycl-target-frontend "-fp-model=fast" -fp-model=precise
```

- To specify `-fp-model=fast` for the host compilation, and `relaxed-math` for the device compilation:

```
icx/icpx -fsycl -Xsycl-target-backend "-options -cl-fast-relaxed-math" -fp-model=fast
```

Or:

```
icx/icpx -fsycl -Xsycl-target-backend "-options -cl-fast-relaxed-math"
(No need to specify ``-fp-model=fast`` since it is the default at -O2 or higher.)
```

The following table shows a summary of how to set the various options (math, fast-math, relaxed math) in SYCL to (a) both the host and the device (second column); (b) to the host only (third column); and (c) to the device only (fourth column).

## SYCL - Summary of Options

Floating Point Semantics	Apply to Host and Device Compilations	Apply to Host Compilation Only	Apply to Device Compilation Only
Precise	<code>-fp-model=precise</code>	<code>-fp-model=precise</code> , and specify <code>-Xsycl-target-frontend "-fp-model=fast"</code>	<code>-Xsycl-target-frontend "-fp-model=precise"</code>
Fast-math	<code>-fp-model=fast</code> (default)	<code>-fp-model=fast</code> , and specify <code>-Xsycl-target-frontend "-fp-model=precise"</code>	<code>-fp-model=precise</code> , and specify <code>-Xsycl-target-frontend "-fp-model=fast"</code>
Relaxed-math (native instructions)	Applies to device only	Applies to device only	<code>-Xsycl-target-backend "-options -cl-fast-relaxed-math"</code>

## Guidelines

In general, here are some guidelines for which options to use:

- Do not specify inconsistent options. The result will be unpredictable.
- The most commonly used option is `-fp-model=fast` for both host and device.
- Use `relaxed-math` for best performance on the device.
- Use `-fp-model=precise` for highest precision.

## Example: The log Function

The math library (a component of a programming language's standard library) contains functions (or subroutines) for the most common mathematical functions, such as exponential, logarithmic, power, and trigonometric functions.

Different implementations of the math library functions may not have the same accuracy or be rounded in the same way. The value returned by a math library function may vary between one compiler release and another, due to algorithmic and optimization changes.

The accuracy of a math library function can be controlled via compiler options or via the source code. We use the `log` (natural logarithm) math function as an example to illustrate this.

### OpenMP / C++ (test\_log\_omp.cpp)

The following is an OpenMP C++ program that calls the `std::log` function on the device (from inside OpenMP target regions). The program includes the `cmath` header file which contains definitions for common math functions.

```
#include <iostream>
#include <assert.h>
#include <chrono>
#include <cmath>

#if FP_SIZE == 32
    typedef float FP_TYPE;
    static constexpr FP_TYPE VALIDATION_THRESHOLD = 1e-3;
#elif FP_SIZE == 64
    typedef double FP_TYPE;
    static constexpr FP_TYPE VALIDATION_THRESHOLD = 1e-6;
#endif

template<typename T>
void do_work (unsigned NELEMENTS, unsigned NREPETITIONS, T initial_value, T *res)
{
    #pragma omp target teams distribute parallel for map(present,alloc:res[0:NELEMENTS])
    for (unsigned j = 0; j < NELEMENTS; j++)
    {
        T tmp = initial_value;
        for (unsigned i = 0; i < NREPETITIONS; ++i)
            tmp += std::log(tmp);
        res[j] = tmp;
    }
}

int main (int argc, char *argv[])
{
    static constexpr unsigned NELEMENTS = 64*1024*1024;
    static constexpr unsigned NREPETITIONS = 1024;

    #pragma omp target
    {

        FP_TYPE initial_value = 2;
        FP_TYPE ref_res = initial_value;
```

```

for (unsigned i = 0; i < NREPETITIONS; ++i)
    ref_res += std::log(ref_res);
std::cout << "reference result = " << ref_res << std::endl;

{
    FP_TYPE * std_res = new FP_TYPE[NELEMENTS];
    assert (std_res != nullptr);

    std::chrono::duration<float, std::micro> elapsed;
    #pragma omp target data map(std_res[0:NELEMENTS])
    {
        auto tbegin = std::chrono::system_clock::now();
        do_work<FP_TYPE> (NELEMENTS, NREPETITIONS, initial_value, std_res);
        auto tend = std::chrono::system_clock::now();
        elapsed = tend - tbegin;
    }
    std::cout << "std::log result[0] = " << std_res[0] << std::endl;

    bool allequal = true;
    for (auto i = 1; i < NELEMENTS; ++i)
        allequal = allequal and std_res[0] == std_res[i];
    if (allequal)
    {
        if (std::abs(ref_res - std_res[0])/std::abs(ref_res) <
            std::abs(VALIDATION_THRESHOLD))
            std::cout << "std::log validates. Total execution time is " << elapsed.count()
            << " us." << std::endl;
        else
            std::cout << "std::log does not validate (ref=" << ref_res << " std_res=" <<
            std_res[0] << " mix=" << std::abs(ref_res - std_res[0])/std::abs(ref_res) << ")" << std::endl;
    }
    else
        std::cout << "std::log does not validate, results are not equal." << std::endl;

    delete [] std_res;
}

return 0;
}

```

Sample compilation and run commands for test\_log\_omp.cpp:

```

icpx -O2 -fopenmp -fopenmp-targets=spir64 test_log_omp.cpp \
    -DREAL_ELEMENT -DFP_SIZE=64 -fp-model=fast -fopenmp-version=51

OMP_TARGET_OFFLOAD=MANDATORY ./a.out

```

### **OpenMP / Fortran (test\_log\_omp\_f\_mod.f90)**

The following is a OpenMP Fortran module that calls the Fortran intrinsic math function, log, on the device (from inside OpenMP target regions):

```

MODULE test

    USE ISO_C_BINDING

CONTAINS

    SUBROUTINE log_real_sp (nelements, nrepetitions, initial_value, res) bind(C,NAME='log_real_sp')
        IMPLICIT NONE

```

```

INTEGER(KIND=C_INT), VALUE :: nelements, nrepetitions
REAL(C_FLOAT), VALUE :: initial_value
REAL(C_FLOAT) :: res(0:nelements-1), tmp
INTEGER :: i, j

!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO PRIVATE(tmp)
DO j = 0, nelements-1
    tmp = initial_value
    DO i = 0, nrepetitions-1
        tmp = tmp + log(tmp)
    END DO
    res(j) = tmp
END DO
RETURN
END SUBROUTINE log_real_sp

SUBROUTINE log_real_dp (nelements, nrepetitions, initial_value, res) bind(C,NAME='log_real_dp')
IMPLICIT NONE
INTEGER(KIND=C_INT), VALUE :: nelements, nrepetitions
REAL(C_DOUBLE), VALUE :: initial_value
REAL(C_DOUBLE) :: res(0:nelements-1), tmp
INTEGER :: i, j

!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO PRIVATE(tmp)
DO j = 0, nelements-1
    tmp = initial_value
    DO i = 0, nrepetitions-1
        tmp = tmp + log(tmp)
    END DO
    res(j) = tmp
END DO
RETURN
END SUBROUTINE log_real_dp

END MODULE test

```

Sample compilation command for `test_log_omp_f.f90`:

```
ifx -c -O2 -fopenmp -fopenmp-targets=spir64 test_log_omp_f_mod.f90
```

### SYCL (`test_log_sycl.cpp`)

In SYCL, you can control floating point semantics at the source-level by choosing which math function to call. For example, the SYCL program below calls the following three different versions of the log function:

- **`std::log`** : Refers to the log function in the C++ standard library. The particular implementation chosen will be according to what the compiler options (`-fp-model` and `-cl-fast-relaxed-math`) prescribe. For example, to get the implementation that uses native math instructions, you need to compile with the `-cl-fast-relaxed-math` option.
- **`sycl::log`** : Refers to the log function in the `sycl` namespace that is provided by SYCL. This function may use native instructions, even when the `-cl-fast-relaxed-math` option is not specified. Precision is spelled out by the SYCL specification.

- **`sycl::native::log`** : Refers to the native log function in the `sycl` namespace that is provided by SYCL. This function uses native math instructions, and the `-cl-fast-relaxed-math` option is not needed. Note that SYCL (and Intel GPUs) support native for single precision (`float`, `real`) only. Precision is spelled out by the SYCL specification.

```
#include <CL/sycl.hpp>
#include <iostream>
#include <assert.h>
#include <chrono>
#include <cmath>

#if FP_SIZE == 32
    typedef float FP_TYPE;
    static constexpr FP_TYPE VALIDATION_THRESHOLD = 1e-3;
#elif FP_SIZE == 64
    typedef double FP_TYPE;
    static constexpr FP_TYPE VALIDATION_THRESHOLD = 1e-6;
#endif

template<typename T>
void do_work_std (sycl::queue &q, unsigned NELEMENTS, unsigned NREPETITIONS, T initial_value, T *res)
{
    q.submit([&](sycl::handler& h) {
        h.parallel_for(NELEMENTS, [=] (auto j)
        {
            FP_TYPE tmp = initial_value;
            for (unsigned i = 0; i < NREPETITIONS; ++i)
                tmp += std::log(tmp);
            res[j] = tmp;
        });
    }).wait();
}

template<typename T>
void do_work_sycl (sycl::queue &q, unsigned NELEMENTS, unsigned NREPETITIONS, T initial_value, T *res)
{
    q.submit([&](sycl::handler& h) {
        h.parallel_for(NELEMENTS, [=] (auto j)
        {
            FP_TYPE tmp = initial_value;
            for (unsigned i = 0; i < NREPETITIONS; ++i)
                tmp += sycl::log(tmp);
            res[j] = tmp;
        });
    }).wait();
}

# if FP_SIZE == 32
template<typename T>
void do_work_sycl_native (sycl::queue &q, unsigned NELEMENTS, unsigned NREPETITIONS, T initial_value, T *res)
{
    q.submit([&](sycl::handler& h) {
        h.parallel_for(NELEMENTS, [=] (auto j)
        {
            FP_TYPE tmp = initial_value;
            for (unsigned i = 0; i < NREPETITIONS; ++i)
```

```
        tmp += sycl::native::log(tmp);
        res[j] = tmp;
    });
}).wait();
}
#endif

int main (int argc, char *argv[])
{
    static constexpr unsigned NELEMENTS = 64*1024*1024;
    static constexpr unsigned NREPETITIONS = 1024;

    sycl::device d (sycl::gpu_selector_v);
    sycl::queue q (d);

    q.submit([&](sycl::handler& h) {
        h.single_task ([=]()
        {
            }).wait();

        FP_TYPE initial_value = 2;
        FP_TYPE ref_res = initial_value;
        for (unsigned i = 0; i < NREPETITIONS; ++i)
            ref_res += std::log(ref_res);
        std::cout << "reference result = " << ref_res << std::endl;

        {
            FP_TYPE * std_res = new FP_TYPE[NELEMENTS];
            assert (std_res != nullptr);

            std::chrono::duration<float, std::micro> elapsed;

            {
                auto * res = sycl::malloc_device<FP_TYPE>(NELEMENTS, q);
                auto tbegin = std::chrono::system_clock::now();
                do_work_std<FP_TYPE>(q, NELEMENTS, NREPETITIONS, initial_value, res);
                auto tend = std::chrono::system_clock::now();
                elapsed = tend - tbegin;
                q.memcpy (std_res, res, NELEMENTS*sizeof(FP_TYPE)).wait();
                sycl::free (res, q);
            }
            std::cout << "std::log result[0] = " << std_res[0] << std::endl;

            bool allequal = true;
            for (auto i = 1; i < NELEMENTS; ++i)
                allequal = allequal and std_res[0] == std_res[i];
            if (allequal)
            {
                if (std::abs(ref_res - std_res[0])/std::abs(ref_res) <
std::abs(VALIDATION_THRESHOLD))
                    std::cout << "std::log validates. Total execution time is " << elapsed.count()
<< " us." << std::endl;
                else
                    std::cout << "std::log does not validate (ref=" << ref_res << " std_res=" <<
std_res[0] << " mix=" << std::abs(ref_res - std_res[0])/std::abs(ref_res) << ")" << std::endl;
            }
            else
                std::cout << "std::log does not validate, results are not equal." << std::endl;
        }
    });
}
```

```

        delete [] std_res;
    }

{
    FP_TYPE * sycl_res = new FP_TYPE[NELEMENTS];
    assert (sycl_res != nullptr);

    std::chrono::duration<float, std::micro> elapsed;

    {
        auto * res = sycl::malloc_device<FP_TYPE>(NELEMENTS, q);
        auto tbegin = std::chrono::system_clock::now();
        do_work_sycl<FP_TYPE>(q, NELEMENTS, NREPETITIONS, initial_value, res);
        auto tend = std::chrono::system_clock::now();
        elapsed = tend - tbegin;
        q.memcpy (sycl_res, res, NELEMENTS*sizeof(FP_TYPE)).wait();
        sycl::free (res, q);
    }
    std::cout << "sycl::log result[0] = " << sycl_res[0] << std::endl;

    bool allequal = true;
    for (auto i = 1; i < NELEMENTS; ++i)
        allequal = allequal and sycl_res[0] == sycl_res[i];
    if (allequal)
    {
        if (std::abs(ref_res - sycl_res[0])/std::abs(ref_res) <
std::abs(VALIDATION_THRESHOLD))
            std::cout << "sycl::log validates. Total execution time is " << elapsed.count()
<< " us." << std::endl;
        else
            std::cout << "sycl::log does not validate (ref=" << ref_res << " sycl_res=" <<
sycl_res[0] << " mix=" << std::abs(ref_res - sycl_res[0])/std::abs(ref_res) << ")" << std::endl;
    }
    else
        std::cout << "sycl::log does not validate, results are not equal." << std::endl;

    delete [] sycl_res;
}
#endif
# if FP_SIZE == 32
{
    FP_TYPE * sycl_native_res = new FP_TYPE[NELEMENTS];
    assert (sycl_native_res != nullptr);

    std::chrono::duration<float, std::micro> elapsed;

    {
        auto * res = sycl::malloc_device<FP_TYPE>(NELEMENTS, q);
        auto tbegin = std::chrono::system_clock::now();
        do_work_sycl_native<FP_TYPE>(q, NELEMENTS, NREPETITIONS, initial_value, res);
        auto tend = std::chrono::system_clock::now();
        elapsed = tend - tbegin;
        q.memcpy (sycl_native_res, res, NELEMENTS*sizeof(FP_TYPE)).wait();
        sycl::free (res, q);
    }
    std::cout << "sycl::native::log result[0] = " << sycl_native_res[0] << std::endl;

    bool allequal = true;
    for (auto i = 1; i < NELEMENTS; ++i)

```

```

        allequal = allequal and sycl_native_res[0] == sycl_native_res[i];
    if (allequal)
    {
        if (std::abs(ref_res - sycl_native_res[0])/std::abs(ref_res) <
std::abs(VALIDATION_THRESHOLD))
            std::cout << "sycl::native::log validates. Total execution time is " <<
elapsed.count() << " us." << std::endl;
        else
            std::cout << "sycl::native::log does not validate (ref=" << ref_res << "
sycl_native_res=" << sycl_native_res[0] << " mix=" << std::abs(ref_res - sycl_native_res[0])/
std::abs(ref_res) << ")" << std::endl;
    }
    else
        std::cout << "sycl::native::log does not validate, results are not equal." <<
std::endl;
    delete [] sycl_native_res;
}
#endif // FP_SIZE == 32

return 0;
}

```

Sample compilation and run commands for test\_log\_sycl.cpp:

```

icpx -fsycl -O2 test_log_sycl.cpp -DREAL_ELEMENT -DFP_SIZE=64 -fp-model=fast
OMP_TARGET_OFFLOAD=MANDATORY ./a.out

```

## Performance Experiments

We present performance results when running the different programs (OpenMP C++, OpenMP Fortran and SYCL) that call the log function. On the particular Intel® Data Center GPU Max Series used (1-stack only), the performance of the log function, in single-precision was as follows.

### Performance of log - Default Precision (Fast-math)

Version	Time (sec)
OpenMP/C++ (std::log)	93,118
OpenMP/Fortran (log)	94,342
SYCL (std::log)	31,835
SYCL (sycl::log)	31,644
SYCL (sycl::native::log)	31,684

### Performance of log - Fast-math

Version	Time (sec)
OpenMP/C++ (std::log)	93,181
OpenMP/Fortran (log)	94,467
SYCL (std::log)	31,657
SYCL (sycl::log)	32,064

Version	Time (sec)
SYCL ( <code>sycl::native::log</code> )	31,452

**Performance of log - Precise**

Version	Time (sec)
OpenMP/C++ ( <code>std::log</code> )	92,971
OpenMP/Fortran ( <code>log</code> )	94,444
SYCL ( <code>std::log</code> )	94,592
SYCL ( <code>sycl::log</code> )	94,852
SYCL ( <code>sycl::native::log</code> )	40,778

**Performance of log - Relaxed-math**

Version	Time (sec)
OpenMP/C++ ( <code>std::log</code> )	35,251
OpenMP/Fortran ( <code>log</code> )	35,787
SYCL ( <code>std::log</code> )	31,314
SYCL ( <code>sycl::log</code> )	32,077
SYCL ( <code>sycl::native::log</code> )	32,141

**Observations:**

- In OpenMP (C and Fortran): `std::log` follows what the compiler options (`-fp-model`, `-cl-relaxed-math`) prescribe. The `-cl-relaxed-math` option is needed to use native instructions for `std::log`.
- In SYCL: `sycl::log` may use native instructions, even with just `-fp-model=fast`.
- In SYCL: `sycl::native::log` will always use native machine instructions. The `-cl-relaxed-math` option is not needed.
- In OpenMP and SYCL: When the `-cl-relaxed-math` option is specified, native machine instructions will be used for the log function on the device.
- `-fp-model=precise` produces more accurate results, but the performance will be lower.

**References**

1. Consistency of Floating-Point Results using the Intel Compiler or Why doesn't my application always give the same answer?, by Dr. Martyn J. Corden and David Kreitzer (2018)
2. LLVM Language Reference Manual - Fast-Math Flags
3. Intel® oneAPI DPC++ Compiler documentation - User's Manual
4. Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference - Alphabetical Option List
5. Intel oneAPI DPC++/C++ Compiler Developer Guide and Reference - Xopenmp-target
6. OpenCL Developer Guide for Intel® Core and Intel® Xeon Processors
7. Intel® SDK for OpenCL Applications Developer Guide

- [8. Target Toolchain Options](#)
- [9. SYCL 2020 Specification - Math functions](#)
- [10. SYCL 2020 Specification - SYCL built-in functions for SYCL host and device](#)
- [11. The OpenCL Specification, Version 1.2, Khronos OpenCL Working Group](#)
- [12. OpenCL™Developer Guide for Intel® Processor Graphics](#)

## 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.

This section covers various topics related to OpenMP offloading, and how to improve the performance of offloaded code.

- [OpenMP Directives](#)
- [OpenMP Execution Model](#)
- [Terminology](#)
- [Compiling and Running an OpenMP Application](#)
- [Offloading oneMKL Computations onto the GPU](#)
- [Tools for Analyzing Performance of OpenMP Applications](#)
- [OpenMP Offload Best Practices](#)

### OpenMP Directives

Intel® compilers, `icx`, `icpx`, and `ifx` support various OpenMP 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 among the master threads of the teams, so each master thread executes a subset of the iterations.

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 `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.
- `tofrom`: 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 uninitialized 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 because they allow the compiler and runtime to decide how to best partition the iterations of an offloaded loop for execution on the GPU.

## OpenMP Execution Model

The OpenMP execution model has a single host device but multiple target devices. A device is a logical execution engine with its own local storage and data environment.

When executing on Intel® Data Center GPU Max Series, the entire GPU (which may have multiple stacks) can be considered as a device, or each stack 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 that encounters the `target` construct 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 or explicit) barriers, or other synchronization mechanisms.

## Terminology

In this chapter, OpenMP and SYCL terminology is used interchangeably to describe the partitioning of iterations of an offloaded parallel loop.

As described in the “SYCL 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 SYCL concepts map to OpenMP and CUDA concepts.

SYCL	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 ( <code>simdlen = 8, 16, 32</code> )	Warp (size = 32)
Maximum number of work-items per work-group	Thread limit	Maximum number of CUDA threads per thread block

## Compiling and Running an OpenMP Application

Use the following compiler options to enable OpenMP offload onto Intel® GPUs. These options apply to both C/C++ and Fortran.

```
-fopenmp -fopenmp-targets=spir64
```

By default the Intel® compiler converts the program into the intermediate language representation, SPIR-V, and stores that in the binary produced by the compilation process. The 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 is called Just-In-Time (JIT) compilation.

To enable the output of the compiler optimization report, add the following options:

```
-qopt-report=3 -O3
```

### Note:

- The `-qopenmp` compiler option is equivalent to `-fopenmp`, and the two options can be used interchangeably.

## Ahead-Of-Time (AOT) Compilation

For Ahead-Of-Time (AOT) compilation for Intel® Data Center GPU Max Series, you need to specify an additional compiler option (`-Xs`), as shown below. This option applies to both C/C++ and Fortran.

```
-fopenmp -fopenmp-targets=spir64_gen -Xs "-device pvc"
```

## OpenMP Runtime Routines

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](#)

## Environment Variables

Below are some environment variables that are useful for debugging or improving the performance of programs.

For additional information on environment variables, see:

- [Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference - Supported Environment Variables](#)
- [Intel® oneAPI Programming Guide - Debug Tools](#)
- [LLVM/OpenMP Runtimes](#)
- [Debugging Variables for Level Zero Plugin](#)

`LIBOMPTARGET_DEBUG=1`

Enables the display of debugging information from `libomptarget.so`.

`LIBOMPTARGET_DEVICES=<DeviceKind>`

Controls how sub-devices are exposed to users.

```
<DeviceKind> := DEVICE | SUBDEVICE | SUBSUBDEVICE |
               device | subdevice | subsubdevice
```

`DEVICE/device`: Only top-level devices are reported as OpenMP devices, and `subdevice` clause is supported.

`SUBDEVICE/subdevice`: Only 1st-level sub-devices are reported as OpenMP devices, and `subdevice` clause is ignored.

`SUBSUBDEVICE/subsubdevice`: Only second-level sub-devices are reported as OpenMP devices, and `subdevice` clause is ignored. On Intel® GPU using Level Zero backend, limiting the `subsubdevice` to a single compute slice within a stack also requires setting additional GPU compute runtime environment variable `CFESingleSliceDispatchCCSMode=1`.

The default is `<DeviceKind>=device`

`LIBOMPTARGET_INFO=<Num>`

Allows the user to request different types of runtime information from `libomptarget`. For details, see:

<https://openmp.llvm.org/design/Runtimes.html#libomptarget-info>

`LIBOMPTARGET_LEVEL0_MEMORY_POOL=<Option>`

Controls how reusable memory pool is configured.

```
<Option>      := 0 | <PoolInfoList>
<PoolInfoList> := <PoolInfo>[,<PoolInfoList>]
<PoolInfo>    := <MemType>[,<AllocMax>[,<Capacity>[,<PoolSize>]]]
<MemType>    := all | device | host | shared
<AllocMax>   := positive integer or empty, max allocation size in MB
<Capacity>   := positive integer or empty, number of allocations from
                  a single block
<PoolSize>   := positive integer or empty, max pool size in MB
```

`Pool` is a list of memory blocks that can serve at least `<Capacity>` allocations of up to `<AllocMax>` size from a single block, with total size not exceeding `<PoolSize>`.

`LIBOMPTARGET_LEVEL0_STAGING_BUFFER_SIZE=<Num>`

Sets the staging buffer size to `<Num>` KB. Staging buffer is used to optimize copy operation between host and device when host memory is not Unified Shared Memory (USM). The staging buffer is only used for discrete devices. The default staging buffer size is 16 KB.

`LIBOMPTARGET_LEVEL_ZERO_COMMAND_BATCH=copy`

Enables batching of commands for data transfer in a target region.

If there are `map(to: )` clauses on a `target` construct, then this environment variable allows multiple data transfers from the host to the device to occur concurrently. Similarly, if there are `map(from: )` clauses on the `target` construct, this environment variable allows multiple data transfers from the device to the host to occur concurrently. Note that `map(tofrom: )` or `map( )` would be split into `map(to: )` and `map(from: )`.

`LIBOMPTARGET_LEVEL_ZERO_USE_IMMEDIATE_COMMAND_LIST=<Bool>`

Enables/disables using immediate command list for kernel submission.

```
<Bool> := 1 | T | t | 0 | F | f
```

By default, using immediate command list is disabled.

`LIBOMPTARGET_PLUGIN=<Name>`

Designates the offload plugin name to use.

```
<Name> := LEVEL0 | OPENCL | X86_64 |
           level0 | opencl | x86_64
```

By default, the offload plugin is LEVEL0.

`LIBOMPTARGET_PLUGIN_PROFILE=<Enable>[,<Unit>]`

Enables basic plugin profiling and displays the result when the program finishes.

```
<Enable> := 1 | T
<Unit>   := usec | unit_usec
```

By default, plugin profiling is disabled.

if `<Unit>` is not specified, microsecond (`usec`) is the default unit

`LIBOMPTARGET_PROFILE=<FileName>`

Allows `libomptarget.so` to generate time profile output similar to Clang's `-ftime-trace` option.

`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.

#### Environment Variables to Control Implicit and Explicit Scaling

To disable implicit scaling and use one GPU stack only, set: `ZE_AFFINITY_MASK=0`

To enable explicit scaling, set: `LIBOMPTARGET_DEVICES=subdevice`

On Intel® Data Center GPU Max Series, implicit scaling is on by default.

#### Environment Variables for SYCL

There are several `SYCL_PI_LEVEL_ZERO` environment variables that are useful for the development and debugging of SYCL programs (not just OpenMP). They are documented at:

<https://github.com/intel/llvm/blob/sycl/sycl/doc/EnvironmentVariables.md>

## References

1. OpenMP Features and Extensions Supported in Intel® oneAPI DPC++/C++ Compiler
2. Fortran Language and OpenMP Features Implemented in Intel® Fortran Compiler
3. Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference - Supported Environment Variables
4. Intel® oneAPI Programming Guide - Debug Tools
5. LLVM/OpenMP Runtimes
6. Debugging Variables for Level Zero Plugin
7. Environment variables that effect DPC++ compiler and runtime

## Offloading oneMKL Computations onto the GPU

The Intel® oneAPI Math Kernel Library (oneMKL) improves performance with math routines for software applications that solve large computational problems. oneMKL provides BLAS, Sparse BLAS, and LAPACK linear algebra routines, fast Fourier transforms, vectorized math functions, random number generation functions, and other functionality.

The oneMKL distribution includes an examples directory which contains examples of various calls to oneMKL routines.

For more information about the Intel® oneAPI Math Kernel Library, see:

- [Developer Reference for Intel® oneAPI Math Kernel Library - C](#)
- [Developer Reference for Intel® oneAPI Math Kernel Library - Fortran](#)
- [Introducing Batch GEMM Operations](#)

## Compile and Link Commands when Using oneMKL OpenMP Offload

The information given in this section is specific to Linux. For information specific to Windows, and for more details, refer to the [Intel® oneAPI Math Kernel Library Link Line Advisor](#).

### Notes:

- The link commands shown below will dynamically link to the oneMKL library.
- The Intel® oneMKL LP64 libraries index arrays with the 32-bit integer type; whereas the Intel® oneMKL ILP64 libraries use the 64-bit integer type (necessary for indexing large arrays, with more than  $2^{31} - 1$  elements).

### C/C++ (Linux)

The compile and link commands for a C/C++ program that uses OpenMP threading and calls oneMKL C/C++ API with **32-bit integers** are as follows.

```
Compile:  
icpx -fiopenmp -fopenmp-targets=spir64 -qmkl -c source.cpp
```

```
Link:  
icpx -fiopenmp -fopenmp-targets=spir64 -qmkl -lOpenCL source.o
```

If the program calls oneMKL C/C++ API with **64-bit integers**, the compile and link commands are:

```
Compile:  
icpx -fiopenmp -fopenmp-targets=spir64 -qmkl -DMKL_ILP64 -c source.cpp
```

```
Link:  
icpx -fiopenmp -fopenmp-targets=spir64 -qmkl -lOpenCL source.o
```

### Fortran (Linux)

The compile and link commands for a Fortran program that uses OpenMP threading and calls oneMKL Fortran API with **32-bit integers** are as follows.

```
Compile:  
ifx -fiopenmp -fopenmp-targets=spir64 -qmkl -fpp -free -c source.f
```

```
Link:  
ifx -fiopenmp -fopenmp-targets=spir64 -qmkl -lOpenCL source.o
```

If the program calls oneMKL Fortran API with **64-bit integers**, the compile and link commands are:

```
Compile:  
ifx -fiopenmp -fopenmp-targets=spir64 -qmkl -i8 -fpp -free -c source.f
```

```
Link:  
ifx -fiopenmp -fopenmp-targets=spir64 -qmkl -lOpenCL source.o
```

#### A Note on the `-qmkl` Compiler Option

Use the `-qmkl` option (equivalent to `-qmkl=parallel`) to link with a certain Intel® oneAPI Math Kernel Library threading layer depending on the threading option provided:

- For `-fiopenmp`, the OpenMP threading layer for Intel® Compilers
- For `-tbb`, the Intel® Threading Building Blocks (Intel® TBB) threading layer

Use `-qmkl=sequential` to link with the sequential version of Intel® oneAPI Math Kernel Library.

Note that `-qmkl=parallel/sequential` affects threading on the CPU only. Offloaded MKL computations will always be parallelized as appropriate, and will occupy as many Vector Engines on the GPU as possible.

## OpenMP Directives to Offload oneMKL Computations

You can use OpenMP directives to offload oneMKL computations onto the GPU. There are two ways to do this.

One way involves using the Intel-specific OpenMP extension target variant dispatch directive. You would place the call to the oneMKL routine inside a target variant dispatch construct, as shown in the example below. In this example, matrices `a`, `b`, and `c` used in the multiplication are mapped to the device before the call to the oneMKL routine `cblas_dgemm`. The `use_device_ptr(A,B,C)` clause is used on the target variant dispatch directive to indicate that `a`, `b`, and `c` point to objects that have corresponding storage on the device. When `cblas_dgemm` is called, the corresponding device pointers for `a`, `b`, and `c` will be passed as arguments, and the device copies of `a`, `b`, and `c` will be used in the computation.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include "mkl.h"
#include "mkl_omp_offload.h"

#define min(x,y) (((x) < (y)) ? (x) : (y))
#define EPSILON 0.0001

int main()
{
    double *A, *B, *C, *C_f1;
    int64_t m, n, k;
    double alpha, beta;
    double sum;
    int64_t i, j, q;
    int fail;
```

```
printf ("\n This example computes real matrix C=alpha*A*B+beta*C using \n"
       " Intel oneMKL function dgemm, where A, B, and C are matrices and \n"
       " alpha and beta are double precision scalars\n\n");

m = 2000, k = 200, n = 1000;
printf (" Initializing data for matrix multiplication C=A*B for matrix \n"
       " A(%li x %li) and matrix B(%li x %li)\n\n", m, k, k, n);
alpha = 1.0; beta = 0.0;

printf (" Allocating memory for matrices aligned on 64-byte boundary for better \n"
       " performance \n\n");
A = (double *)mkl_malloc( m * k * sizeof( double ), 64 );
B = (double *)mkl_malloc( k * n * sizeof( double ), 64 );
C = (double *)mkl_malloc( m * n * sizeof( double ), 64 );
C_f1 = (double *)mkl_malloc( m*n*sizeof( double ), 64 );

if (A == NULL || B == NULL || C == NULL || C_f1 == NULL) {
    printf( "\n ERROR: Cannot allocate memory for matrices. Exiting... \n\n");
    return 1;
}

printf (" Intializing matrices \n\n");
for (i = 0; i < (m*k); i++) {
    A[i] = (double)(i+1);
}

for (i = 0; i < (k*n); i++) {
    B[i] = (double)(-i-1);
}

for (i = 0; i < (m*n); i++) {
    C[i] = 0.0;
    C_f1[i] = 0.0;
}

printf (" Computing matrix product using Intel oneMKL dgemm function via CBLAS interface \n
\n");

#pragma omp target data map(to: A[0:m*k], B[0:k*n]) map(tofrom: C[0:m*n])
{
    #pragma omp target variant dispatch use_device_ptr(A, B, C)
    {
        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                    m, n, k, alpha, A, k, B, n, beta, C, n);
    }
}

printf ("\n Top left corner of matrix C: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C[j+i*n]);
    }
    printf ("\n");
}

printf (" Computing matrix product using for-loops \n");
```

```

for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (q = 0; q < k; q++) {
            sum += A[k*i+q] * B[n*q+j];
        }
        C_fl[n*i+j] = alpha * sum + beta * C_fl[n*i+j];
    }
}

printf ("\n Top left corner of matrix C_fl: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C_fl[j+i*n]);
    }
    printf ("\n");
}

printf (" Computations completed. Verifying... \n\n");

fail = 0;
for (i = 0; i < (m*n); i++) {
    if (fabs(C[i] - C_fl[i]) > EPSILON) {
        fail = 1;
        break;
    }
}

if (fail)
    printf ("\n **** FAIL **** \n");
else
    printf ("\n **** PASS **** \n");

printf ("\n Deallocating memory \n\n");
mkl_free(A);
mkl_free(B);
mkl_free(C);

return fail;
}

```

Another way to inform the compiler that oneMKL computations should be offloaded onto the GPU is by using the OpenMP 5.1 `dispatch` directive, as shown in the example below. In this example too, matrices `a`, `b`, and `c` are mapped to the device before the call to the oneMKL routine `cblas_dgemm`. When `cblas_dgemm` is called, the corresponding device pointers for `a`, `b`, and `c` will be passed as arguments, so the device copies of `a`, `b`, and `c` will be used in the computation.

The `use_device_ptr` clause is not needed on the `dispatch` directive. With OpenMP 5.1, the list of device pointers needed by the oneMKL routines is given in the oneMKL OpenMP offload header file, `mkl_omp_offload.h`, where the GPU variant function is declared. The user should carefully review the list of device pointers required in the oneMKL header file and make sure that the corresponding matrices are accessible from the device before calling the oneMKL routine.

Note that, depending on the version of the compiler you are using, you may need to add the compiler option `-fopenmp-version=51` in order for the `dispatch` directive to be accepted.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include "mkl.h"
#include "mkl_omp_offload.h"

#define min(x,y) (((x) < (y)) ? (x) : (y))
#define EPSILON 0.0001

int main()
{
    double *A, *B, *C, *C_f1;
    int64_t m, n, k;
    double alpha, beta;
    double sum;
    int64_t i, j, q;
    int fail;

    printf ("\n This example computes real matrix C=alpha*A*B+beta*C using \n"
           " Intel oneMKL function dgemm, where A, B, and C are matrices and \n"
           " alpha and beta are double precision scalars\n\n");

    m = 2000, k = 200, n = 1000;
    printf (" Initializing data for matrix multiplication C=A*B for matrix \n"
           " A(%li x %li) and matrix B(%li x %li)\n\n", m, k, k, n);
    alpha = 1.0; beta = 0.0;

    printf (" Allocating memory for matrices aligned on 64-byte boundary for better \n"
           " performance \n\n");
    A = (double *)mkl_malloc( m * k * sizeof( double ), 64 );
    B = (double *)mkl_malloc( k * n * sizeof( double ), 64 );
    C = (double *)mkl_malloc( m * n * sizeof( double ), 64 );
    C_f1 = (double *)mkl_malloc( m*n*sizeof( double ), 64 );

    if (A == NULL || B == NULL || C == NULL || C_f1 == NULL) {
        printf( "\n ERROR: Cannot allocate memory for matrices. Exiting... \n\n");
        return 1;
    }

    printf (" Intializing matrices \n\n");
    for (i = 0; i < (m*k); i++) {
        A[i] = (double)(i+1);
    }

    for (i = 0; i < (k*n); i++) {
        B[i] = (double)(-i-1);
    }

    for (i = 0; i < (m*n); i++) {
        C[i] = 0.0;
        C_f1[i] = 0.0;
    }

    printf (" Computing matrix product using Intel oneMKL dgemm function via CBLAS interface \n"
           "\n");
```

```
#pragma omp target data map(to: A[0:m*k], B[0:k*n]) map(tofrom: C[0:m*n])
{
    #pragma omp target variant dispatch use_device_ptr(A, B, C)
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                m, n, k, alpha, A, k, B, n, beta, C, n);
}

printf ("\n Top left corner of matrix C: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C[j+i*n]);
    }
    printf ("\n");
}

printf (" Computing matrix product using for-loops \n");

for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (q = 0; q < k; q++) {
            sum += A[k*i+q] * B[n*q+j];
        }
        C_fl[n*i+j] = alpha * sum + beta * C_fl[n*i+j];
    }
}

printf ("\n Top left corner of matrix C_fl: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C_fl[j+i*n]);
    }
    printf ("\n");
}

printf (" Computations completed. Verifying... \n\n");

fail = 0;
for (i = 0; i < (m*n); i++) {
    if (fabs(C[i] - C_fl[i]) > EPSILON) {
        fail = 1;
        break;
    }
}

if (fail)
    printf ("\n **** FAIL **** \n");
else
    printf ("\n **** PASS **** \n");

printf ("\n Deallocating memory \n\n");
mkl_free(A);
mkl_free(B);
mkl_free(C);

return fail;
}
```

**Notes:**

- oneMKL routines expect the arrays/matrices to be on the device before the computation will be run. So the user has to map the data to the device, or allocate the data directly on the device, before calling a oneMKL routine.
- If a oneMKL routine is not called from a target variant dispatch (or dispatch) region, or if offload is disabled, then the oneMKL computations will be executed on the CPU.
- Only one call to a oneMKL routine can be issued from an OpenMP target variant dispatch (or dispatch) construct. If there are two consecutive calls to oneMKL routines, then the calls should be placed in separate target variant dispatch (or dispatch) constructs.

**Fortran**

When calling oneMKL routines from Fortran code, be sure to add the following include statement:

```
include "mkl_omp_offload.f90"
```

Also, if calling oneMKL Fortran API with **32-bit integers**, add the following module use statement:

```
use onemkl_blas_omp_offload_lp64
```

On the other hand, if calling oneMKL Fortran API with **64-bit integers**, add the following module use statement:

```
use onemkl_blas_omp_offload_ilp64
```

The following Fortran example illustrates how DGEMM is called from a Fortran program, and the `include` and `use` statements mentioned above.

```
include "mkl_omp_offload.f90"

program DGEMM_MAIN

#if defined(MKL_ILP64)
    use onemkl_blas_omp_offload_ilp64
#else
    use onemkl_blas_omp_offload_lp64
#endif
    use omp_lib
    use iso_fortran_env
    implicit none

    integer, parameter :: m = 20
    integer, parameter :: k = 5
    integer, parameter :: n = 10
    double precision    a(m,k), b(k,n), c1(m,n), c2(m,n)
    double precision    alpha, beta
    integer             i, j

    print*
    print*, '  D G E M M  EXAMPLE PROGRAM'

    ! Initialize

    alpha = 1.025
    beta  = 0.75

    do i = 1, m
        do j = 1, k
            a(i,j) = (i-1) - (0.25 * k)
```

```

    end do
end do

do i = 1, k
  do j = 1, n
    b(i,j) = -((i-1) + j)
  end do
end do

do i = 1, m
  do j = 1, n
    c1(i,j) = 0.2 + i - j
    c2(i,j) = 0.2 + i - j
  end do
end do

! Execute DGEMM on host.

call DGEMM('N','N',m,n,k,alpha,a,m,b,k,beta,c1,m)

print *
print *, 'c1 - After DGEMM host execution'

do i=1,m
  print 110, (c1(i,j),j=1,n)
end do
print*

! Execute DGEMM on device

 !$omp target data map(to: a, b) map(tofrom: c2)

 !$omp dispatch
   call DGEMM('N','N',m,n,k,alpha,a,m,b,k,beta,c2,m)

 !$omp end target data

print *
print *, 'c2 - After DGEMM device execution'

do i=1,m
  print 110, (c2(i,j),j=1,n)
end do
print *

110  format(7x,10(f10.2,2x))

end

```

To compile and link the above Fortran example with **32-bit integers**:

```

ifx -fiopenmp -fopenmp-targets=spir64 -qmkl -fpp -free -c dgemm_dispatch_f.f90
ifx -fiopenmp -fopenmp-targets=spir64 -qmkl -fsycl -L${MKLROOT}/lib/intel64 -liomp5 -lsycl -
lOpenCL -lstdc++ -lpthread -lm -ldl -lmkl_sycl dgemm_dispatch_f.o

```

To compile and link the above Fortran example with **64-bit integers**:

```
ifx -fopenmp -fopenmp-targets=spir64 -qmkl -m64 -DMKL_ILP64 -i8 -fpp -free -c
dgemm_dispatch_f.f90
ifx -fopenmp -fopenmp-targets=spir64 -qmkl -fsycl -L${MKLROOT}/lib/intel64 -liomp5 -lsycl -
lOpenCL -lstdc++ -lpthread -lm -ldl -lmkl_sycl dgemm_dispatch_f.o
```

After generating the executable (a.out), from a C/C++ or Fortran program, you can run the executable under [unitrace](#) and look for the heading “Device Timing Results” in the generated trace. Below that heading we should see the oneMKL kernels listed. This way we confirm that oneMKL computations have been offloaded onto the GPU.

Example run command:

```
OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0 unitrace -h -d ./a.out
```

## Batching of oneMKL GEMM Calls

The oneMKL library includes “batch” routines that allow the user to batch several oneMKL calls into a single oneMKL call. At runtime, oneMKL will intelligently execute all of the matrix operations to optimize overall performance.

For example, the `blas_dgemm` routine computes a matrix-matrix product of two general matrices `a` and `b`, returning the result in a matrix `c`. The `blas_dgemm` interface is shown below.

```
void blas_dgemm (const CBLAS_LAYOUT layout,
const CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb,
const MKL_INT m, const MKL_INT n, const MKL_INT k,
const double alpha, const double *a,
const MKL_INT lda, const double *b,
const MKL_INT ldb, const double beta,
double *c, const MKL_INT ldc);
```

The `blas_dgemm_batch` routine is similar to the `blas_dgemm` routine, but the `blas_dgemm_batch` routine performs matrix-matrix operations on groups of matrices, processing a number of groups at once.

The `blas_dgemm_batch` interface is shown below. Note that the interface resembles the `blas_dgemm` interface. However, it involves passing matrix arguments as arrays of pointers to matrices, and passing parameters as arrays of parameters.

```
void blas_dgemm_batch (const CBLAS_LAYOUT layout,
const CBLAS_TRANSPOSE* transa_array, const CBLAS_TRANSPOSE* transb_array,
const MKL_INT* m_array, const MKL_INT* n_array, const MKL_INT* k_array,
const double* alpha_array, const double **a_array,
const MKL_INT* lda_array, const double **b_array,
const MKL_INT* ldb_array, const double* beta_array,
double **c_array, const MKL_INT* ldc_array,
const MKL_INT group_count, const MKL_INT* group_size);
```

The batch operation is defined as follows:

```
idx = 0
for i = 0 .. group_count - 1
    alpha and beta in alpha_array[i] and beta_array[i]
    for j = 0 .. group_size[i] - 1
        a, b, and c matrices in a_array[idx], b_array[idx], and c_array[idx], respectively
        c := alpha*op(a)*op(b) + beta*c,
        idx = idx + 1
    end for
end for
```

where:

- $\text{op}(X)$  is one of  $\text{op}(X) = X$ , or  $\text{op}(X) = X^T$ , or  $\text{op}(X) = XH$ ,
- alpha and beta are scalar elements of `alpha_array` and `beta_array`,
- a, b and c are matrices such that for m, n, and k which are elements of `m_array`, `n_array`, and `k_array`:
- $\text{op}(a)$  is an m-by-k matrix,
- $\text{op}(b)$  is a k-by-n matrix,
- C is an m-by-n matrix.
- a, b, and c represent matrices stored at addresses pointed to by `a_array`, `b_array`, and `c_array`, respectively. The number of entries in `a_array`, `b_array`, and `c_array`, or `total_batch_count`, is equal to the sum of all of the `group_size` entries.

It is possible to batch the multiplications of different shapes and parameters by packaging them into groups, where each group consists of multiplications of matrices of the same shapes (same m, n, and k) and the same parameters.

The basic assumption for the batch API are that all operations in a batch (whether in the same group or different groups) are independent of one another. So oneMKL does not guarantee any particular ordering between operations in a batch, and will try to execute multiple operations in parallel.

In general, the larger you can make the batch size, the better. This allows oneMKL to better parallelize the operations and distribute the work across the GPU.

We illustrate how two calls to `blas_dgemm` can be replaced with one call to `blas_dgemm_batch`. The following example includes two calls to `blas_dgemm`.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include "mkl.h"
#include "mkl_omp_offload.h"

#define min(x,y) (((x) < (y)) ? (x) : (y))
#define epsilon 0.0000001f

bool compare(double x, double y)
{
    // returns true if x and y are the same
    return fabs(x - y) <= epsilon;
}

int main()
{
    double *A1, *B1, *C1, *C1_fl;
    double *A2, *B2, *C2, *C2_fl;
    int m, n, k, i, j, q;
    double alpha, beta;
    double sum;
    int fail;
    double t_start, t_end;

    m = 2000, k = 200, n = 1000;
    alpha = 1.0; beta = 0.0;

    printf (" Allocating memory for matrices aligned on 64-byte boundary for better \n"
           " performance \n\n");
    A1 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
    B1 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
    C1 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
    C1_fl = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
```

```

A2 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
B2 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
C2 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
C2_fl = (double *)mkl_malloc (m*n*sizeof( double ), 64 );

if (A1 == NULL || B1 == NULL || C1 == NULL || C1_fl == NULL ||
    A2 == NULL || B2 == NULL || C2 == NULL || C2_fl == NULL) {
    printf( "\n ERROR: Can't allocate memory for matrices. Aborting... \n\n");
    return 1;
}

printf (" Intializing matrix data \n\n");
for (i = 0; i < (m*k); i++) {
    A1[i] = A2[i] = (double)(i+1);
}

for (i = 0; i < (k*n); i++) {
    B1[i] = B2[i] = (double)(-i-1);
}

for (i = 0; i < (m*n); i++) {
    C1[i] = C2[i] = 0.0;
    C1_fl[i] = C2_fl[i] = 0.0;
}

printf (" \nComputing matrix product using Intel MKL cblas_dgemm function \n");

t_start = omp_get_wtime();

#pragma omp target data
map(to: A1[0:m*k], B1[0:k*n], A2[0:m*k], B2[0:k*n]) \
map(tofrom: C1[0:m*n], C2[0:m*n])
{
    #pragma omp dispatch
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                m, n, k, alpha, A1, k, B1, n, beta, C1, n);

    #pragma omp dispatch
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                m, n, k, alpha, A2, k, B2, n, beta, C2, n);
}

t_end = omp_get_wtime();

printf ("\n Top left corner of matrix C1: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C1[j+i*n]);
    }
    printf ("\n");
}

printf ("\n Top left corner of matrix C2: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C2[j+i*n]);
    }
}

```

```
    printf ("\n");
}

printf (" \nComputing matrix product using for-loops \n");

for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (q = 0; q < k; q++)
            sum += A1[k*i+q] * B1[n*q+j];
        C1_fl[n*i+j] = sum;
    }
}

for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (q = 0; q < k; q++)
            sum += A2[k*i+q] * B2[n*q+j];
        C2_fl[n*i+j] = sum;
    }
}

printf ("\n Top left corner of matrix C1: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C1_fl[j+i*n]);
    }
    printf ("\n");
}

printf ("\n Top left corner of matrix C2: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C2_fl[j+i*n]);
    }
    printf ("\n");
}

printf ("\n Computations completed. Verifying... \n\n");

fail = 0;
for (i = 0; i < (m*n); i++) {
    if (! compare(C1[i], C1_fl[i]) || ! compare(C2[i], C2_fl[i])) {
        fail = 1;
        break;
    }
}

if (fail) {
    printf (" **** FAIL **** \n");
}
else {
    printf(" time = %lf seconds\n", t_end - t_start);
    printf (" **** PASS **** \n");
}

mkl_free(A1);
```

```

mkl_free(B1);
mkl_free(C1);
mkl_free(C1_f1);
mkl_free(A2);
mkl_free(B2);
mkl_free(C2);
mkl_free(C2_f1);

    return 0;
}

```

The two calls to `cblas_dgemm` in the above example can be batched together, resulting in one call to `cblas_dgemm_batch`, as shown in the following example. Note that the batch is composed of one group of size 2, since we have two matrix multiplications with the same set of parameters (`layout`, `transa`, `transb`, `m`, `n`, `k`, `alpha`, `lda`, `ldb`, `beta`, and `ldc`). `total_batch_size` in this case is 2.

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include "mkl.h"
#include "mkl_omp_offload.h"

#define min(x,y) ((x) < (y)) ? (x) : (y)
#define epsilon 0.0000001f

bool compare(double x, double y)
{
    // return true if x and y are the same
    return (fabs(x - y) <= epsilon);
}

int main()
{
    double *A1, *B1, *C1, *C1_f1;
    double *A2, *B2, *C2, *C2_f1;
    int m, n, k, i, j, q;
    double alpha, beta;
    double sum;
    int fail;
    double t_start, t_end;

    m = 2000, k = 200, n = 1000;
    alpha = 1.0; beta = 0.0;

    printf (" Allocating memory for matrices aligned on 64-byte boundary for better \n"
           " performance \n\n");
    A1 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
    B1 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
    C1 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
    C1_f1 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );

    A2 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
    B2 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
    C2 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
    C2_f1 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );

    if (A1 == NULL || B1 == NULL || C1 == NULL || C1_f1 == NULL ||

```

```
A2 == NULL || B2 == NULL || C2 == NULL || C2_f1 == NULL) {  
    printf( "\n ERROR: Can't allocate memory for matrices. Aborting... \n\n");  
    return 1;  
}  
  
printf (" Intializing matrix data \n\n");  
for (i = 0; i < (m*k); i++) {  
    A1[i] = A2[i] = (double)(i+1);  
}  
  
for (i = 0; i < (k*n); i++) {  
    B1[i] = B2[i] = (double)(-i-1);  
}  
  
for (i = 0; i < (m*n); i++) {  
    C1[i] = C2[i] = 0.0;  
    C1_f1[i] = C2_f1[i] = 0.0;  
}  
  
printf (" \nComputing matrix product using Intel MKL cblas_dgemm_batch function \n");  
  
#define GRP_COUNT 1 // 1 group  
  
MKL_INT group_count = GRP_COUNT;  
MKL_INT group_sizes[GRP_COUNT] = {2}; // 8 matrix multiplications  
  
CBLAS_TRANSPOSE transa_array[GRP_COUNT] = {CblasNoTrans};  
CBLAS_TRANSPOSE transb_array[GRP_COUNT] = {CblasNoTrans};  
  
MKL_INT m_array[GRP_COUNT] = {m};  
MKL_INT n_array[GRP_COUNT] = {n};  
MKL_INT k_array[GRP_COUNT] = {k};  
  
MKL_INT lda_array[GRP_COUNT] = {k};  
MKL_INT ldb_array[GRP_COUNT] = {n};  
MKL_INT ldc_array[GRP_COUNT] = {n};  
  
double alpha_array[GRP_COUNT] = {alpha};  
double beta_array[GRP_COUNT] = {beta};  
  
// Number of matrix multiplications = 2  
double **a_array, **b_array, **c_array;  
a_array = (double **)mkl_calloc(2, sizeof( double* ), 64);  
b_array = (double **)mkl_calloc(2, sizeof( double* ), 64);  
c_array = (double **)mkl_calloc(2, sizeof( double* ), 64);  
  
t_start = omp_get_wtime();  
  
// Call cblas_dgemm_batch  
#pragma omp target enter data \  
map(to: A1[0:m*k], B1[0:k*n], C1[0:m*n]) \  
map(to: A2[0:m*k], B2[0:k*n], C2[0:m*n])  
  
#pragma omp target data use_device_ptr(A1, B1, C1, A2, B2, C2)  
{  
    a_array[0] = A1, a_array[1] = A2;  
    b_array[0] = B1, b_array[1] = B2;  
    c_array[0] = C1, c_array[1] = C2;
```

```
}

#pragma omp target data \
map(to:a_array[0:2], b_array[0:2], c_array[0:2])
{
    #pragma omp dispatch
    cblas_dgemm_batch (
        CblasRowMajor,
        transa_array,
        transb_array,
        m_array,
        n_array,
        k_array,
        alpha_array,
        (const double **)a_array,
        lda_array,
        (const double **)b_array,
        ldb_array,
        beta_array,
        c_array,
        ldc_array,
        group_count,
        group_sizes);
} // end target data map

#pragma omp target exit data \
map(from: C1[0:m*n], C2[0:m*n])

t_end = omp_get_wtime();

printf ("\n Top left corner of matrix C1: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C1[j+i*n]);
    }
    printf ("\n");
}

printf ("\n Top left corner of matrix C2: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C2[j+i*n]);
    }
    printf ("\n");
}

printf (" \nComputing matrix product using for-loops \n");

for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (q = 0; q < k; q++)
            sum += A1[k*i+q] * B1[n*q+j];
        C1_fl[n*i+j] = sum;
    }
}

for (i = 0; i < m; i++) {
```

```

        for (j = 0; j < n; j++) {
            sum = 0.0;
            for (q = 0; q < k; q++)
                sum += A2[k*i+q] * B2[n*q+j];
            C2_f1[n*i+j] = sum;
        }
    }

printf ("\n Top left corner of matrix C1: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C1_f1[j+i*n]);
    }
    printf ("\n");
}

printf ("\n Top left corner of matrix C2: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C2_f1[j+i*n]);
    }
    printf ("\n");
}

printf ("\n Computations completed. Verifying... \n\n");

fail = 0;
for (i = 0; i < (m*n); i++) {
    if (! compare(C1[i], C1_f1[i]) || ! compare(C2[i], C2_f1[i])) {
        fail = 1;
        break;
    }
}

if (fail) {
    printf (" **** FAIL **** \n");
}
else {
    printf(" time = %lf seconds\n", t_end - t_start);
    printf (" **** PASS **** \n");
}

mkl_free(A1);
mkl_free(B1);
mkl_free(C1);
mkl_free(C1_f1);
mkl_free(A2);
mkl_free(B2);
mkl_free(C2);
mkl_free(C2_f1);

return 0;
}

```

The performance of the above two examples when running on the particular GPU used (1-stack only) was as follows:

```

dgemm_example_01_c.cpp (two calls to cblas_dgemm): 2.976183 seconds
dgemm_batch_example_01_c.cpp (one call to cblas_dgemm_batch): 1.881641 seconds

```

A more complex example of batching is shown below. In this example, we have a batch composed of 3 groups (GROUP\_COUNT=3). The size of each group is a randomly chosen number between 1 and 10. Several parameters (layout, transA, transB, m, n, and k) are chosen randomly, but in each group the parameters are the same for all the multiplications. The total\_batch\_size is equal to the sum of all the group sizes.

```
#include <stdio.h>
#include <omp.h>
#include "mkl.h"
#include "mkl_omp_offload.h"
#include "common.h"

#define GROUP_COUNT 3

int dnum = 0;

int main() {

    CBLAS_LAYOUT layout = (rand_int(0,1) == 0) ? CblasColMajor : CblasRowMajor;
    CBLAS_TRANSPOSE *transA, *transB;
    MKL_INT *m, *n, *k, *lda, *ldb, *ldc;
    double *alpha, *beta;
    MKL_INT *group_size, *sizea_array, *sizeb_array, *sizec_array, total_batch_size = 0, sizea,
    sizeb, sizec;
    double **a_array, **b_array, **c_array, **c_ref_array;
    double **a_array_dev, **b_array_dev, **c_array_dev;

    transA = (CBLAS_TRANSPOSE *)mkl_malloc(GROUP_COUNT * sizeof(CBLAS_TRANSPOSE), 64);
    transB = (CBLAS_TRANSPOSE *)mkl_malloc(GROUP_COUNT * sizeof(CBLAS_TRANSPOSE), 64);

    m = (MKL_INT *)mkl_malloc(GROUP_COUNT * sizeof(MKL_INT), 64);
    n = (MKL_INT *)mkl_malloc(GROUP_COUNT * sizeof(MKL_INT), 64);
    k = (MKL_INT *)mkl_malloc(GROUP_COUNT * sizeof(MKL_INT), 64);
    lda = (MKL_INT *)mkl_malloc(GROUP_COUNT * sizeof(MKL_INT), 64);
    ldb = (MKL_INT *)mkl_malloc(GROUP_COUNT * sizeof(MKL_INT), 64);
    ldc = (MKL_INT *)mkl_malloc(GROUP_COUNT * sizeof(MKL_INT), 64);
    group_size = (MKL_INT *)mkl_malloc(GROUP_COUNT * sizeof(MKL_INT), 64);
    alpha = (double *)mkl_malloc(GROUP_COUNT * sizeof(double), 64);
    beta = (double *)mkl_malloc(GROUP_COUNT * sizeof(double), 64);

    if ((m == NULL) || (n == NULL) || (k == NULL) || (lda == NULL) || (ldb == NULL) || (ldc ==
    NULL) ||
        (group_size == NULL) || (alpha == NULL) || (beta == NULL)) {
        printf("Cannot allocate input arrays\n");
        return 1;
    }

    MKL_INT i, j, p, idx;

    for (i = 0; i < GROUP_COUNT; i++) {
        transA[i] = (rand_int(0,1) == 0) ? CblasNoTrans : CblasTrans;
        transB[i] = (rand_int(0,1) == 0) ? CblasNoTrans : CblasTrans;
        alpha[i] = rand_double_scalar();
        beta[i] = rand_double_scalar();
        m[i] = rand_int(1, 20);
        n[i] = rand_int(1, 20);
        k[i] = rand_int(1, 20);
        lda[i] = MAX(m[i], k[i]);
        ldb[i] = MAX(k[i], n[i]);
        total_batch_size += m[i];
    }
}
```

```

        ldc[i] = MAX(m[i], n[i]);
        group_size[i] = rand_int(1, 10);
        total_batch_size += group_size[i];
#endif MKL_ILP64
        printf("Group %lld: layout = %s, transA = %s, transB = %s, m = %lld, n = %lld, k = %lld,
lda = %lld, ldb = %lld, ldc = %lld, alpha = %lf, beta = %lf, group_size = %lld\n",
i, (layout == CblasColMajor) ? "Column Major" : "Row Major",
(transA[i] == CblasNoTrans) ? "Non Transpose" : "Transpose",
(transB[i] == CblasNoTrans) ? "Non Transpose" : "Transpose",
m[i], n[i], k[i], lda[i], ldb[i], ldc[i], alpha[i], beta[i], group_size[i]);
#else
        printf("Group %d: layout = %s, transA = %s, transB = %s, m = %d, n = %d, k = %d, lda =
%d, ldb = %d, ldc = %d, alpha = %lf, beta = %lf, group_size = %d\n",
i, (layout == CblasColMajor) ? "Column Major" : "Row Major",
(transA[i] == CblasNoTrans) ? "Non Transpose" : "Transpose",
(transB[i] == CblasNoTrans) ? "Non Transpose" : "Transpose",
m[i], n[i], k[i], lda[i], ldb[i], ldc[i], alpha[i], beta[i], group_size[i]);
#endif
}

sizea_array = (MKL_INT *)mkl_malloc(sizeof(MKL_INT) * total_batch_size, 64);
sizeb_array = (MKL_INT *)mkl_malloc(sizeof(MKL_INT) * total_batch_size, 64);
sizec_array = (MKL_INT *)mkl_malloc(sizeof(MKL_INT) * total_batch_size, 64);

a_array = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);
b_array = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);
c_array = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);
a_array_dev = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);
b_array_dev = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);
c_array_dev = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);
c_ref_array = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);

if ((sizea_array == NULL) || (sizeb_array == NULL) || (sizec_array == NULL) || (a_array ==
NULL) ||
(b_array == NULL) || (c_array == NULL) || (a_array_dev == NULL) || (b_array_dev == NULL)
||
(c_array_dev == NULL) || (c_ref_array == NULL)) {
printf("Cannot allocate matrices and size arrays\n");
return 1;
}

idx = 0;
for (i = 0; i < GROUP_COUNT; i++) {
    sizea = (((layout == CblasRowMajor) && (transA[i] == CblasTrans)) ||
              ((layout == CblasColMajor) && (transA[i] == CblasNoTrans))) ? lda[i] * k[i] :
m[i] * lda[i];
    sizeb = (((layout == CblasRowMajor) && (transB[i] == CblasTrans)) ||
              ((layout == CblasColMajor) && (transB[i] == CblasNoTrans))) ? ldb[i] * n[i] :
k[i] * ldb[i];
    sizec = (layout == CblasColMajor) ? ldc[i] * n[i] : ldc[i] * m[i];
    for (j = 0; j < group_size[i]; j++) {
        a_array[idx] = (double *)mkl_malloc(sizeof(double) * sizea, 64);
        a_array_dev[idx] = a_array[idx];
        sizea_array[idx] = sizea;
        if (a_array[idx] == NULL) {
            printf("cannot allocate a matrices\n");
            return 1;
        }
    }
}

```

```
b_array[idx] = (double *)mkl_malloc(sizeof(double) * sizeb, 64);
b_array_dev[idx] = b_array[idx];
sizeb_array[idx] = sizeb;
if (b_array[idx] == NULL) {
    printf("cannot allocate b matrices\n");
    return 1;
}
c_array[idx] = (double *)mkl_malloc(sizeof(double) * sizec, 64);
c_array_dev[idx] = c_array[idx];
sizec_array[idx] = sizec;
if (c_array[idx] == NULL) {
    printf("cannot allocate c matrices\n");
    return 1;
}
c_ref_array[idx] = (double *)mkl_malloc(sizeof(double) * sizec, 64);
if (c_ref_array[idx] == NULL) {
    printf("cannot allocate c_ref matrices\n");
    return 1;
}
init_double_array(sizea, a_array[idx], 1);
init_double_array(sizeb, b_array[idx], 1);
init_double_array(sizec, c_array[idx], 1);
for (p = 0; p < sizec_array[idx]; p++) c_ref_array[idx][p] = c_array[idx][p];
idx++;
}
}

// run gemm_batch on host, use standard oneMKL interface
cblas_dgemm_batch(layout, transA, transB, m, n, k, alpha, (const double **) a_array, lda,
                  (const double **) b_array, ldb, beta, c_ref_array, ldc, GROUP_COUNT,
group_size);

double *a, *b, *c;
for (i = 0; i < total_batch_size; i++) {
    a = a_array[i];
    b = b_array[i];
    c = c_array[i];
#pragma omp target enter data map(to:a[0:sizea_array[i]],b[0:sizeb_array[i]],c[0:sizec_array[i]])
#pragma omp target data use_device_ptr(a,b,c)
    {
        a_array_dev[i] = a;
        b_array_dev[i] = b;
        c_array_dev[i] = c;
    }
}
#pragma omp target data map(to:a_array_dev[0:total_batch_size], \
                           b_array_dev[0:total_batch_size], \
                           c_array_dev[0:total_batch_size]) device(dnum)
{

#pragma omp dispatch
    cblas_dgemm_batch(layout, transA, transB, m, n, k, alpha, (const double **) a_array_dev,
lda, (const double **) b_array_dev, ldb, beta, c_array_dev, ldc, GROUP_COUNT, group_size);
}

for (i = 0; i < total_batch_size; i++) {
    a = a_array[i];
```

```

        b = b_array[i];
        c = c_array[i];
#pragma omp target exit data
map(from:a[0:sizea_array[i]],b[0:sizeb_array[i]],c[0:sizetc_array[i]])
}

double computed, reference, diff;
MKL_INT l;
idx = 0;
for (p = 0; p < GROUP_COUNT; p++) {
    for (l = 0; l < group_size[p]; l++) {
        for (i = 0; i < m[p]; i++) {
            for (j = 0; j < n[p]; j++) {
                if (layout == CblasColMajor) {
                    computed = c_array[idx][i + j * ldc[p]];
                    reference = c_ref_array[idx][i + j * ldc[p]];
                }
                else {
                    computed = c_array[idx][j + i * ldc[p]];
                    reference = c_ref_array[idx][j + i * ldc[p]];
                }
                diff = computed - reference;
                diff = (diff > 0) ? diff : -diff;
                if (diff > 0.0001) {
#endif MKL_ILP64
                    printf("Error in matrix %lld (group = %lld, matrix index in group =
%lld) at index [%lld][%lld], computed = %lf, reference = %lf, difference = %lf\n", idx, p, l, i,
j, computed, reference, diff);
#else
                    printf("Error in matrix %d at index [%d][%d], computed = %lf, reference
= %lf, difference = %lf\n", idx, i, j, computed, reference, diff);
#endif
                }
                free_double_matrices(a_array, total_batch_size);
                free_double_matrices(b_array, total_batch_size);
                free_double_matrices(c_array, total_batch_size);
                free_double_matrices(c_ref_array, total_batch_size);
                mkl_free(a_array);
                mkl_free(b_array);
                mkl_free(c_array);
                mkl_free(c_ref_array);
                mkl_free(a_array_dev);
                mkl_free(b_array_dev);
                mkl_free(c_array_dev);
                mkl_free(sizea_array);
                mkl_free(sizeb_array);
                mkl_free(sizetc_array);
                mkl_free(transA); mkl_free(transB);
                mkl_free(m); mkl_free(n); mkl_free(k);
                mkl_free(lda); mkl_free(ldb); mkl_free(ldc); mkl_free(group_size);
                mkl_free(alpha); mkl_free(beta);
                return 1;
            }
        }
        idx++;
    }
}

```

```
printf("Validation PASSED\n");
free_double_matrices(a_array, total_batch_size);
free_double_matrices(b_array, total_batch_size);
free_double_matrices(c_array, total_batch_size);
free_double_matrices(c_ref_array, total_batch_size);
mkl_free(a_array);
mkl_free(b_array);
mkl_free(c_array);
mkl_free(c_ref_array);
mkl_free(a_array_dev);
mkl_free(b_array_dev);
mkl_free(c_array_dev);
mkl_free(sizea_array);
mkl_free(sizeb_array);
mkl_free(sizec_array);
mkl_free(transA); mkl_free(transB);
mkl_free(m); mkl_free(n); mkl_free(k);
mkl_free(lda); mkl_free(ldb); mkl_free(ldc); mkl_free(group_size);
mkl_free(alpha); mkl_free(beta);
return 0;
}
```

## Speeding Up Independent, Consecutive GEMM Calls

There are various ways to speed up the execution of consecutive GEMM calls that can be executed independently. One way is to batch the GEMM calls by calling the batch version of GEMM as shown above.

Another way is to enclose the calls to GEMM by an OpenMP parallel construct, so each OpenMP thread executing the parallel region dispatches one of the GEMM calls. This parallel approach is illustrated in the following example.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include "mkl.h"
#include "mkl_omp_offload.h"

#define min(x,y) (((x) < (y)) ? (x) : (y))
#define epsilon 0.0000001f

bool compare(double x, double y)
{
    // returns true if x and y are the same
    return fabs(x - y) <= epsilon;
}

int main()
{
    double *A1, *B1, *C1, *C1_f1;
    double *A2, *B2, *C2, *C2_f1;
    int m, n, k, i, j, q;
    double alpha, beta;
    double sum;
    int fail;
    double t_start, t_end;
```

```
m = 2000, k = 200, n = 1000;
alpha = 1.0; beta = 0.0;

printf (" Allocating memory for matrices aligned on 64-byte boundary for better \n"
        " performance \n\n");
A1 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
B1 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
C1 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
C1_fl = (double *)mkl_malloc (m*n*sizeof( double ), 64 );

A2 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
B2 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
C2 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
C2_fl = (double *)mkl_malloc (m*n*sizeof( double ), 64 );

if (A1 == NULL || B1 == NULL || C1 == NULL || C1_fl == NULL ||
    A2 == NULL || B2 == NULL || C2 == NULL || C2_fl == NULL) {
    printf( "\n ERROR: Can't allocate memory for matrices. Aborting... \n\n");
    return 1;
}

printf (" Intializing matrix data \n\n");
for (i = 0; i < (m*k); i++) {
    A1[i] = A2[i] = (double)(i+1);
}

for (i = 0; i < (k*n); i++) {
    B1[i] = B2[i] = (double)(-i-1);
}

for (i = 0; i < (m*n); i++) {
    C1[i] = C2[i] = 0.0;
    C1_fl[i] = C2_fl[i] = 0.0;
}

printf (" \nComputing matrix product using Intel MKL cblas_dgemm function \n");

t_start = omp_get_wtime();

#pragma omp target data
map(to: A1[0:m*k], B1[0:k*n], A2[0:m*k], B2[0:k*n]) \
map(tofrom: C1[0:m*n], C2[0:m*n])
{
    #pragma omp parallel num_threads(2)
    {
        int id = omp_get_thread_num();

        if (id == 0) {
            #pragma omp dispatch
            cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                        m, n, k, alpha, A1, k, B1, n, beta, C1, n);
        }
        else if (id == 1) {
            #pragma omp dispatch
            cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                        m, n, k, alpha, A2, k, B2, n, beta, C2, n);
        }
    }
}
```

```
}

t_end = omp_get_wtime();

printf ("\n Top left corner of matrix C1: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C1[j+i*n]);
    }
    printf ("\n");
}

printf ("\n Top left corner of matrix C2: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C2[j+i*n]);
    }
    printf ("\n");
}

printf (" \nComputing matrix product using for-loops \n");

for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (q = 0; q < k; q++)
            sum += A1[k*i+q] * B1[n*q+j];
        C1_fl[n*i+j] = sum;
    }
}

for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (q = 0; q < k; q++)
            sum += A2[k*i+q] * B2[n*q+j];
        C2_fl[n*i+j] = sum;
    }
}

printf ("\n Top left corner of matrix C1: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C1_fl[j+i*n]);
    }
    printf ("\n");
}

printf ("\n Top left corner of matrix C2: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C2_fl[j+i*n]);
    }
    printf ("\n");
}

printf ("\n Computations completed. Verifying... \n\n");
```

```

fail = 0;
for (i = 0; i < (m*n); i++) {
    if (! compare(C1[i], C1_f1[i]) || ! compare(C2[i], C2_f1[i])) {
        fail = 1;
        break;
    }
}

if (fail) {
    printf (" **** FAIL **** \n");
}
else {
    printf(" time = %lf seconds\n", t_end - t_start);
    printf (" **** PASS **** \n");
}

mkl_free(A1);
mkl_free(B1);
mkl_free(C1);
mkl_free(C1_f1);
mkl_free(A2);
mkl_free(B2);
mkl_free(C2);
mkl_free(C2_f1);

return 0;
}

```

Yet another way to speed up the execution of independent, consecutive GEMM calls is to use the `nowait` clause on the `dispatch` construct so the host thread does not have to wait for a dispatched GEMM call to complete before dispatching the next one. After the last GEMM call, we insert an OpenMP `taskwait` directive to guarantee that all the dispatched MKL calls complete before the host thread proceeds any further. This `nowait` approach is illustrated in the following example.

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include "mkl.h"
#include "mkl_omp_offload.h"

#define min(x,y) (((x) < (y)) ? (x) : (y))
#define epsilon 0.0000001f

bool compare(double x, double y)
{
    // returns true if x and y are the same
    return fabs(x - y) <= epsilon;
}

int main()
{
    double *A1, *B1, *C1, *C1_f1;
    double *A2, *B2, *C2, *C2_f1;
    int m, n, k, i, j, q;
    double alpha, beta;
    double sum;
    int fail;

```

```

double t_start, t_end;

m = 2000, k = 200, n = 1000;
alpha = 1.0; beta = 0.0;

printf (" Allocating memory for matrices aligned on 64-byte boundary for better \n"
        " performance \n\n");
A1 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
B1 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
C1 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
C1_f1 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );

A2 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
B2 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
C2 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
C2_f1 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );

if (A1 == NULL || B1 == NULL || C1 == NULL || C1_f1 == NULL ||
    A2 == NULL || B2 == NULL || C2 == NULL || C2_f1 == NULL) {
    printf( "\n ERROR: Can't allocate memory for matrices. Aborting... \n\n");
    return 1;
}

printf (" Intializing matrix data \n\n");
for (i = 0; i < (m*k); i++) {
    A1[i] = A2[i] = (double)(i+1);
}

for (i = 0; i < (k*n); i++) {
    B1[i] = B2[i] = (double)(-i-1);
}

for (i = 0; i < (m*n); i++) {
    C1[i]      = C2[i]      = 0.0;
    C1_f1[i]   = C2_f1[i]   = 0.0;
}

printf (" \nComputing matrix product using Intel MKL cblas_dgemm function \n");

t_start = omp_get_wtime();

#pragma omp target data
map(to: A1[0:m*k], B1[0:k*n], A2[0:m*k], B2[0:k*n]) \
map(tofrom: C1[0:m*n], C2[0:m*n])
{
    #pragma omp dispatch nowait
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                m, n, k, alpha, A1, k, B1, n, beta, C1, n);

    #pragma omp dispatch nowait
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                m, n, k, alpha, A2, k, B2, n, beta, C2, n);

    #pragma omp taskwait
}

t_end = omp_get_wtime();

```

```

printf ("\n Top left corner of matrix C1: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C1[j+i*n]);
    }
    printf ("\n");
}

printf ("\n Top left corner of matrix C2: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C2[j+i*n]);
    }
    printf ("\n");
}

printf (" \nComputing matrix product using for-loops \n");

for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (q = 0; q < k; q++)
            sum += A1[k*i+q] * B1[n*q+j];
        C1_fl[n*i+j] = sum;
    }
}

for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (q = 0; q < k; q++)
            sum += A2[k*i+q] * B2[n*q+j];
        C2_fl[n*i+j] = sum;
    }
}

printf ("\n Top left corner of matrix C1: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C1_fl[j+i*n]);
    }
    printf ("\n");
}

printf ("\n Top left corner of matrix C2: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C2_fl[j+i*n]);
    }
    printf ("\n");
}

printf ("\n Computations completed. Verifying... \n\n");

fail = 0;
for (i = 0; i < (m*n); i++) {
    if (! compare(C1[i], C1_fl[i]) || ! compare(C2[i], C2_fl[i])) {
        fail = 1;
    }
}

```

```

        break;
    }

    if (fail) {
        printf (" **** FAIL **** \n");
    }
    else {
        printf(" time = %lf seconds\n", t_end - t_start);
        printf (" **** PASS **** \n");
    }

    mkl_free(A1);
    mkl_free(B1);
    mkl_free(C1);
    mkl_free(C1_fl);
    mkl_free(A2);
    mkl_free(B2);
    mkl_free(C2);
    mkl_free(C2_fl);

    return 0;
}

```

## Padding of Matrices Used in GEMM Computations

GEMM calls can be sped up by padding the leading dimensions, `lda`, `ldb`, and `ldc`, of the matrices `a`, `b`, and `c`, respectively.

The leading dimension of a matrix depends on the layout of the matrix in memory:

- Column major layout (C/C++): The leading dimension of a matrix is the number of columns of the matrix.
- Row major layout (Fortran): The leading dimension of a matrix is the number of rows of the matrix.

There are two rules to keep in mind for choosing the sizes of matrices passed to DGEMM calls.

**Rule 1:** For best performance, the leading dimensions of matrices passed to GEMM calls should be a multiple of 64 bytes (full cache line size). For single precision data, this means the leading dimension should be a multiple of `(64 / sizeof(float))`, which is equal to 16. For double precision data, this means the leading dimension should be a multiple of `(64 / sizeof(double))`, which is equal to 8.

Preferably, all matrices (`a`, `b`, and `c`) should be padded. However, padding matrix `c` is less important than padding `a` and `b`.

**Rule 2:** For best performance, leading dimensions should not be a multiple of a large power of 2 (e.g. 4096 bytes). Increasing the leading dimension slightly (e.g. from 4096 bytes to 4096+64 bytes) can improve performance in some cases.

### Padding Example (Fortran)

The following Fortran example illustrates how matrices passed to DGEMM calls may be padded for improved performance.

```

include "mkl_omp_offload.f90"

! This subroutine reads command line arguments m1, k1, and n1.
subroutine get_arguments (m1, k1, n1)
    implicit none
    integer :: m1, k1, n1
    character(len=32) :: m1_char, k1_char, n1_char

```

```
! First, make sure that the right number of command line arguments
! have been provided.
if (command_argument_count() .ne. 3) then
    print *, "ERROR: Three command-line arguments expected; stopping."
    stop
endif

! Get command line arguments.
call get_command_argument(1, m1_char)
call get_command_argument(2, k1_char)
call get_command_argument(3, n1_char)

! Convert arguments to integers.
read (m1_char,*) m1
read (k1_char,*) k1
read (n1_char,*) n1
end subroutine get_arguments

! This function returns the smallest multiple of 8 that is >= n.
! Examples:
! if n = 3, then get_mul8 = 8
! if n = 9, then get_mul8 = 16
! if n = 30, then get_mul8 = 32
! if n = 80, then get_mul8 = 8
integer function get_mul8 (n)
    implicit none
    integer :: n
    integer :: mod
    if (mod(n,8) .eq. 0) then
        get_mul8 = n
    else
        get_mul8 = ((n/8) + 1) * 8
    endif
end function get_mul8

! This subroutine initializes matrices.
subroutine init_matrix (m, k, n, a, b, c)
    implicit none
    integer :: m, k, n
    double precision :: a(m,k), b(k,n), c(m,n)
    integer :: i, j

    do i = 1, m
        do j = 1, k
            a(i,j) = (i-1) - (0.25 * k)
        end do
    end do

    do i = 1, k
        do j = 1, n
            b(i,j) = -((i-1) + j)
        end do
    end do

    do i = 1, m
```

```
    do j = 1, n
        c(i,j) = 0.2 + i - j
    end do
end do
end subroutine init_matrix

program DGEMM_MAIN

#if defined(MKL_ILP64)
    use onemkl_blas_omp_offload_ilp64
#else
    use onemkl_blas_omp_offload_lp64
#endif
use omp_lib
use iso_fortran_env
implicit none

interface
    integer function get_mul8 (n)
        implicit none
        integer :: n
    end function get_mul8
end interface

double precision :: alpha, beta
integer :: m1, k1, n1, m2, k2, n2
double precision, allocatable :: a1(:,:)
double precision, allocatable :: b1(:,:)
double precision, allocatable :: c1(:,:)

double precision, allocatable :: a2(:,:)
double precision, allocatable :: b2(:,:)
double precision, allocatable :: c2(:,:)

double precision :: start_t1, end_t1
double precision :: start_t2, end_t2

! Read command line arguments m1, k1, and n1.

call get_arguments (m1, k1, n1)

!
! Initialize alpha, beta, and m2, k2, n2
!

alpha = 1.025
beta = 0.75

m2 = get_mul8(m1)
k2 = get_mul8(k1)
n2 = get_mul8(n1)

!
! Allocate and initialize matrices.
!
allocate( a1(1:m1,1:k1) )
allocate( b1(1:k1,1:n1) )
```

```
allocate( c1(1:m1,1:n1) )
allocate( a2(1:m2,1:k2) )
allocate( b2(1:k2,1:n2) )
allocate( c2(1:m2,1:n1) )
call init_matrix (m1, k1, n1, a1, b1, c1)
call init_matrix (m2, k2, n2, a2, b2, c2)

!$omp target data map(to: a1, b1, a2, b2) map(tofrom: c1, c2)

! Warm up run on device
 !$omp dispatch
 call DGEMM('N','N',m1,n1,k1,alpha,a1,m1,b1,k1,beta,c1,m1)

!
! Run DGEMM on device (using matrices a1, b1, and c1)
!
 start_t1 = omp_get_wtime()

 !$omp dispatch
 call DGEMM('N','N',m1,n1,k1,alpha,a1,m1,b1,k1,beta,c1,m1)

 end_t1 = omp_get_wtime()

! Warm up run on device
 !$omp dispatch
 call DGEMM('N','N',m2,n2,k2,alpha,a2,m2,b2,k2,beta,c2,m2)

!
! Run DGEMM on device (using padded matrices a2, b2, and c2)
!
 start_t2 = omp_get_wtime()

 !$omp dispatch
 call DGEMM('N','N',m2,n2,k2,alpha,a2,m2,b2,k2,beta,c2,m2)

 end_t2 = omp_get_wtime()

 !$omp end target data

 print 100, alpha, beta
print *
print 101, m1, n1, k1
print 111, (end_t1 - start_t1)
print *
print 102, m2, n2, k2
print 112, (end_t2 - start_t2)

100  format(7x, "ALPHA =", f10.4, " BETA =",f10.4)
101  format(7x, "M1 =", i5," N1 =", i5, " K1 =",i5)
111  format(7x, "Time (non-padded arrays) =", f10.4, " sec")
102  format(7x, "M2 =", i5," N2 =", i5, " K2 =",i5)
112  format(7x, "Time (padded arrays) =", f10.4, " sec")

end
```

In the above example, the array bounds (`m1`, `k1`, and `n1`) are input as command line arguments. The matrices `a1(1:m1, 1:k1)`, `b1(1:k1, 1:n1)`, and `c1(1:m1, 1:n1)` are allocated and initialized.

Also, the padded matrices `a2(1:m2, 1:k2)`, `b2(1:k2, 1:n2)`, and `c2(1:m2, 1:n2)` are allocated and initialized. `m2` is the smallest multiple of 8 that is  $\geq m1$ . Similarly, `k2` is the smallest multiple of 8 that is  $\geq k1$ , and `n2` is the smallest multiple of 8 that is  $\geq n1$ .

The program compares the time taken by the DGEMM computation on `a1`, `b1`, and `c1` versus the time taken by the DGEMM computation on `a2`, `b2`, and `c2`.

The compilation, link and run commands used are shown below.

```
Compile:  
ifx -fopenmp -fopenmp-targets=spir64 -qmkl -fpp -free -c dgemm_pad_f_01.f  
  
Link:  
ifx -fopenmp -fopenmp-targets=spir64 -qmkl -lOpenCL dgemm_pad_f_01.o  
  
Run:  
ZE_AFFINITY_MASK=0 LIBOMPTARGET_DEBUG=0 ./a.out 12001 12001 12001
```

The output on the particular GPU used (1-stack only) was as follows:

```
ALPHA = 1.0250 BETA = 0.7500  
  
M1 =12001 N1 =12001 K1 =12001  
Time (non-padded arrays) = 0.2388 sec  
  
M2 =12008 N2 =12008 K2 =12008  
Time (padded arrays) = 0.1648 sec
```

The above shows that padding arrays `a`, `b`, and `c`, the time taken by the DGEMM calls was reduced from 0.2388 seconds to 0.1648 seconds.

### Padding Example (C/C++)

The following is a C/C++ example that illustrates how matrices passed to DGEMM calls may be padded for improved performance.

```
#include "mkl.h"  
#include "mkl_omp_offload.h"  
#include <float.h>  
#include <math.h>  
#include <omp.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/time.h>  
  
#if defined(PRECISION)  
#if PRECISION == 1  
#define FLOAT double  
#else  
#define FLOAT float  
#endif  
#else  
#define PRECISION 1  
#define FLOAT double  
#endif  
  
#define index(i, j, ld) (((j) * (ld)) + (i))
```

```
#define RAND() ((FLOAT)rand() / (FLOAT)RAND_MAX * 2.0 - 1.0)

#define MALLOC(x) mkl_malloc((x), 64);
#define FREE mkl_free

#define MALLOC_CHECK(p) \
    if (p == NULL) { \
        fprintf(stderr, "%s:%d: memory allocation error\n", __FILE__, __LINE__); \
        return EXIT_FAILURE; \
    }

#ifndef max
#define max(a, b) (((a) > (b)) ? (a) : (b))
#endif
#ifndef min
#define min(a, b) (((a) < (b)) ? (a) : (b))
#endif

void printMat(FLOAT *P, int m, int n, int ld) {
    int i, j;
    for (i = 0; i < m; i++) {
        printf("\n");
        for (j = 0; j < n; j++)
            printf("%f ", P[index(i, j, ld)]);
    }
    printf("\n");
}

void gemm_naive(int m, int n, int k, FLOAT alpha, const FLOAT *A, int lda,
                const FLOAT *B, int ldb, FLOAT beta, FLOAT *C, int ldc) {
    int i, j, kk;
    FLOAT temp;
    for (j = 0; j < n; j++) {
        for (i = 0; i < m; i++) {
            temp = 0.0;
            for (kk = 0; kk < k; kk++)
                temp += alpha * A[index(i, kk, lda)] * B[index(kk, j, ldb)];
            C[index(i, j, ldc)] = temp + beta * C[index(i, j, ldc)];
        }
    }
}

FLOAT infinity_norm_error(int m, int n, int ld, const FLOAT *ans,
                           const FLOAT *ref) {
    int i, j, ind;
    FLOAT norm = 0.0, temp;
    for (i = 0; i < m; i++) {
        temp = 0.0;
        for (j = 0; j < n; j++) {
            ind = index(i, j, ld);
            temp += fabs(ref[ind] - ans[ind]);
        }
        norm = max(norm, temp);
    }
    return norm;
}
```

```
double mysecond() {
    struct timeval tp;
    struct timezone tzp;
    int i;
    i = gettimeofday(&tp, &tzp);
    if (i != 0) {
        fprintf(stderr, "%s:%d: timing error %d\n", __FILE__, __LINE__, i);
        return EXIT_FAILURE;
    }
    return ((double)tp.tv_sec + (double)tp.tv_usec * 1.e-6);
}

#if defined(USE_MKL)
int dnum = 0;
#endif

#if PRECISION == 1
#define LD_ALIGN 256
#define LD_BIAS 8
#else
#define LD_ALIGN 512
#define LD_BIAS 16
#endif

#define HPL_PTR(ptr_, al_) (((size_t)(ptr_) + (al_)-1) / (al_)) * (al_)

#if defined(PAD_LD)
static inline int getld(int x) {
    int ld;
    ld = HPL_PTR(x, LD_ALIGN); // Rule 1
    if (ld - LD_BIAS >= x)
        ld -= LD_BIAS;
    else
        ld += LD_BIAS; // Rule 2
    return ld;
}
#else
static inline int getld(int x) { return x; }
#endif

int main(int argc, char **argv) {
    int i, j;

    if ((argc < 4) || (argc > 4 && argc < 8)) {
        printf("Performs a DGEMM test C = alpha*A*B + beta*C\n");
        printf("A matrix is MxK and B matrix is KxN\n");
        printf("All matrices are stored in column-major format\n");
        printf("Run as ./dgemm_cublas <M> <K> <N> [<alpha> <beta> <iterations> "
               "<verify>]\n");
        printf("Required inputs are:\n");
        printf("      M: number of rows of matrix A\n");
        printf("      K: number of cols of matrix A\n");
        printf("      N: number of cols of matrix B\n");
        printf("Optional inputs are (all must be provided if providing any):\n");
        printf("      alpha: scalar multiplier (default: 1.0)\n");
        printf("      beta: scalar multiplier (default: 0.0)\n");
        printf("      iterations: number of blocking DGEMM calls to perform "
               "(default: 10)\n");
    }
}
```

```
printf("      verify: set to 1 to check solution against CPU reference, "
      "not recommended for large M|K|N (default: 0)\n");
return EXIT_FAILURE;
}

FLOAT alpha, beta;
int niter, verify;
int HA = atoi(argv[1]);
int WA = atoi(argv[2]);
int WB = atoi(argv[3]);

if ((HA == 0) || (WA == 0) || (WB == 0))
    exit(1);

if (argc > 4) {

#if PRECISION == 1
    sscanf(argv[4], "%lf", &alpha);
    sscanf(argv[5], "%lf", &beta);
#else
    sscanf(argv[4], "%f", &alpha);
    sscanf(argv[5], "%f", &beta);
#endif
    niter = atoi(argv[6]);
    verify = atoi(argv[7]);
} else {
    alpha = 1.0;
    beta = 0.0;
    niter = 10;
    verify = 0;
}

#if PRECISION == 1
    printf("DGEMM performance test\n");
#else
    printf("SGEMM performance test\n");
#endif

int HB = WA;
int WC = WB;
int HC = HA;

int ldA = getld(HA);
int ldB = getld(HB);
int ldC = getld(HC);

printf("M = %d, K = %d, N = %d, ldA = %d, ldB = %d, ldC = %d, alpha = %f, "
      "beta = %f, iterations = %d, verify? = %d\n",
      HA, WA, WB, ldA, ldB, ldC, alpha, beta, niter, verify);

double start_t, end_t, tot_t = 0.0, best_t = DBL_MAX;

/*ALLOCATE HOST ARRAYS*/
FLOAT *A = (FLOAT *)MALLOC(ldA * WA * sizeof(FLOAT));
MALLOC_CHECK(A);
FLOAT *B = (FLOAT *)MALLOC(ldB * WB * sizeof(FLOAT));
MALLOC_CHECK(B);
FLOAT *C = (FLOAT *)MALLOC(ldC * WC * sizeof(FLOAT));
```

```

MALLOC_CHECK(C);
FLOAT *Cref = NULL;
if (verify) {
    Cref = (FLOAT *)MALLOC(ldC * WC * sizeof(FLOAT));
    MALLOC_CHECK(Cref);
}

printf("\n-----\n");
printf("Array A: %d x %d\n", ldA, WA);
printf("Array B: %d x %d\n", ldB, WB);
printf("Array C: %d x %d\n", ldC, WC);
printf("-----\n");

/*INITIALIZE WITH PSEUDO-RANDOM DATA*/
srand(2864);
for (j = 0; j < WA; j++)
    for (i = 0; i < HA; i++)
        A[index(i, j, ldA)] = RAND();

for (j = 0; j < WB; j++)
    for (i = 0; i < HB; i++)
        B[index(i, j, ldB)] = RAND();

if (beta != 0.0) {
    for (j = 0; j < WC; j++)
        for (i = 0; i < HC; i++)
            C[index(i, j, ldC)] = RAND();
} else {
    for (j = 0; j < WC; j++)
        for (i = 0; i < HC; i++)
            C[index(i, j, ldC)] = 0.0;
}

if (verify) {
    for (j = 0; j < WC; j++)
        for (i = 0; i < HC; i++)
            Cref[index(i, j, ldC)] = C[index(i, j, ldC)];
}

#ifndef USE_MKL
size_t sizea = (size_t)ldA * WA;
size_t sizeb = (size_t)ldB * WB;
size_t sizec = (size_t)ldC * WC;
#endif

#pragma omp target data map(to: A [0:sizea], B [0:sizeb]) map(tofrom: C [0:sizec])
{
    // warm-up run
    #pragma omp dispatch
#endif PRECISION == 1
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, HA, WB, WA, alpha, A,
                ldA, B, ldB, beta, C, ldC);
#else
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, HA, WB, WA, alpha, A,
                ldA, B, ldB, beta, C, ldC);
#endif

    // run gemm on gpu, using dispatch construct
}

```

```

for (i = 0; i < niter; i++) {
    start_t = mysecond();

    #pragma omp dispatch
#endif PRECISION == 1
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, HA, WB, WA, alpha,
                A, ldA, B, ldB, beta, C, ldc);
#else
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, HA, WB, WA, alpha,
                A, ldA, B, ldB, beta, C, ldc);
#endif

    end_t = mysecond();
    tot_t += end_t - start_t;
    best_t = min(best_t, end_t - start_t);
} // end for
}
#endif // end #pragma omp target data

double tflop_count;
tflop_count = (double)2.0 * HA * WB * WA;
if (beta != 0.0)
    tflop_count += (double)HA * WB;
tflop_count *= 1.E-12;

printf("Total runtime for %d iterations: %f seconds.\n", niter, tot_t);
printf("Mean TFLOP/s: %f\n", (double)niter * tflop_count / tot_t);
printf("Best TFLOP/s: %f\n", (double)tflop_count / best_t);

if (verify) {
    // compute reference solution on host (1 added to niter to account for the
    // warm-up run)
    for (i = 0; i < (niter + 1); i++) {
        gemm_naive(HA, WB, WA, alpha, A, ldA, B, ldB, beta, Cref, ldc);
    }

    printf("Error Matrix Infinity Norm = %f\n",
           infinity_norm_error(HA, WB, ldc, C, Cref));
}

FREE(A);
FREE(B);
FREE(C);
if (verify)
    FREE(Cref);

return EXIT_SUCCESS;
}

```

The program reads command line arguments that specify the dimensions of the matrices `a`, `b`, and `c`, as well as some additional (optional) parameters. If the `PAD_LD` macro is defined, then the leading dimensions of the matrices are padded, according to Rule1 and Rule2 above. Then the program calls `cblas_dgemm` or `cblas_sgemm` `niter` times, and outputs the total runtime.

### Compile and run (``LD_PAD`` macro is not defined)

Compile:

```
icpx -fopenmp -fopenmp-targets=spir64 -qmkl -DUSE_MKL -DPRECISION=1 -c dgemm_pad_c_01.cpp
```

```
icpx -fopenmp -fopenmp-targets=spir64 -qmkl -lOpenCL dgemm_pad_c_01.o -o dgemm_nopad.exe
```

Run:

```
ZE_AFFINITY_MASK=0 LIBOMPTARGET_DEBUG=0 ./dgemm_nopad.exe 4001 4001 4001 1.0 0.0 100 0
```

The performance (with no padding) was as follows on the particular GPU used (1-stack only):

```
DGEMM performance test
M = 15000, K = 15000, N = 15000, ldA = 15000, ldB = 15000, ldC = 15000, alpha = 1.000000, beta = 0.000000, iterations = 100, verify? = 0

-----
Array A: 15000 x 15000
Array B: 15000 x 15000
Array C: 15000 x 15000

-----
Total runtime for 100 iterations: 138.368065 seconds.
Mean TFLOP/s: 4.878293
Best TFLOP/s: 5.657242
```

### Compile and run (``LD\_PAD`` macro is defined)

Compile:

```
icpx -fopenmp -fopenmp-targets=spir64 -qmkl -DUSE_MKL -DPRECISION=1 -DPAD_LD -c
dgemm_pad_c_01.cpp
```

Link:

```
icpx -fopenmp -fopenmp-targets=spir64 -qmkl -lOpenCL dgemm_pad_c_01.o -o dgemm_pad.exe
```

Run:

```
ZE_AFFINITY_MASK=0 LIBOMPTARGET_DEBUG=0 ./dgemm_pad.exe 4001 4001 4001 1.0 0.0 100 0
```

The performance with padding, where the leading dimensions of matrices a, b, and c was increased from 15000 to 15096), was as follows on the particular GPU used (1-stack only).

```
DGEMM performance test
M = 15000, K = 15000, N = 15000, ldA = 15096, ldB = 15096, ldC = 15096, alpha = 1.000000, beta = 0.000000, iterations = 100, verify? = 0

-----
Array A: 15096 x 15000
Array B: 15096 x 15000
Array C: 15096 x 15000

-----
Total runtime for 100 iterations: 145.219459 seconds.
Mean TFLOP/s: 4.648137
Best TFLOP/s: 5.525268
```

## References

- [1. Developer Reference for Intel® oneAPI Math Kernel Library - C](#)
- [2. Developer Reference for Intel® oneAPI Math Kernel Library - Fortran](#)
- [3. Developer Reference for Intel® oneAPI Math Kernel Library - Fortran \(Matrix Arguments\)](#)
- [4. Introducing Batch GEMM Operations](#)
- [5. Intel® oneAPI Math Kernel Library Link Line Advisor](#)

## Tools for Analyzing Performance of OpenMP Applications

There are various tools and mechanisms available that help in analyzing the performance of OpenMP programs and identifying bottlenecks.

- Intel® VTune™ Profiler

Intel® VTune™ Profiler can be used to analyze the performance of an application. It helps identify the most time-consuming (hot) functions in the application, whether the application is CPU- or GPU-bound, how effectively it offloads code to the GPU, and the best sections of code to optimize for sequential performance and for threaded performance, among other things. For more information about VTune Profiler, refer to the [Intel® VTune™ Profiler User Guide](#).

- Intel® Level Zero and OpenCL Tracing Tool

The [unitrace](#) is a host and device activity tracing tool for Level Zero and OpenCL backend with support for SYCL and OpenMP GPU offload. For information about this tool, see the [Performance Tools in Intel® Profiling Tools Interfaces for GPU](#) section of this document.

When using unitrace with the `-h` and `-d` options, look at host- and device-side summaries at the end of the trace, under the headings "API Timing Summary" and "Device Timing Summary", respectively.

Note that only explicit data transfers appear in the trace. Transfers of data allocated in Unified Shared Memory (USM) may not appear in the trace.

**Note:**

- unitrace is useful for confirming that offloading of oneMKL kernels has occurred. The environment variable `OMP_TARGET_OFFLOAD=MANDATORY` environment variable does not affect oneMKL, and therefore cannot be used to guarantee that offloading of oneMKL kernels has occurred. One way to check that offloading of oneMKL kernels (and other kernels) has occurred is to see which kernels are listed under "Device Timing Results" in the trace generated by unitrace.

**SYCL\_PI\_TRACE=2 environment variable.** The DPC++ Runtime Plugin Interface (PI) is an interface layer between the device-agnostic part of SYCL runtime and the device-specific runtime layers which control execution on devices. Setting `SYCL_PI_TRACE=2` provides a trace of all PI calls made with arguments and returned values. For more information, see the [DPC++ Runtime Plugin Interface documentation](#).

**LIBOMPTARGET\_DEBUG=1 environment variable.** `LIBOMPTARGET_DEBUG` controls whether or not debugging information from libomptarget.so will be displayed.

The debugging output provides useful information about things like ND-range partitioning of loop iterations, data transfers between host and device, memory usage, etc., as shown in the :[Using More GPU Resources](#) and :[Minimizing Data Transfers and Memory Allocations](#) sections of this document.

For more information about `LIBOMPTARGET_DEBUG`, see [LLVM/OpenMP Runtimes](#).

**LIBOMPTARGET\_PLUGIN\_PROFILE environment variable.** `LIBOMPTARGET_PROFILE` allows libomptarget.so to generate time profile output. For more information, see [LLVM/OpenMP Runtimes](#).

**Dump of compiler-generated assembly for the device.** You can dump the compiler-generated assembly by setting the following two environment variables before doing Just-In-Time (JIT) compilation (or before running the program in the case of Ahead-Of-Time (AOT) compilation).

```
export IGC_ShaderDumpEnable=1
export IGC_DumpToCustomDir=my_dump_dir
```

LLVM IR, assembly, and GenISA files will be dumped in the sub-directory named `my_dump_dir` (or any other name you choose). In this sub-directory, you will find a `*.asm` file for each kernel. The filename indicates the source line number on which the kernel occurs. The header of the file provides information about SIMD width, compiler options, as well as other information.

Also, in `my_dump_dir`, you will find a file named `HardwareCaps.txt` that provides information about the GPU, such as Vector Engine count, thread count, slice count, etc.

For more information about the Intel® Graphics Compiler and a listing of available flags (environment variables) to control the compilation, see [Intel® Graphics Compiler for OpenCL™ Configuration Flags for Linux Release](#)

For additional information about debugging and profiling, refer to the [Performance Profiling and Analysis](#) section of this document.

## 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](#)
- [Minimizing Data Transfers and Memory Allocations](#)
- [Making Better Use of OpenMP Constructs](#)
- [Memory Allocation](#)
- [Fortran Example](#)
- [Clauses: is\\_device\\_ptr, use\\_device\\_ptr, has\\_device\\_addr, use\\_device\\_addr](#)
- [Prefetching](#)

**Note:**

Used the following when collecting OpenMP performance numbers:

- 2-stack Intel® GPU
- One GPU stack only (no implicit or explicit scaling).
- Intel® compilers, runtimes, and GPU drivers
- Level-Zero plugin
- Introduced a dummy target construct at the beginning of a program, so as not to measure startup time.
- Just-In-Time (JIT) compilation mode.

## Using More GPU Resources

The performance of offloaded code can be improved by using a larger number of work-items that can run in parallel, thus utilizing more GPU resources (filling up the GPU).

**Note:**

- ND-range partitioning of loop iterations is decided by compiler and runtime heuristics, and also depends on the GPU driver and the hardware configuration. So it can change over time. However, the methodology of figuring out the partitioning based on LIBOMPARGET\_DEBUG=1 output will remain the same.

## 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` directive indicates that the outermost loop (on line 53) is parallel. The number of iterations in the loop is `BLOCKS`, which is equal to 8.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <math.h>
#include <omp.h>

#define P 16
#define BLOCKS 8
```

```
#define SIZE (BLOCKS * P * P * P)

#define MAX 100
#define scaled_rand() ((rand() % MAX) / (1.0 * MAX))

#define IDX2(i, j) (i * P + j)
#define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)

int main(void) {
    double w[SIZE];           /* output */
    double u[SIZE], dx[P * P]; /* input */
    int b, i, j, k, l;        /* loop counters */
    double start, end;        /* timers */

    omp_set_default_device(0);

    /* dummy target region, so as not to measure startup time. */
    #pragma omp target
    { ; }

    /* initialize input with random values */
    srand(0);
    for (int i = 0; i < SIZE; i++)
        u[i] = scaled_rand();

    for (int i = 0; i < P * P; i++)
        dx[i] = scaled_rand();

    /* map data to device */
    #pragma omp target enter data map(to: u[0:SIZE], dx[0:P * P])

    start = omp_get_wtime();

    /* offload the kernel with no collapse clause */
    #pragma omp target teams distribute parallel for \
        private(b, i, j, k, l)
    for (b = 0; b < BLOCKS; b++) {
        for (i = 0; i < P; i++) {
            for (j = 0; j < P; j++) {
                for (k = 0; k < P; k++) {
                    double ur = 0.;
                    double us = 0.;
                    double ut = 0.;

                    for (l = 0; l < P; l++) {
                        ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
                        us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
                        ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
                    }

                    w[IDX4(b, i, j, k)] = ur * us * ut;
                }
            }
        }
    }

    end = omp_get_wtime();
}
```

```
#pragma omp target exit data map(from: w[0:SIZE])

/* print result */
printf("no-collapse-clause: w[0]=%lf time=%lf\n", w[0], end - start);

return 0;
}
```

### Compilation command:

```
icx -fiopenmp -fopenmp-targets=spir64 test_noCollapse.cpp
```

### Run command:

```
OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0 LIBOMPTARGET_DEBUG=1 ./a.out
```

libomptarget.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, L<sub>b</sub> and U<sub>b</sub> 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 50.

```
Libomptarget --> Launching target execution __omp_offloading_3d_9b5f515d_Z4main_145 with
pointer 0x00000000143d5d8 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x00000000143d5d8...
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 = {1, 1, 1}
Target LEVEL0 RTL --> Group counts = {8, 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 8 work-groups, with one work-item each (total work-group count = 8 x 1 x 1 = 8, and each work-group size = 1 x 1 x 1 = 1 work-item). The kernel is vectorized using SIMD 32, which means every 32 work-items in a work-group are combined into one sub-group. Since we have only one work-item per work-group, it follows that each work-group has only one sub-group where only one SIMD lane is 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.

```
/* offload the kernel with collapse clause */
#pragma omp target teams distribute parallel for collapse(2) \
    private(b, i, j, k, l)
for (b = 0; b < BLOCKS; b++) {
    for (i = 0; i < P; i++) {
        for (j = 0; j < P; j++) {
            for (k = 0; k < P; k++) {
                double ur = 0.;
                double us = 0.;
                double ut = 0.;

                for (l = 0; l < P; l++) {
                    ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
                    us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
                    ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
                }
            }
        }
    }
}
```

```

        w[IDX4(b, i, j, k)] = ur * us * ut;
    }
}
}
}
}
```

LIBOMPTARGET\_DEBUG=1 output shows the following partitioning when collapse(2) is used.

```

Libomptarget --> Launching target execution __omp_offloading_3d_9b5f515f__Z4main_145 with
pointer 0x00000000017f45d8 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x00000000017f45d8...
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 = 15, Stride = 1
Target LEVEL0 RTL --> Level 1: Lb = 0, Ub = 7, Stride = 1
Target LEVEL0 RTL --> Group sizes = {1, 1, 1}
Target LEVEL0 RTL --> Group counts = {16, 8, 1}
```

Note that with collapse(2), the number of parallel loop iterations = BLOCKS x P = 8 x 16 = 128. In this case, we end up with 128 work-groups, and each work-group has 1 work-item (total work-group count = 16 x 8 x 1 = 128, and each work-group size = 1 x 1 x 1 = 1 work-item). The kernel is vectorized using SIMD 32, which means every 32 work-items in a work-group are combined into one sub-group. Since we have only one work-item per work-group, it follows that each work-group has only one sub-group where only one SIMD lane is active.

On the other hand, if we use the collapse(3) clause, LIBOMPTARGET\_DEBUG=1 output shows the following partitioning.

```

Libomptarget --> Launching target execution __omp_offloading_3d_9b5f5160__Z4main_145 with
pointer 0x0000000001728d08 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x0000000001728d08...
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 = 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 = {8, 1, 1}
Target LEVEL0 RTL --> Group counts = {2, 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 256 work-groups, and each work-group has 8 work-items (total work-group count = 2 x 16 x 8 = 256, and each work-group size = 8 x 1 x 1 = 8 work-items). The kernel is vectorized using SIMD 32, which means every 32 work-items in a work-group are combined into one sub-group. Since we have only 8 work-items per work-group, it follows that we have only one sub-group where only 8 SIMD lanes are active.

If we were to use the collapse(4) clause, instead of collapse(3), LIBOMPTARGET\_DEBUG=1 output shows the following partitioning.

```

Target LEVEL0 RTL --> Executing a kernel 0x0000000001aab5d8...
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 = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {64, 1, 1}
Target LEVEL0 RTL --> Group counts = {512, 1, 1}
```

With `collapse(4)`, the number of resulting parallel loop iterations =  $BLOCKS \times P \times P \times P = 8 \times 16 \times 16 \times 16 = 32768$ . In this case, we have 512 work-groups, and each work-group has 64 work-items (total work-group count =  $512 \times 1 \times 1 = 512$ , and each work-group size =  $64 \times 1 \times 1 = 64$  work-items). The kernel is vectorized using SIMD 32, which means every 32 work-items are combined into one sub-group. It follows that each work-group has 2 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 GPU used (1-stack only) was as follows:

```
no collapse version : 0.002430 seconds
collapse(2) version : 0.000839 seconds
collapse(3) version : 0.000321 seconds
collapse(4) version : 0.000325 seconds
```

The above timings show that adding the `collapse(3)` or `collapse(4)` clause gives a performance boost of about 7.5x. (0.000321 seconds versus 0.002430 seconds).

#### Notes:

- 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 host and device.

Place the `target enter data` directive before the first `target` construct to transfer data from host to device, and place the `target exit data` directive after the last `target` construct to transfer data from device to host.

Consider the following example where we have two `target` constructs (on lines 47 and 71), and each `target` construct reads arrays `dx` and `u` and writes to array `w`.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <math.h>
#include <omp.h>

#define P 16
#define BLOCKS 8
#define SIZE (BLOCKS * P * P * P)

#define MAX 100
#define scaled_rand() ((rand() % MAX) / (1.0 * MAX))

#define IDX2(i, j) (i * P + j)
#define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)
```

```
int main(void) {
    double w[SIZE];           /* output */
    double u[SIZE], dx[P * P]; /* input */
    int b, i, j, k, l;        /* loop counters */
    double start, end;        /* timers */

    omp_set_default_device(0);

    /* dummy target region, so as not to measure startup time. */
    #pragma omp target
    { ; }

    /* initialize input with random values */
    srand(0);
    for (int i = 0; i < SIZE; i++)
        u[i] = scaled_rand();

    for (int i = 0; i < P * P; i++)
        dx[i] = scaled_rand();

    start = omp_get_wtime();

    /* offload kernel #1 */
    #pragma omp target teams distribute parallel for collapse(4) \
        map(to: u[0:SIZE], dx[0:P * P]) map(from: w[0:SIZE]) \
        private(b, i, j, k, l)
    for (b = 0; b < BLOCKS; b++) {
        for (i = 0; i < P; i++) {
            for (j = 0; j < P; j++) {
                for (k = 0; k < P; k++) {
                    double ur = 0.;
                    double us = 0.;
                    double ut = 0.;

                    for (l = 0; l < P; l++) {
                        ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
                        us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
                        ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
                    }
                    w[IDX4(b, i, j, k)] = ur * us * ut;
                }
            }
        }
    }

    /* offload kernel #2 */
    #pragma omp target teams distribute parallel for collapse(4) \
        map(to: u[0:SIZE], dx[0:P * P]) map(tofrom: w[0:SIZE]) \
        private(b, i, j, k, l)
    for (b = 0; b < BLOCKS; b++) {
        for (i = 0; i < P; i++) {
            for (j = 0; j < P; j++) {
                for (k = 0; k < P; k++) {
                    double ur = b + i + j - k;
                    double us = b + i + j - k;
                    double ut = b + i + j - k;
                }
            }
        }
    }
}
```

```

        for (l = 0; l < P; l++) {
            ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
            us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
            ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
        }

        w[IDX4(b, i, j, k)] += ur * us * ut;
    }
}
}

end = omp_get_wtime();

/* print result */
printf("target region: w[0]=%lf time=%lf\n", w[0], end - start);

return 0;
}

```

**Compilation command:**

```
icx-cl -fiopenmp -fopenmp-targets=spir64 test_no_target_enter_exit_data.cpp
```

**Run command:**

```
OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0 LIBOMPTARGET_DEBUG=1 ./a.out
```

When the first target construct (on line 47) is encountered:

- Since arrays `dx` and `u` appear in a `map` clause with the `to` map-type, storage is allocated for the arrays 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 71) 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 47 and 71) have the same data partitioning.

```

Libomptarget --> Launching target execution __omp_offloading_3d_15ece5c8__Z4main_142 with
pointer 0x00000000024cb5d8 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x00000000024cb5d8...
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 = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {64, 1, 1}
Target LEVEL0 RTL --> Group counts = {512, 1, 1}

Target LEVEL0 RTL --> Executing a kernel 0x0000000002b9c5e0...
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 = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {64, 1, 1}
Target LEVEL0 RTL --> Group counts = {512, 1, 1}
Target LEVEL0 RTL --> Kernel Pointer argument 0 (value: 0xff00fffffe0000) was set successfully
for device 0.

```

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:0x00007fff60f05030) -> (tgt:0xff00fffffe0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fff60ec5030) -> (tgt:0xff00fffffe0000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00fffffe20000) -> (hst:0x00007fff60e85030)
Libomptarget --> Moving 2048 bytes (hst:0x00007fff60f05030) -> (tgt:0xff00fffffe0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fff60ec5030) -> (tgt:0xff00fffffe0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fff60e85030) -> (tgt:0xff00fffffe20000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00fffffe20000) -> (hst:0x00007fff60e85030)

```

You can reduce the copying of data from host to device and vice versa by using the target enter data and target exit data directives as shown in this modified example.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <math.h>
#include <omp.h>

#define P 16
#define BLOCKS 8
#define SIZE (BLOCKS * P * P * P)

#define MAX 100
#define scaled_rand() ((rand() % MAX) / (1.0 * MAX))

#define IDX2(i, j) (i * P + j)
#define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)

int main(void) {
    double w[SIZE];           /* output */
    double u[SIZE], dx[P * P]; /* input */
    int b, i, j, k, l;        /* loop counters */
    double start, end;        /* timers */

    omp_set_default_device(0);

    /* dummy target region, so as not to measure startup time. */
    #pragma omp target
    { ; }

    /* initialize input with random values */
    srand(0);

```

```
for (int i = 0; i < SIZE; i++)
    u[i] = scaled_rand();

for (int i = 0; i < P * P; i++)
    dx[i] = scaled_rand();

start = omp_get_wtime();

/* map data to device. alloc for w avoids map(tofrom: w[0:SIZE])
   on target by default. */
#pragma omp target enter data map(to: u[0:SIZE], dx[0:P * P]) \
map(alloc: w[0:SIZE])

/* offload kernel #1 */
#pragma omp target teams distribute parallel for collapse(4) \
private(b, i, j, k, l)
for (b = 0; b < BLOCKS; b++) {
    for (i = 0; i < P; i++) {
        for (j = 0; j < P; j++) {
            for (k = 0; k < P; k++) {
                double ur = 0.;
                double us = 0.;
                double ut = 0.;

                for (l = 0; l < P; l++) {
                    ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
                    us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
                    ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
                }
                w[IDX4(b, i, j, k)] = ur * us * ut;
            }
        }
    }
}

/* offload kernel #2 */
#pragma omp target teams distribute parallel for collapse(4) \
private(b, i, j, k, l)
for (b = 0; b < BLOCKS; b++) {
    for (i = 0; i < P; i++) {
        for (j = 0; j < P; j++) {
            for (k = 0; k < P; k++) {
                double ur = b + i + j - k;
                double us = b + i + j - k;
                double ut = b + i + j - k;

                for (l = 0; l < P; l++) {
                    ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
                    us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
                    ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
                }
                w[IDX4(b, i, j, k)] += ur * us * ut;
            }
        }
    }
}
```

```

#pragma omp target exit data map(from: w[0:SIZE])

end = omp_get_wtime();

/* print result */
printf("target region: w[0]=%lf time=%lf\n", w[0], end - start);

return 0;
}

```

In the modified example, when the `target enter data` directive (on line 48) 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 52) 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 75) 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 device to host will occur.

When the `target exit data` directive (on line 97) 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 GPU used (1-stack only):

```

No target enter/exit data version : 0.001204 seconds
target enter/exit data version     : 0.000934 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`):

```

Libomptarget --> Looking up mapping(HstPtrBegin=0x00007ffd899939c0, Size=2048)...
Libomptarget --> Mapping exists with HstPtrBegin=0x00007ffd899939c0,
TgtPtrBegin=0xff00fffffee0000, Size=2048, DynRefCount=2 (update suppressed), HoldRefCount=0
Libomptarget --> Obtained target argument (Begin: 0xff00fffffee0000, Offset: 0) from host
pointer 0x00007ffd899939c0
Libomptarget --> Looking up mapping(HstPtrBegin=0x00007ffd899539c0, Size=262144)...
Libomptarget --> Mapping exists with HstPtrBegin=0x00007ffd899539c0,
TgtPtrBegin=0xff00fffffef0000, Size=262144, DynRefCount=2 (update suppressed), HoldRefCount=0

```

```

Libomptarget --> Obtained target argument (Begin: 0xff00fffffef0000, Offset: 0) from host
pointer 0x00007ffd899539c0
Libomptarget --> Looking up mapping(HstPtrBegin=0x00007ffd899139c0, Size=262144)...

Libomptarget --> Launching target execution __omp_offloading_3d_fadb4d_Z4main_147 with pointer
0x0000000002b9c5d8 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x0000000002b9c5d8...
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 = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {64, 1, 1}
Target LEVEL0 RTL --> Group counts = {512, 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:0x00007ffd899539c0) -> (tgt:0xff00fffffef0000)
Libomptarget --> Moving 2048 bytes (hst:0x00007ffd899939c0) -> (tgt:0xff00fffffee0000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00fffffff30000) -> (hst:0x00007ffd899139c0)

```

## 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`.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <math.h>
#include <omp.h>

#define P 16
#define BLOCKS 8
#define SIZE (BLOCKS * P * P * P)

#define MAX 100
#define scaled_rand() ((rand() % MAX) / (1.0 * MAX))

#define IDX2(i, j) (i * P + j)
#define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)

int main(void) {
    double w[SIZE];           /* output */
    double u[SIZE], dx[P * P]; /* input */
    int b, i, j, k, l;        /* loop counters */
    double start, end;        /* timers */

    omp_set_default_device(0);

    /* dummy target region, so as not to measure startup time. */
    #pragma omp target
    { ; }

    /* initialize input with random values */

```

```

srand(0);
for (int i = 0; i < SIZE; i++)
    u[i] = scaled_rand();

for (int i = 0; i < P * P; i++)
    dx[i] = scaled_rand();

start = omp_get_wtime();

#pragma omp target teams distribute parallel for \
private(b, i, j, k, l) \
map(tofrom: u[0:SIZE], dx[0:P * P]) \
map(tofrom: w [0:SIZE])
for (int n = 0; n < SIZE; n++) {
    k = n - (n / P) * P;
    j = (n - k) / P;
    i = (n - (j * P + k)) / (P * P);
    b = n / (P * P * P);

    double ur = 0.;
    double us = 0.;
    double ut = 0.;

    for (l = 0; l < P; l++) {
        ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
        us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
        ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
    }

    w[IDX4(b, i, j, k)] = ur * us * ut;
}

end = omp_get_wtime();

printf("offload: w[0]=%lf time=%lf\n", w[0], end - start);

return 0;
}

```

### Compilation command:

```
icpx -fopenmp -fopenmp-targets=spir64 test_map_tofrom.cpp
```

### Run command:

```
OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0 LIBOMPTARGET_DEBUG=1 ./a.out
```

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.

```

#pragma omp target teams distribute parallel for \
private(b, i, j, k, l) \
map(to: u[0:SIZE], dx[0:P * P]) \
map(from: w [0:SIZE])
for (int n = 0; n < SIZE; n++) {
    k = n - (n / P) * P;
    j = (n - k) / P;
    i = (n - (j * P + k)) / (P * P);
    b = n / (P * P * P);

```

```

double ur = 0.;
double us = 0.;
double ut = 0.;

for (l = 0; l < P; l++) {
    ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
    us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
    ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
}

w[IDX4(b, i, j, k)] = ur * us * ut;
}

```

Using more specific map-types (to or from, instead of tofrom), reduced the runtime on the particular GPU used (1-stack only):

```

tofrom map-types version      : 0.001141 seconds
to or from map-types version : 0.000908  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:0x00007fff1f6ad540) -> (tgt:0xff00ffffffffe0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fff1f66d540) -> (tgt:0xff00ffffffffe0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fff1f62d540) -> (tgt:0xff00ffffffff20000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00ffffffff20000) -> (hst:0x00007fff1f62d540)
Libomptarget --> Moving 262144 bytes (tgt:0xff00ffffffffe0000) -> (hst:0x00007fff1f66d540)
Libomptarget --> Moving 2048 bytes (tgt:0xff00ffffffffe0000) -> (hst:0x00007fff1f6ad540)

```

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 host to device, and one transfer to copy the values of `w` from device to host:

```

$ grep "Libomptarget --> Moving" test_map_to_or_from.debug
Libomptarget --> Moving 2048 bytes (hst:0x00007fffc2258fd0) -> (tgt:0xff00ffffffffe0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fffc2218fd0) -> (tgt:0xff00ffffffffe0000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00ffffffff20000) -> (hst:0x00007fffc21d8fd0)

```

## 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.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <math.h>
#include <omp.h>

```

```
#define P 16
#define BLOCKS 8
#define SIZE (BLOCKS * P * P * P)

#define MAX 100
#define scaled_rand() ((rand() % MAX) / (1.0 * MAX))

#define IDX2(i, j) (i * P + j)
#define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)

int main(void) {
    double w[SIZE]; /* output */
    double u[SIZE], dx[P * P]; /* input */
    double s1, s2, s3; /* scalars */
    int b, i, j, k, l; /* loop counters */
    double start, end; /* timers */

    omp_set_default_device(0);

    /* dummy target region, so as not to measure startup time. */
    #pragma omp target
    { ; }

    /* initialize input with random values */
    srand(0);
    for (int i = 0; i < SIZE; i++)
        u[i] = scaled_rand();

    for (int i = 0; i < P * P; i++)
        dx[i] = scaled_rand();

    /* initialize scalars */
    s1 = u[SIZE / 2];
    s2 = scaled_rand();
    s3 = 0.145;

    /* map data to device */
    #pragma omp target enter data map(to: u[0:SIZE], dx[0:P * P])

    start = omp_get_wtime();

    /* offload the kernel with collapse clause */
    #pragma omp target teams distribute parallel for collapse(4) \
        map(to: s1, s2, s3) private(b, i, j, k, l)
    for (b = 0; b < BLOCKS; b++) {
        for (i = 0; i < P; i++) {
            for (j = 0; j < P; j++) {
                for (k = 0; k < P; k++) {
                    double ur = 0.;
                    double us = 0.;
                    double ut = 0.;

                    for (l = 0; l < P; l++) {
                        ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)] + s1;
                        us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)] - s2;
                        ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)] * s3;
                    }
                }
            }
        }
    }
}
```

```

        w[IDX4(b, i, j, k)] = ur * us * ut;
    }
}
}

end = omp_get_wtime();

#pragma omp target exit data map(from: w[0:SIZE])

/* print result */
printf("collapse-clause: w[0]=%lf time=%lf\n", w[0], end - start);

return 0;
}

```

**Compilation command:**

```
icpx -fiopenmp -fopenmp-targets=spir64 test_scalars_map.cpp
```

**Run command:**

```
OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0 LIBOMPTARGET_DEBUG=1 ./a.out
```

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.

```

/* offload the kernel with collapse clause */
#pragma omp target teams distribute parallel for collapse(4) \
    firstprivate(s1, s2, s3) private(b, i, j, k, l)
for (b = 0; b < BLOCKS; b++) {
    for (i = 0; i < P; i++) {
        for (j = 0; j < P; j++) {
            for (k = 0; k < P; k++) {
                double ur = 0.;
                double us = 0.;
                double ut = 0.;

                for (l = 0; l < P; l++) {
                    ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)] + s1;
                    us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)] - s2;
                    ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)] * s3;
                }

                w[IDX4(b, i, j, k)] = ur * us * ut;
            }
        }
    }
}

```

Using `firstprivate(s1, s2, s3)`, instead of `map(to:s1, s2, s3)`, reduced the runtime on the particular GPU used (1-stack 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)`.

```
Libomptarget --> Launching target execution __omp_offloading_3d_9b49d7d8__Z4main_151 with
pointer 0x0000000002b295d8 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x0000000002b295d8...
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 = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {64, 1, 1}
Target LEVEL0 RTL --> Group counts = {512, 1, 1}
```

```
Libomptarget --> Launching target execution __omp_offloading_3d_9b49d7dd__Z4main_151 with
pointer 0x0000000001f475d8 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x0000000001f475d8...
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 = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {64, 1, 1}
Target LEVEL0 RTL --> Group counts = {512, 1, 1}
```

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.

```
Target LEVEL0 RTL --> Memory usage for device memory, device 0x000000000278e470
Target LEVEL0 RTL --> -- Allocator: Native, Pool
Target LEVEL0 RTL --> -- Requested: 1179648, 526360
Target LEVEL0 RTL --> -- Allocated: 1179648, 526528
Target LEVEL0 RTL --> -- Freed : 1179648, 262336
Target LEVEL0 RTL --> -- InUse : 0, 264192
Target LEVEL0 RTL --> -- PeakUse : 1179648, 526528
Target LEVEL0 RTL --> -- NumAllocs: 3, 6
```

Note that the memory allocated is 1,179,648 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 fewer 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.

```
Target LEVEL0 RTL --> Memory usage for device memory, device 0x0000000001bab440
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, using the `map(to: )` clause results in more data transfers from host to device. This can be seen by grepping for "Libomptarget --> Moving" in the **LIBOMPTARGET\_DEBUG=1** output:

```
$ grep "Libomptarget --> Moving" test_scalars_map.debug
Libomptarget --> Moving 262144 bytes (hst:0x00007ffdf5526760) -> (tgt:0xff00ffffffff0000)
Libomptarget --> Moving 2048 bytes (hst:0x00007ffdf5566760) -> (tgt:0xff00ffffffff0000)
Libomptarget --> Moving 8 bytes (hst:0x00007ffdf55670a0) -> (tgt:0xff00ffffffff0000)
```

```
Libomptarget --> Moving 8 bytes (hst:0x00007ffd55670a8) -> (tgt:0xff00fffffffffed0040)
Libomptarget --> Moving 8 bytes (hst:0x00007ffd55670b0) -> (tgt:0xff00fffffffffed0080)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffd54e6760) -> (tgt:0xff00ffffffff30000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00ffffffff30000) -> (hst:0x00007ffd54e6760)
```

In contrast, when the `firstprivate(to:s1, s2, s3)` clause is used, LIBOMPTARGET\_DEBUG=1 output shows:

```
$ grep "Libomptarget --> Moving" test_scalars_fp.debug
Libomptarget --> Moving 262144 bytes (hst:0x00007ffda809c4a0) -> (tgt:0xff00ffffffffef0000)
Libomptarget --> Moving 2048 bytes (hst:0x00007ffda80dc4a0) -> (tgt:0xff00ffffffffee0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffda805c4a0) -> (tgt:0xff00ffffffff30000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00ffffffff30000) -> (hst:0x00007ffda805c4a0)
```

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 host to device.

## Do Not Map Loop Bounds to Get Better ND-Range Partitioning

As mentioned earlier, 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 host to device. Loop partitioning will also be affected, and may result in 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 53).

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <math.h>
#include <omp.h>

#define P 16
#define BLOCKS 8
#define SIZE (BLOCKS * P * P * P)

#define MAX 100
#define scaled_rand() ((rand() % MAX) / (1.0 * MAX))

#define IDX2(i, j) (i * P + j)
#define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)

int main(void) {
    double w[SIZE];           /* output */
    double u[SIZE], dx[P * P]; /* input */
    int b, i, j, k, l;        /* loop counters */
    int upper;
    double start, end;        /* timers */

    omp_set_default_device(0);

    /* dummy target region, so as not to measure startup time. */
    #pragma omp target
```

```

{ ; }

/* initialize input with random values */
srand(0);
for (int i = 0; i < SIZE; i++)
    u[i] = scaled_rand();

for (int i = 0; i < P * P; i++)
    dx[i] = scaled_rand();

upper = (int)dx[0] + SIZE;

/* map data to device */
#pragma omp target enter data map(to: u[0:SIZE], dx[0:P * P])

start = omp_get_wtime();

/* offload kernel */
#pragma omp target teams distribute parallel for private(b, i, j, k, l) \
map(to: upper)
for (int n = 0; n < upper; n++) {
    double ur = 0.;
    double us = 0.;
    double ut = 0.;

    k = n - (n / P) * P;
    j = (n - k) / P;
    i = (n - (j * P + k)) / (P * P);
    b = n / (P * P * P);

    for (l = 0; l < P; l++) {
        ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
        us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
        ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
    }

    w[IDX4(b, i, j, k)] = ur * us * ut;
}

end = omp_get_wtime();

/* map data from device */
#pragma omp target exit data map(from: w[0:SIZE])

printf("offload: w[0]=%lf time=%lf\n", w[0], end - start);

return 0;
}

```

### Compilation command:

```
icpx -fiopenmp -fopenmp-targets=spir64 test_loop_bounds_map.cpp
```

### Run command:

```
OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0 LIBOMPTARGET_DEBUG=1 ./a.out
```

Since `upper` is mapped, the value of the variable `upper` on the host may be different from the value on the device. Because of this, 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.so`) 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, 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, some of the work-items will not do any work. An if-condition inside the loop generated by the compiler will 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.

```
Libomptarget --> Launching target execution __omp_offloading_3d_1ff4bf1c__Z4main_148 with
pointer 0x00000000021175d8 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x00000000021175d8...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Group sizes = {1024, 1, 1}
Target LEVEL0 RTL --> Group counts = {512, 1, 1}
```

Note that in the above partitioning, the total number of work-items =  $512 \times 1024 = 524,288$ , 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.

```
#pragma omp target teams distribute parallel for private(b, i, j, k, l) \
    firstprivate(upper)
for (int n = 0; n < upper; n++) {
    double ur = 0.;
    double us = 0.;
    double ut = 0.;

    k = n - (n / P) * P;
    j = (n - k) / P;
    i = (n - (j * P + k)) / (P * P);
    b = n / (P * P * P);

    for (l = 0; l < P; l++) {
        ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
        us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
        ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
    }

    w[IDX4(b, i, j, k)] = ur * us * ut;
}
```

For the modified example (where `upper` is `firstprivate`), `LIBOMPTARGET_DEBUG=1` shows the following ND-range partitioning.

```
Libomptarget --> Launching target execution __omp_offloading_3d_1fed0edf__Z4main_148 with
pointer 0x00000000029b3d08 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x00000000029b3d08...
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 = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {64, 1, 1}
Target LEVEL0 RTL --> Group counts = {512, 1, 1}
```

Note that in the above partitioning, the total number of work-items =  $512 \times 64 = 32,767$ , which exactly matches the actual number of loop iterations.

Using `firstprivate(upper)` instead of `map(to:upper)` reduced the runtime on the particular GPU used (1-stack only):

```
map(to:upper) version      : 0.000415 seconds
firstprivate(upper) version : 0.000307 seconds
```

## Allocate Memory Directly on the Device

As is known, 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 51-53).

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <math.h>
#include <omp.h>

#define P 16
#define BLOCKS 8
#define SIZE (BLOCKS * P * P * P)

#define MAX 100
#define scaled_rand() ((rand() % MAX) / (1.0 * MAX))

#define IDX2(i, j) (i * P + j)
#define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)

int main(void) {
    double w[SIZE]; /* output */
    double u[SIZE], dx[P * P]; /* input */
    double ur[SIZE], us[SIZE], ut[SIZE]; /* work arrays */
    int b, i, j, k, l; /* loop counters */
    double start, end; /* timers */

    omp_set_default_device(0);

    /* dummy target region, so as not to measure startup time. */
    #pragma omp target
    { ; }

    /* initialize input with random values */
    srand(0);
    for (int i = 0; i < SIZE; i++)
        u[i] = scaled_rand();

    for (int i = 0; i < P * P; i++)
```

```

dx[i] = scaled_rand();

start = omp_get_wtime();

/* offload the kernel */
#pragma omp target teams distribute parallel for simd simdlen(16) collapse(4) \
map(to:u[0:SIZE],dx[0:P*P]) \
map(from:w[0:SIZE]) \
map(to:ur[0:SIZE]) \
map(to:us[0:SIZE]) \
map(to:ut[0:SIZE]) \
private(b,i,j,k,l)
for (b = 0; b < BLOCKS; b++) {
    for (i = 0; i < P; i++) {
        for (j = 0; j < P; j++) {
            for (k = 0; k < P; k++) {
                w[IDX4(b, i, j, k)] = 0.;
                ur[IDX4(b, i, j, k)] = 0.;
                us[IDX4(b, i, j, k)] = 0.;
                ut[IDX4(b, i, j, k)] = 0.;

                for (l = 0; l < P; l++) {
                    ur[IDX4(b, i, j, k)] += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
                    us[IDX4(b, i, j, k)] += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
                    ut[IDX4(b, i, j, k)] += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
                }

                w[IDX4(b, i, j, k)] = ur[IDX4(b, i, j, k)] * us[IDX4(b, i, j, k)] *
                    ut[IDX4(b, i, j, k)];
            }
        }
    }
}

end = omp_get_wtime();

/* print result */
printf("collapse-clause: w[0]=%lf time=%lf\n", w[0], end - start);

return 0;
}

```

### Compilation command:

```
icpx -fopenmp -fopenmp-targets=spir64 test_map_to.cpp
```

### Run command:

```
OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0 LIBOMPTARGET_DEBUG=1 ./a.out
```

The amount of data transferred between host and 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 host to device for each of the arrays:

```
$ grep "Libomptarget --> Moving" test_map_to.debug
Libomptarget --> Moving 262144 bytes (hst:0x00007fffca630880) -> (tgt:0xff00ffffffff30000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fffca670880) -> (tgt:0xff00ffffffff70000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fffca6b0880) -> (tgt:0xff00ffffffffb0000)
```

```
Libomptarget --> Moving 2048 bytes (hst:0x00007fffca770880) -> (tgt:0xff00fffffee0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fffca730880) -> (tgt:0xff00fffffdde0000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00fffffef0000) -> (hst:0x00007fffca6f0880)
```

These data transfers are wasteful because 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 the `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 host and device. This approach is illustrated in the following modified example.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <math.h>
#include <omp.h>

#define P 16
#define BLOCKS 8
#define SIZE (BLOCKS * P * P * P)

#define MAX 100
#define scaled_rand() ((rand() % MAX) / (1.0 * MAX))

#define IDX2(i, j) (i * P + j)
#define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)

#pragma omp declare target
double ur[SIZE], us[SIZE], ut[SIZE]; /* work arrays */
#pragma omp end declare target

int main(void) {
    double w[SIZE]; /* output */
    double u[SIZE], dx[P * P]; /* input */
    int b, i, j, k, l; /* loop counters */
    double start, end; /* timers */

    omp_set_default_device(0);

    /* dummy target region, so as not to measure startup time. */
    #pragma omp target
    { ; }

    /* initialize input with random values */
    srand(0);
    for (int i = 0; i < SIZE; i++)
        u[i] = scaled_rand();

    for (int i = 0; i < P * P; i++)
        dx[i] = scaled_rand();

    start = omp_get_wtime();

    /* offload the kernel */
    #pragma omp target teams distribute parallel for simd simdlen(16) collapse(4) \
        map(to:u[0:SIZE],dx[0:P*P]) \
        map(from:w[0:SIZE]) \

```

```

private(b,i,j,k,l)
for (b = 0; b < BLOCKS; b++) {
    for (i = 0; i < P; i++) {
        for (j = 0; j < P; j++) {
            for (k = 0; k < P; k++) {
                w[IDX4(b, i, j, k)] = 0.;
                ur[IDX4(b, i, j, k)] = 0.;
                us[IDX4(b, i, j, k)] = 0.;
                ut[IDX4(b, i, j, k)] = 0.;

                for (l = 0; l < P; l++) {
                    ur[IDX4(b, i, j, k)] += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
                    us[IDX4(b, i, j, k)] += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
                    ut[IDX4(b, i, j, k)] += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
                }
            }
        }
    }
}

end = omp_get_wtime();

/* print result */
printf("collapse-clause: w[0]=%lf time=%lf\n", w[0], end - start);

return 0;
}

```

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 host to device for each of the arrays:

```
$ grep "Libomptarget --> Moving" test_declare_target.debug
Libomptarget --> Moving 2048 bytes (hst:0x00007ffc546bfec0) -> (tgt:0xff00ffffffff0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffc5467fec0) -> (tgt:0xff00ffffffff30000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00ffffffffef0000) -> (hst:0x00007ffc5463fec0)
```

An alternative approach for allocating memory on the device, without transferring any data between host and device, uses the `map(alloc: )` clause instead of the `map(to: )` clause, as shown below (lines 51-53).

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <math.h>
#include <omp.h>

#define P 16
#define BLOCKS 8
#define SIZE (BLOCKS * P * P * P)

#define MAX 100
#define scaled_rand() ((rand() % MAX) / (1.0 * MAX))

#define IDX2(i, j) (i * P + j)

```

```
#define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)

int main(void) {
    double w[SIZE]; /* output */
    double u[SIZE], dx[P * P]; /* input */
    double ur[SIZE], us[SIZE], ut[SIZE]; /* work arrays */
    int b, i, j, k, l; /* loop counters */
    double start, end; /* timers */

    omp_set_default_device(0);

    /* dummy target region, so as not to measure startup time. */
    #pragma omp target
    { ; }

    /* initialize input with random values */
    srand(0);
    for (int i = 0; i < SIZE; i++)
        u[i] = scaled_rand();

    for (int i = 0; i < P * P; i++)
        dx[i] = scaled_rand();

    start = omp_get_wtime();

    /* offload the kernel */
    #pragma omp target teams distribute parallel for simd simdlen(16) collapse(4) \
        map(to:u[0:SIZE],dx[0:P*P]) \
        map(from:w[0:SIZE]) \
        map(alloc:ur[0:SIZE]) \
        map(alloc:us[0:SIZE]) \
        map(alloc:ut[0:SIZE]) \
        private(b,i,j,k,l)
    for (b = 0; b < BLOCKS; b++) {
        for (i = 0; i < P; i++) {
            for (j = 0; j < P; j++) {
                for (k = 0; k < P; k++) {
                    w[IDX4(b, i, j, k)] = 0.;
                    ur[IDX4(b, i, j, k)] = 0.;
                    us[IDX4(b, i, j, k)] = 0.;
                    ut[IDX4(b, i, j, k)] = 0.;

                    for (l = 0; l < P; l++) {
                        ur[IDX4(b, i, j, k)] += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
                        us[IDX4(b, i, j, k)] += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
                        ut[IDX4(b, i, j, k)] += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
                    }
                }
            }
        }
    }

    end = omp_get_wtime();

    /* print result */
}
```

```

    printf("collapse-clause: w[0]=%lf time=%lf\n", w[0], end - start);

    return 0;
}

```

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:0x00007ffd46f256c0) -> (tgt:0xff00ffffffffe0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffd46ee56c0) -> (tgt:0xff00ffffffffde0000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00ffffffffef0000) -> (hst:0x00007ffd46ea56c0)

```

The performance of the various versions when running on the particular GPU used (1-stack only) was as follows:

```

map(to: ) version : 0.001430 seconds
declare target / end declare target version : 0.000874 seconds
map(alloc: ) version : 0.000991 seconds

```

## Making Better Use of OpenMP Constructs

### Reduce Synchronizations Using `nowait`

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 48). Half of the computations are performed on the host by the team of threads executing the `parallel` region. The other half of the computations are 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.

```

/*
 * This test is taken from OpenMP API 5.0.1 Examples (June 2020)
 * https://www.openmp.org/wp-content/uploads/openmp-examples-5-0-1.pdf
 * (4.13.2 nowait Clause on target Construct)
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

#define N 100000 // N must be even

void init(int n, float *v1, float *v2) {
    int i;

```

```

for(i=0; i<n; i++){
    v1[i] = i * 0.25;
    v2[i] = i - 1.25;
}
}

int main() {
    int i, n=N;
    float v1[N],v2[N],vxv[N];
    double start,end; // timers

    init(n, v1, v2);

    /* Dummy parallel and target regions, so as not to measure startup
       time. */
    #pragma omp parallel
    {
        #pragma omp master
        #pragma omp target
        {};
    }

    start=omp_get_wtime();

    #pragma omp parallel
    {
        #pragma omp master
        #pragma omp target teams distribute parallel for      \
            map(to: v1[0:n/2])                                \
            map(to: v2[0:n/2])                                \
            map(from: vxv[0:n/2])
        for(i=0; i<n/2; i++){
            vxv[i] = v1[i]*v2[i];
        }
        /* Master thread will wait for target region to be completed
           before proceeding beyond this point. */

        #pragma omp for
        for(i=n/2; i<n; i++) {
            vxv[i] = v1[i]*v2[i];
        }
        /* Implicit barrier at end of worksharing for. */
    }

    end=omp_get_wtime();

    printf("vxv[0]=%f, vxv[n-1]=%f, time=%lf\n", vxv[0], vxv[n-1], end-start);
    return 0;
}

```

**Compilation command:**

```
icpx -fiopenmp -fopenmp-targets=spir64 test_target_no_nowait.cpp
```

**Run command:**

```
OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0 LIBOMPTARGET_DEBUG=1 ./a.out
```

Performance could 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.

```
/*
 * This test is taken from OpenMP API 5.0.1 Examples (June 2020)
 * https://www.openmp.org/wp-content/uploads/openmp-examples-5-0-1.pdf
 * (4.13.2 nowait Clause on target Construct)
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

#define N 100000 // N must be even

void init(int n, float *v1, float *v2) {
    int i;

    for(i=0; i<n; i++){
        v1[i] = i * 0.25;
        v2[i] = i - 1.25;
    }
}

int main() {
    int i, n=N;
    float v1[N],v2[N],vxv[N];
    double start,end; // timers

    init(n, v1,v2);

    /* Dummy parallel and target (nowait) regions, so as not to measure
       startup time. */
    #pragma omp parallel
    {
        #pragma omp master
        #pragma omp target nowait
        {};
    }

    start=omp_get_wtime();

    #pragma omp parallel
    {
        #pragma omp master
        #pragma omp target teams distribute parallel for nowait \
            map(to: v1[0:n/2]) \
            map(to: v2[0:n/2]) \
            map(from: vxv[0:n/2])
        for(i=0; i<n/2; i++){
            vxv[i] = v1[i]*v2[i];
        }
    }

    #pragma omp for
```

```

    for(i=n/2; i<n; i++) {
        vxv[i] = v1[i]*v2[i];
    }
    /* Implicit barrier at end of worksharing for. Target region is
       guaranteed to be completed by this point. */
}

end=omp_get_wtime();

printf("vxv[1]=%f, vxv[n-1]=%f, time=%lf\n", vxv[1], vxv[n-1], end-start);
return 0;
}

```

The performance of the two versions when running on one of our lab machines was as follows:

```

no nowait version      : 0.008220 seconds
nowait on target version : 0.002110 seconds

```

## Fortran

The same `nowait` example shown above may be written in Fortran as follows.

```

!
! This test is from OpenMP API 5.0.1 Examples (June 2020)
! https://www.openmp.org/wp-content/uploads/openmp-examples-5-0-1.pdf
!(4.13.2 nowait Clause on target Construct)
!

subroutine init(n, v1, v2)
integer :: i, n
real :: v1(n), v2(n)

do i = 1, n
    v1(i) = i * 0.25
    v2(i) = i - 1.25
end do
end subroutine init

program test_target_nowait
use omp_lib
use iso_fortran_env
implicit none

integer, parameter :: NUM=100000 ! NUM must be even
real :: v1(NUM), v2(NUM), vxv(NUM)
integer :: n, i
real(kind=REAL64) :: start, end

n = NUM
call init(n, v1, v2)

! Dummy parallel and target (nowait) regions, so as not to measure
! startup time.
 !$omp parallel
   !$omp master
     !$omp target nowait
     !$omp end target
   !$omp end master
 !$omp end parallel

```

```

start=omp_get_wtime()

 !$omp parallel

 !$omp master
 !$omp target teams distribute parallel do nowait &
 !$omp& map(to: v1(1:n/2)) &
 !$omp& map(to: v2(1:n/2)) &
 !$omp& map(from: vxv(1:n/2))
 do i = 1, n/2
   vxv(i) = v1(i)*v2(i)
 end do
 !$omp end master

 !$omp do
 do i = n/2+1, n
   vxv(i) = v1(i)*v2(i)
 end do

 !$omp end parallel

end=omp_get_wtime()

110 write(*,110) "vxv(1)=", vxv(1), ", vxv(n-1)=", vxv(n-1), ", time=", end-start
      format (A, F10.6, A, F17.6, A, F10.6)

end program test_target_nowait

```

## Memory Allocation

This section looks at various ways of allocating memory, and the types of allocations that are supported. A pointer on the host has the same size as a pointer on the device.

**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. Therefore, 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 Intel® Data Center GPU Max Series systems. However, shared allocations where memory is allocated by an allocation API are supported.

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 because 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, because 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 because accesses are to device-local memory.

## OpenMP Runtime Routines for Memory Allocation

Intel compilers support a number of OpenMP runtime routines for performing 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.

The following examples use the above OpenMP memory allocation routines. Compare those to the ones 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 SYCL part of this guide

## Using the `map` Clause

The first example 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 58 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.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <omp.h>

#define iterations 100
#define length      64*1024*1024

int main(void)
{
    size_t bytes = length*sizeof(double);
    double * __restrict A;
    double * __restrict B;
    double * __restrict C;
    double scalar = 3.0;
    double nstream_time = 0.0;

    // Allocate arrays on the host using plain malloc()

    A = (double *) malloc(bytes);
    if (A == NULL){
        printf(" ERROR: Cannot allocate space for A using plain malloc().\n");
        exit(1);
    }

    B = (double *) malloc(bytes);
    if (B == NULL){
        printf(" ERROR: Cannot allocate space for B using plain malloc().\n");
        exit(1);
    }

    C = (double *) malloc(bytes);
    if (C == NULL){
        printf(" ERROR: Cannot allocate space for C using plain malloc().\n");
        exit(1);
    }

    // Initialize the arrays

    #pragma omp parallel for
    for (size_t i=0; i<length; i++) {
        A[i] = 2.0;
        B[i] = 2.0;
        C[i] = 0.0;
    }

    // Perform the computation

    nstream_time = omp_get_wtime();
    for (int iter = 0; iter<iterations; iter++) {
        #pragma omp target teams distribute parallel for \
            map(to: A[0:length], B[0:length]) \
            map(tofrom: C[0:length])
        for (size_t i=0; i<length; i++) {
```

```

        C[i] += A[i] + scalar * B[i];
    }
}

nstream_time = omp_get_wtime() - nstream_time;

// Validate and output results

double ar = 2.0;
double br = 2.0;
double cr = 0.0;
for (int iter = 0; iter<iterations; iter++) {
    for (int i=0; i<length; i++) {
        cr += ar + scalar * br;
    }
}

double asum = 0.0;
#pragma omp parallel for reduction(+:asum)
for (size_t i=0; i<length; i++) {
    asum += fabs(C[i]);
}

free(A);
free(B);
free(C);

double epsilon=1.e-8;
if (fabs(cr - asum)/asum > epsilon) {
    printf("Failed Validation on output array\n"
          "      Expected checksum: %lf\n"
          "      Observed checksum: %lf\n"
          "ERROR: solution did not validate\n", cr, asum);
    return 1;
} else {
    printf("Solution validates\n");
    double avgtime = nstream_time/iterations;
    printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
}

return 0;
}

```

**Compilation command:**

```
icpx -fopenmp -fopenmp-targets=spir64 test_target_map.cpp
```

**Run command:**

```
OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0 LIBOMPTARGET_DEBUG=1 ./a.out
```

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.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <omp.h>

#define iterations 100
#define length      64*1024*1024

int main(void)
{
    size_t bytes = length*sizeof(double);
    double * __restrict A;
    double * __restrict B;
    double * __restrict C;
    double scalar = 3.0;
    double nstream_time = 0.0;

    // Allocate arrays on the host using plain malloc()

    A = (double *) malloc(bytes);
    if (A == NULL){
        printf(" ERROR: Cannot allocate space for A using plain malloc().\n");
        exit(1);
    }

    B = (double *) malloc(bytes);
    if (B == NULL){
        printf(" ERROR: Cannot allocate space for B using plain malloc().\n");
        exit(1);
    }

    C = (double *) malloc(bytes);
    if (C == NULL){
        printf(" ERROR: Cannot allocate space for C using plain malloc().\n");
        exit(1);
    }

    // Initialize the arrays

    #pragma omp parallel for
    for (size_t i=0; i<length; i++) {
        A[i] = 2.0;
        B[i] = 2.0;
        C[i] = 0.0;
    }

    // Perform the computation

    nstream_time = omp_get_wtime();
    #pragma omp target data map(to: A[0:length], B[0:length]) \
        map(tofrom: C[0:length])
    {
        for (int iter = 0; iter<iterations; iter++) {
```

```

#pragma omp target teams distribute parallel for
for (size_t i=0; i<length; i++) {
    C[i] += A[i] + scalar * B[i];
}
}
nstream_time = omp_get_wtime() - nstream_time;

// Validate and output results

double ar = 2.0;
double br = 2.0;
double cr = 0.0;
for (int iter = 0; iter<iterations; iter++) {
    for (int i=0; i<length; i++) {
        cr += ar + scalar * br;
    }
}

double asum = 0.0;
#pragma omp parallel for reduction(+:asum)
for (size_t i=0; i<length; i++) {
    asum += fabs(C[i]);
}

free(A);
free(B);
free(C);

double epsilon=1.e-8;
if (fabs(cr - asum)/asum > epsilon) {
    printf("Failed Validation on output array\n"
          "      Expected checksum: %lf\n"
          "      Observed checksum: %lf\n"
          "ERROR: solution did not validate\n", cr, asum);
    return 1;
} else {
    printf("Solution validates\n");
    double avgtime = nstream_time/iterations;
    printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
}

return 0;
}

```

### **omp\_target\_alloc**

Next, the example above is modified 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.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <omp.h>

#define iterations 100
#define length      64*1024*1024

int main(void)
{
    int device_id = omp_get_default_device();
    size_t bytes = length*sizeof(double);
    double * __restrict A;
    double * __restrict B;
    double * __restrict C;
    double scalar = 3.0;
    double nstream_time = 0.0;

    // Allocate arrays in device memory

    A = (double *) omp_target_alloc(bytes, device_id);
    if (A == NULL){
        printf(" ERROR: Cannot allocate space for A using omp_target_alloc().\n");
        exit(1);
    }

    B = (double *) omp_target_alloc(bytes, device_id);
    if (B == NULL){
        printf(" ERROR: Cannot allocate space for B using omp_target_alloc().\n");
        exit(1);
    }

    C = (double *) omp_target_alloc(bytes, device_id);
    if (C == NULL){
        printf(" ERROR: Cannot allocate space for C using omp_target_alloc().\n");
        exit(1);
    }

    // Initialize the arrays

    #pragma omp target teams distribute parallel for \
        is_device_ptr(A,B,C)
    for (size_t i=0; i<length; i++) {
        A[i] = 2.0;
        B[i] = 2.0;
        C[i] = 0.0;
    }

    // Perform the computation 'iterations' number of times
```

```

nstream_time = omp_get_wtime();
for (int iter = 0; iter<iterations; iter++) {
    #pragma omp target teams distribute parallel for \
        is_device_ptr(A,B,C)
    for (size_t i=0; i<length; i++) {
        C[i] += A[i] + scalar * B[i];
    }
}
nstream_time = omp_get_wtime() - nstream_time;

// Validate and output results

double ar = 2.0;
double br = 2.0;
double cr = 0.0;
for (int iter = 0; iter<iterations; iter++) {
    for (int i=0; i<length; i++) {
        cr += ar + scalar * br;
    }
}

double asum = 0.0;
#pragma omp target teams distribute parallel for reduction(+:asum) \
    map(tofrom: asum) is_device_ptr(C)
for (size_t i=0; i<length; i++) {
    asum += fabs(C[i]);
}

omp_target_free(A, device_id);
omp_target_free(B, device_id);
omp_target_free(C, device_id);

double epsilon=1.e-8;
if (fabs(cr - asum)/asum > epsilon) {
    printf("Failed Validation on output array\n"
          "      Expected checksum: %lf\n"
          "      Observed checksum: %lf\n"
          "ERROR: solution did not validate\n", cr, asum);
    return 1;
} else {
    printf("Solution validates\n");
    double avgtime = nstream_time/iterations;
    printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
}

return 0;
}

```

**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` with 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.

```
// Allocate arrays in device memory

A = (double *) omp_target_alloc_device(bytes, device_id);
if (A == NULL) {
    printf(" ERROR: Cannot allocate space for A using omp_target_alloc_device().\n");
    exit(1);
}

B = (double *) omp_target_alloc_device(bytes, device_id);
if (B == NULL) {
    printf(" ERROR: Cannot allocate space for B using omp_target_alloc_device().\n");
    exit(1);
}

C = (double *) omp_target_alloc_device(bytes, device_id);
if (C == NULL) {
    printf(" ERROR: Cannot allocate space for C using omp_target_alloc_device().\n");
    exit(1);
}
```

#### 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 the 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, 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, for portability the code includes the directive.

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.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <omp.h>

#pragma omp requires unified_address

#define iterations 100
#define length      64*1024*1024

int main(void)
{
    int device_id = omp_get_default_device();
    size_t bytes = length*sizeof(double);
    double * __restrict A;
    double * __restrict B;
    double * __restrict C;
    double scalar = 3.0;
    double nstream_time = 0.0;

    // Allocate arrays in host memory

    A = (double *) omp_target_alloc_host(bytes, device_id);
    if (A == NULL){
        printf(" ERROR: Cannot allocate space for A using omp_target_alloc_host().\n");
        exit(1);
    }

    B = (double *) omp_target_alloc_host(bytes, device_id);
    if (B == NULL){
        printf(" ERROR: Cannot allocate space for B using omp_target_alloc_host().\n");
        exit(1);
    }

    C = (double *) omp_target_alloc_host(bytes, device_id);
    if (C == NULL){
        printf(" ERROR: Cannot allocate space for C using omp_target_alloc_host().\n");
        exit(1);
    }

    // Initialize the arrays

    #pragma omp parallel for
    for (size_t i=0; i<length; i++) {
        A[i] = 2.0;
        B[i] = 2.0;
        C[i] = 0.0;
    }

    // Perform the computation

    nstream_time = omp_get_wtime();
    for (int iter = 0; iter<iterations; iter++) {
        #pragma omp target teams distribute parallel for
        for (size_t i=0; i<length; i++) {
```

```
        C[i] += A[i] + scalar * B[i];
    }
}

nstream_time = omp_get_wtime() - nstream_time;

// Validate and output results

double ar = 2.0;
double br = 2.0;
double cr = 0.0;
for (int iter = 0; iter<iterations; iter++) {
    for (int i=0; i<length; i++) {
        cr += ar + scalar * br;
    }
}

double asum = 0.0;
#pragma omp parallel for reduction(+:asum)
for (size_t i=0; i<length; i++) {
    asum += fabs(C[i]);
}

omp_target_free(A, device_id);
omp_target_free(B, device_id);
omp_target_free(C, device_id);

double epsilon=1.e-8;
if (fabs(cr - asum)/asum > epsilon) {
    printf("Failed Validation on output array\n"
          "      Expected checksum: %lf\n"
          "      Observed checksum: %lf\n"
          "ERROR: solution did not validate\n", cr, asum);
    return 1;
} else {
    printf("Solution validates\n");
    double avgtime = nstream_time/iterations;
    printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
}

return 0;
}
```

**Notes:**

- When calling `omp_target_alloc_host`, the device number specified must be one of the supported devices, 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).
- Intel® Data Center GPU Max Series does not support atomic operations (or algorithms that use atomic operations, such as some reductions) on host allocations (i.e., memory allocated via `omp_target_alloc_host`). Use atomic operations on memory allocated via `omp_target_alloc_device`, instead.

### **omp\_target\_alloc\_shared**

The above example is modified so that 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.

```
// Allocate arrays in shared memory

A = (double *) omp_target_alloc_shared(bytes, device_id);
if (A == NULL) {
    printf(" ERROR: Cannot allocate space for A using omp_target_alloc_shared().\n");
    exit(1);
}

B = (double *) omp_target_alloc_shared(bytes, device_id);
if (B == NULL) {
    printf(" ERROR: Cannot allocate space for B using omp_target_alloc_shared().\n");
    exit(1);
}

C = (double *) omp_target_alloc_shared(bytes, device_id);
if (C == NULL) {
    printf(" ERROR: Cannot allocate space for C using omp_target_alloc_shared().\n");
    exit(1);
}
```

#### **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).
- Concurrent access from host and device to memory allocated via `omp_target_alloc_shared` is not supported.

### **omp\_target\_memcpy**

The following example shows how the runtime routine `omp_target_memcpy` may be used to copy memory from host to device, and from device to host. 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 104, 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`.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <omp.h>

#define iterations 100
#define length      64*1024*1024

int main(void)
{
    int device_id = omp_get_default_device();
    int host_id = omp_get_initial_device();
    size_t bytes = length*sizeof(double);
    double * __restrict h_A;
    double * __restrict h_B;
    double * __restrict h_C;
    double * __restrict d_A;
    double * __restrict d_B;
    double * __restrict d_C;
    double scalar = 3.0;
    double nstream_time = 0.0;

    // Allocate arrays h_A, h_B, and h_C on the host using plain malloc()

    h_A = (double *) malloc(bytes);
    if (h_A == NULL){
        printf(" ERROR: Cannot allocate space for h_A using plain malloc().\n");
        exit(1);
    }

    h_B = (double *) malloc(bytes);
    if (h_B == NULL){
        printf(" ERROR: Cannot allocate space for h_B using plain malloc().\n");
        exit(1);
    }

    h_C = (double *) malloc(bytes);
    if (h_C == NULL){
        printf(" ERROR: Cannot allocate space for h_C using plain malloc().\n");
        exit(1);
    }

    // Allocate arrays d_A, d_B, and d_C on the device using omp_target_alloc()

    d_A = (double *) omp_target_alloc(bytes, device_id);
    if (d_A == NULL){
        printf(" ERROR: Cannot allocate space for d_A using omp_target_alloc().\n");
        exit(1);
    }

    d_B = (double *) omp_target_alloc(bytes, device_id);
    if (d_B == NULL){
        printf(" ERROR: Cannot allocate space for d_B using omp_target_alloc().\n");
    }
```

```
    exit(1);
}

d_C = (double *) omp_target_alloc(bytes, device_id);
if (d_C == NULL) {
    printf(" ERROR: Cannot allocate space for d_C using omp_target_alloc().\n");
    exit(1);
}

// Initialize the arrays on the host

#pragma omp parallel for
for (size_t i=0; i<length; i++) {
    h_A[i] = 2.0;
    h_B[i] = 2.0;
    h_C[i] = 0.0;
}

// Call omp_target_memcpy() to copy values from host to device

int rc = 0;
rc = omp_target_memcpy(d_A, h_A, bytes, 0, 0, device_id, host_id);
if (rc) {
    printf("ERROR: omp_target_memcpy(A) returned %d\n", rc);
    exit(1);
}

rc = omp_target_memcpy(d_B, h_B, bytes, 0, 0, device_id, host_id);
if (rc) {
    printf("ERROR: omp_target_memcpy(B) returned %d\n", rc);
    exit(1);
}

rc = omp_target_memcpy(d_C, h_C, bytes, 0, 0, device_id, host_id);
if (rc) {
    printf("ERROR: omp_target_memcpy(C) returned %d\n", rc);
    exit(1);
}

// Perform the computation

nstream_time = omp_get_wtime();
for (int iter = 0; iter<iterations; iter++) {
    #pragma omp target teams distribute parallel for \
        is_device_ptr(d_A,d_B,d_C)
    for (size_t i=0; i<length; i++) {
        d_C[i] += d_A[i] + scalar * d_B[i];
    }
}
nstream_time = omp_get_wtime() - nstream_time;

// Call omp_target_memcpy() to copy values from device to host

rc = omp_target_memcpy(h_C, d_C, bytes, 0, 0, host_id, device_id);
if (rc) {
    printf("ERROR: omp_target_memcpy(A) returned %d\n", rc);
    exit(1);
}
```

```

// Validate and output results

double ar = 2.0;
double br = 2.0;
double cr = 0.0;
for (int iter = 0; iter<iterations; iter++) {
    for (int i=0; i<length; i++) {
        cr += ar + scalar * br;
    }
}

double asum = 0.0;
#pragma omp parallel for reduction(+:asum)
for (size_t i=0; i<length; i++) {
    asum += fabs(h_C[i]);
}

free(h_A);
free(h_B);
free(h_C);
omp_target_free(d_A, device_id);
omp_target_free(d_B, device_id);
omp_target_free(d_C, device_id);

double epsilon=1.e-8;
if (fabs(cr - asum)/asum > epsilon) {
    printf("Failed Validation on output array\n"
          "      Expected checksum: %lf\n"
          "      Observed checksum: %lf\n"
          "ERROR: solution did not validate\n", cr, asum);
    return 1;
} else {
    printf("Solution validates\n");
    double avgtime = nstream_time/iterations;
    printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
}

return 0;
}

```

## 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 that 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` = 100). The average kernel time is computed by dividing the total time taken by the `iterations` loop by 100.

```

// Perform the computation 'iterations' number of times

nstream_time = omp_get_wtime();
for (int iter = 0; iter<iterations; iter++) {
    #pragma omp target teams distribute parallel for \
        is_device_ptr(A,B,C)
    for (size_t i=0; i<length; i++) {
        C[i] += A[i] + scalar * B[i];
}

```

```

    }
}

nstream_time = omp_get_wtime() - nstream_time;

```

LIBOMPTARGET\_DEBUG=1 output shows that all the above examples have the same ND\_range partitioning.

```

Target LEVEL0 RTL --> Allocated a device memory 0xff00000020200000
Libomptarget --> omp_target_alloc returns device ptr 0xff00000020200000
Libomptarget --> Call to omp_target_alloc for device 0 requesting 536870912 bytes
Libomptarget --> Call to omp_get_num_devices returning 1
Libomptarget --> Call to omp_get_initial_device returning 1
Libomptarget --> Checking whether device 0 is ready.
Libomptarget --> Is the device 0 (local ID 0) initialized? 1
Libomptarget --> Device 0 is ready to use.

```

The following table shows the average times taken by the kernel in the various versions when running on the particular GPU used (1-stack only).

Version	Time (seconds)
map	0.183604
map + target data	0.012757
omp_target_alloc	0.002501
omp_target_alloc_device	0.002499
omp_target_alloc_host	0.074412
omp_target_alloc_shared	0.012491
omp_target_memcpy	0.011072

The above performance numbers show that the `map` version is the slowest version (0.183604 seconds). This is because of the data transfers that occur at the beginning and end of each kernel launch. The main kernel is launched 100 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 0.012757 seconds, because the transfers occur once at the launch of the first kernel in the `iterations` loop, and again after the last kernel in that loop.

The `omp_target_alloc` and `omp_target_alloc_device` versions have the best performance (0.002501 and 0.002499 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. This is a useful model for applications that use scratch arrays on the device side. These arrays never need to be accessed on the host. In such cases, the recommendation is to allocate the scratch arrays on the device and not worry about data transfers, as illustrated in this example.

The `omp_target_alloc_shared` version also performs well, but is somewhat slower (0.012491 seconds). In this version, storage for `A`, `B`, and `C` is allocated in shared memory. So the data can migrate between the host and the device. There is the overhead of migration but, after migration, accesses on the device happen from much faster device-local memory. In this version, the initialization of the arrays happens on the host. At the first kernel launch, the arrays are migrated to the device, and the kernels access the arrays locally on the device. Finally, before the host performs the reduction computation, the entire `C` array is migrated back to the host.

The `omp_target_alloc_host` version (0.074412 seconds) takes almost 6x more time than the `omp_target_alloc_shared` version. This is because data allocated in host memory does not migrate from the host to the device. When the kernel tries to access the data, the data is typically sent over a bus, such as PCI Express, that connects the device to the host. This is slower than accessing local device memory. If the device accesses only a small part of an array infrequently, then that array may be allocated in host memory using `omp_target_alloc_host`. However, if the array is accessed frequently on the device side, then it should be kept in device memory. Keeping the data in host memory and accessing it over the PCI will degrade performance.

Finally, a note regarding data transfers: The amount of data transferred in the `map` version can be seen in `LIBOMPTARGET_DEBUG=1` output by grepping for "Libomptarget --> Moving". Notice that each launch of the main kernel yields the following data transfers:

```
$ grep "Libomptarget --> Moving" test_target_map.debug
Libomptarget --> Moving 536870912 bytes (hst:0x000007f1a5fc8b010) -> (tgt:0xff00000000200000)
Libomptarget --> Moving 536870912 bytes (hst:0x000007f1a9fc8d010) -> (tgt:0xff00000020200000)
Libomptarget --> Moving 536870912 bytes (hst:0x000007f1a7fc8c010) -> (tgt:0xff00000040200000)
Libomptarget --> Moving 536870912 bytes (tgt:0xff00000000200000) -> (hst:0x000007f1a5fc8b010)
```

On the other hand, data transfers in the `omp_target_alloc_...` versions are handled by a lower layer of the runtime system. So grepping for "Libomptarget --> Moving" in `LIBOMPTARGET_DEBUG=1` output for these versions will not show the data transfers that took place.

## Fortran Examples

The Fortran version of the example using `target` data and `map` clauses is shown below.

```
program main
use iso_fortran_env
use omp_lib
implicit none

integer, parameter :: iterations=100
integer, parameter :: length=64*1024*1024
real(kind=REAL64), parameter :: epsilon=1.D-8
real(kind=REAL64), allocatable :: A(:)
real(kind=REAL64), allocatable :: B(:)
real(kind=REAL64), allocatable :: C(:)
real(kind=REAL64) :: scalar=3.0
real(kind=REAL64) :: ar, br, cr, asum
real(kind=REAL64) :: nstream_time, avgtime
integer :: err, i, iter

!
! Allocate arrays on the host using plain allocate

allocate( A(length), stat=err )
if (err .ne. 0) then
    print *, "Allocation of A returned ", err
    stop 1
endif

allocate( B(length), stat=err )
if (err .ne. 0) then
    print *, "Allocation of B returned ", err
    stop 1
endif
```

```
allocate( C(length), stat=err )
if (err .ne. 0) then
    print *, "Allocation of C returned ", err
    stop 1
endif

!
! Initialize the arrays

 !$omp parallel do
do i = 1, length
    A(i) = 2.0
    B(i) = 2.0
    C(i) = 0.0
end do

!
! Perform the computation

nstream_time = omp_get_wtime()
 !$omp target data map(to: A, B) map(tofrom: C)

do iter = 1, iterations
    !$omp target teams distribute parallel do
    do i = 1, length
        C(i) = C(i) + A(i) + scalar * B(i)
    end do
end do

 !$omp end target data
nstream_time = omp_get_wtime() - nstream_time

!
! Validate and output results

ar = 2.0
br = 2.0
cr = 0.0
do iter = 1, iterations
    do i = 1, length
        cr = cr + ar + scalar * br
    end do
end do

asum = 0.0
 !$omp parallel do reduction(+:asum)
do i = 1, length
    asum = asum + abs(C(i))
end do

if (abs(cr - asum)/asum > epsilon) then
    write(*,110) "Failed Validation on output array: Expected =", cr, ", Observed =", asum
else
    avgtime = nstream_time/iterations
    write(*,120) "Solution validates: Checksum =", asum, ", Avg time (s) =", avgtime
endif

110 format (A, F20.6, A, F20.6)
```

```
120 format (A, F20.6, A, F10.6)

deallocate(A)
deallocate(B)
deallocate(C)

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 37) to indicate that the arrays have device addresses, and these addresses should be used in the target region.

```
use iso_fortran_env
use omp_lib
implicit none

integer, parameter :: iterations=100
integer, parameter :: length=64*1024*1024
real(kind=REAL64), parameter :: epsilon=1.D-8
real(kind=REAL64), allocatable :: A(:)
real(kind=REAL64), allocatable :: B(:)
real(kind=REAL64), allocatable :: C(:)
real(kind=REAL64) :: scalar=3.0
real(kind=REAL64) :: ar, br, cr, asum
real(kind=REAL64) :: nstream_time, avgtime
integer :: i, iter

!
! Allocate arrays in device memory

!$omp allocators allocate(allocator(omp_target_device_mem_alloc): A)
allocate(A(length))

!$omp allocators allocate(allocator(omp_target_device_mem_alloc): B)
allocate(B(length))

!$omp allocators allocate(allocator(omp_target_device_mem_alloc): C)
allocate(C(length))

!
! Begin target data

!$omp target data use_device_addr(A, B, C)

!
! Initialize the arrays

!$omp target teams distribute parallel do
do i = 1, length
    A(i) = 2.0
    B(i) = 2.0
    C(i) = 0.0
end do

!
! Perform the computation
```

```
nstream_time = omp_get_wtime()
do iter = 1, iterations
    !$omp target teams distribute parallel do
    do i = 1, length
        C(i) = C(i) + A(i) + scalar * B(i)
    end do
end do
nstream_time = omp_get_wtime() - nstream_time

!
! Validate and output results

ar = 2.0
br = 2.0
cr = 0.0
do iter = 1, iterations
    do i = 1, length
        cr = cr + ar + scalar * br
    end do
end do

asum = 0.0
!$omp target teams distribute parallel do reduction(+:asum) &
!$omp map(tofrom: asum)
do i = 1, length
    asum = asum + abs(C(i))
end do

!
! End target data

!$omp end target data

if (abs(cr - asum)/asum > epsilon) then
    write(*,110) "Failed Validation on output array: Expected =", cr, ", Observed =", asum
else
    avgtime = nstream_time/iterations
    write(*,120) "Solution validates: Checksum =", asum, ", Avg time (s) =", avgtime
endif

110 format (A, F20.6, A, F20.6)
120 format (A, F20.6, A, F10.6)

deallocate(A)
deallocate(B)
deallocate(C)

end program main
```

## Fortran Example

We will use a Fortran example to illustrate how choosing memory allocations appropriately can avoid redundant data transfers and boost performance.

The Fortran example uses OpenMP offload and is adapted from NWChem, a high-performance computational chemistry application.

In the example, there is a 4-level loop nest. The innermost k-loop calls the `omp_fbody` routine which offloads two calls to `sgemm` onto the device, and then offloads a reduction computation (reduction variables `emp4i` and `emp5i`) onto the device.

In the loop nest, some arrays are updated on the host (namely, arrays `Tkj`, `Kkj`, `Jia`, `Kia`, `Tia`, `Xia`, and `t1v1`). So we need to make the values of these arrays on the device consistent with their values on the host.

## Version 1

In the first (naive) version of the program, we allocate all 11 arrays in Shared memory using the `allocate` directive.

```
!$omp allocators allocate(allocator(omp_target_shared_mem_alloc): f1n)
allocate( f1n(1:nvir,1:nvir) )

!$omp allocators allocate(allocator(omp_target_shared_mem_alloc): f2n)
allocate( f2n(1:nvir,1:nvir) )

!$omp allocators allocate(allocator(omp_target_shared_mem_alloc): eorb)
allocate( eorb(1:nbf) )
...
```

Shared allocations are accessible by the host and the device, and automatically migrate between the host and the device as needed. So the same pointer to a Shared allocation may be used on the host and device.

Version 1 is shown below.

```
include "mkl_omp_offload.f90"

subroutine omp_fbody(f1n,f2n,eorb,
                     ncor,nocc,nvir, emp4,emp5,a,i,j,k,klo, &
                     Jia, Kia, Tia, Xia, Tkj, Kkj,           &
                     t1v1,t1v2)

use omp_lib
use onemkl_blas_omp_offload_lp64
use iso_fortran_env
implicit none

real, intent(inout) :: emp4,emp5
integer, intent(in) :: ncor,nocc,nvir
integer, intent(in) :: a,i,j,k, klo
real, intent(inout) :: f1n(nvir,nvir)
real, intent(inout) :: f2n(nvir,nvir)
real, intent(in)   :: eorb(*)
real, intent(in)   :: Jia(*), Kia(*), Tia(*), Xia(*)
real, intent(in)   :: Tkj(*), Kkj(*)
real, intent(in)   :: t1v1(nvir),t1v2(nvir)
real    :: emp4i,emp5i
real    :: eaijk,denom
integer :: lnov,lnvv
integer :: b,c
real    :: f1nbc,f1ncb,f2nbc,f2ncb
real    :: t1v1b,t1v2b

lnov=nocc*nvir
lnvv=nvir*nvir
emp4i = 0.0
emp5i = 0.0
```

```

 !$omp dispatch
 call sgemm('n','t',nvir,nvir,nvir,1.0,Jia,nvir,
            Tkj(1+(k-klo)*lnvv),nvir,1.0,f1n,nvir)      &

 !$omp dispatch
 call sgemm('n','t',nvir,nvir,nvir,1.0,Kia,nvir,
            Tkj(1+(k-klo)*lnvv),nvir,1.0, f2n,nvir)      &

 !$omp dispatch
 call sgemm('n','n',nvir,nvir,nocc,-1.0,Tia,nvir,
            Kkj(1+(k-klo)*lnov),nocc,1.0, f1n,nvir)      &

 !$omp dispatch
 call sgemm('n','n',nvir,nvir,nocc,-1.0,Xia,nvir,
            Kkj(1+(k-klo)*lnov),nocc,1.0, f2n,nvir)      &

 eaijk = eorb(a) - ( eorb(ncor+i)+eorb(ncor+j)+eorb(ncor+k) )

 !$omp target teams distribute parallel do collapse(2)      &
 !$omp reduction(:emp4i,emp5i)                                &
 !$omp private(f1nbc,f1ncb,f2nbc,f2ncb)                  &
 !$omp private(t1v1b,t1v2b)                                &
 !$omp private(denom) firstprivate(eaijk,nvir,ncor,nocc)
 do b=1,nvir
   do c=1,nvir
     denom=-1.0/( eorb(ncor+nocc+b)+eorb(ncor+nocc+c)+eaijk )

     f1nbc = f1n(b,c);
     f1ncb = f1n(c,b);
     f2nbc = f2n(b,c);
     f2ncb = f2n(c,b);
     t1v1b = t1v1(b);
     t1v2b = t1v2(b);

     emp4i = emp4i + (denom*t1v1b*f1nbc) + (denom*2*f1ncb)
     emp5i = emp5i + (denom*t1v2b*f2nbc) + (denom*3*f2ncb)
   enddo
 enddo
 !$omp end target teams distribute parallel do

 emp4 = emp4 + emp4i
 emp5 = emp5 + emp5i

 end

 subroutine init_array_1(arr, m)
 implicit none
 real, intent(inout) :: arr(m)
 integer m, i

 do i = 1, m
   arr(i) = 1.0/(100.0 + i-1)
 end do
 end subroutine init_array_1

 subroutine init_array_2(arr, m, n)

```

```
implicit none
real, intent(inout) :: arr(m, n)
integer m, n, i, j

do i = 1, m
    do j = 1, n
        arr(i,j) = 1.0/(100.0 + ((i-1) * n) + j)
    end do
end do
end subroutine init_array_2

program main

use omp_lib
use iso_fortran_env
implicit none

interface
    subroutine omp_fbody(f1n,f2n,eorb,
                         ncor,nocc,nvir, emp4,emp5,a,i,j,k,klo, &
                         Jia, Kia, Tia, Xia, Tkj, Kkj,             &
                         t1v1,t1v2)
        real, intent(inout) :: emp4,emp5
        integer, intent(in) :: ncor,nocc,nvir
        integer, intent(in) :: a,i,j,k, klo
        real, intent(inout) :: f1n(nvir,nvir)
        real, intent(inout) :: f2n(nvir,nvir)
        real, intent(in) :: eorb(*)
        real, intent(in) :: Jia(*), Kia(*), Tia(*), Xia(*), Tkj(*), Kkj(*)
        real, intent(in) :: t1v1(nvir),t1v2(nvir)
    end subroutine omp_fbody
end interface

integer :: ncor, nocc, nvir, maxiter, nkpass
integer :: nbff, lnnv, lnov, kchunk
real, allocatable :: eorb(:)
real, allocatable :: f1n(:,:)
real, allocatable :: f2n(:,:)

real, allocatable :: Jia(:)
real, allocatable :: Kia(:)
real, allocatable :: Tia(:)
real, allocatable :: Xia(:)
real, allocatable :: Tkj(:)
real, allocatable :: Kkj(:)

real, allocatable :: t1v1(:),t1v2(:)
real emp4, emp5
integer :: a, b, c, i, j, k
integer :: klo, khi, iter
double precision, allocatable :: timers(:)
double precision :: t0, t1, tsum, tmax, tmin, tavg

!
! Run parameters
nocc = 256
nvir = 2048
maxiter = 50
```

```
nkpass = 1
ncor = 0

print *, "Run parameters:"
print *, "nocc      =", nocc
print *, "nvir      =", nvir
print *, "maxiter   =", maxiter
print *, "nkpass    =", nkpass
print *, "ncor     =", ncor
print *, " "

!
! Allocate and initialize arrays.

nbf = ncor + nocc + nvir
lnvv = nvir * nvir
lnov = nocc * nvir
kchunk = (nocc - 1)/nkpass + 1

!$omp allocators allocate(allocator(omp_target_shared_mem_alloc): f1n)
allocate( f1n(1:nvir,1:nvir) )

!$omp allocators allocate(allocator(omp_target_shared_mem_alloc): f2n)
allocate( f2n(1:nvir,1:nvir) )

!$omp allocators allocate(allocator(omp_target_shared_mem_alloc): eorb)
allocate( eorb(1:nbf) )

!$omp allocators allocate(allocator(omp_target_shared_mem_alloc): Jia)
allocate( Jia(1:lnvv) )

!$omp allocators allocate(allocator(omp_target_shared_mem_alloc): Kia)
allocate( Kia(1:lnvv) )

!$omp allocators allocate(allocator(omp_target_shared_mem_alloc): Tia)
allocate( Tia(1:lnov*nocc) )

!$omp allocators allocate(allocator(omp_target_shared_mem_alloc): Xia)
allocate( Xia(1:lnov*nocc) )

!$omp allocators allocate(allocator(omp_target_shared_mem_alloc): Tkj)
allocate( Tkj(1:kchunk*lnvv) )

!$omp allocators allocate(allocator(omp_target_shared_mem_alloc): Kkj)
allocate( Kkj(1:kchunk*lnvv) )

!$omp allocators allocate(allocator(omp_target_shared_mem_alloc): t1v1)
allocate( t1v1(1:lnvv) )

!$omp allocators allocate(allocator(omp_target_shared_mem_alloc): t1v2)
allocate( t1v2(1:lnvv) )

!
call init_array_1(eorb, nbf)
call init_array_1(Jia, lnnv)
call init_array_1(Kia, lnnv)
call init_array_1(Tia, lnov*nocc)
call init_array_1(Xia, lnov*nocc)
call init_array_1(Tkj, kchunk*lnvv)
call init_array_1(Kkj, kchunk*lnov)
```

```

call init_array_1(t1v1, lnnv)
call init_array_1(t1v2, lnnv)

call init_array_2(f1n, nvir, nvir)
call init_array_2(f2n, nvir, nvir)

print *, "End of initialization"

allocate (timers(1:maxiter))

emp4=0.0
emp5=0.0
a=1
iter=1

do klo = 1, nocc, kchunk
    khi = MIN(nocc, klo+kchunk-1)
    do j = 1, nocc

#if defined(DO_UPDATE_ARRAYS)
!
!           Update elements of Tkj and KKj.
        Tkj((khi-klo+1)*lnvv) = Tkj((khi-klo+1)*lnvv) + 1.0
        Kkj((khi-klo+1)*lnov) = Kkj((khi-klo+1)*lnov) + 1.0
#endif

        do i = 1, nocc

#if defined(DO_UPDATE_ARRAYS)
!
!           Update elements of Jia, Kia, Tia, Xia arrays.
        Jia(lnvv) = Jia(lnvv) + 1.0
        Kia(lnvv) = Kia(lnvv) + 1.0
        Tia(lnov) = Tia(lnov) + 1.0
        Xia(lnov) = Xia(lnov) + 1.0
#endif

        do k = klo, MIN(khi,i)

#if defined(DO_UPDATE_ARRAYS)
!
!           Update elements of t1v1 array.
        t1v1(:) = Tkj(lnvv-nvir+1:lnvv)
#endif

        t0 = omp_get_wtime()

        call omp_fbody(f1n,f2n,eorb,
                      &
                      ncor,nocc,nvir, emp4,emp5,a,i,j,k,klo, &
                      Jia, Kia, Tia, Xia, Tkj, Kkj,          &
                      t1v1,t1v2)

        t1 = omp_get_wtime()
        timers(iter) = (t1-t0)
        if (iter .eq. maxiter) then
            print *, "Stopping after ", iter, "iterations"
            print *, " "
            goto 150
        endif

!
!           Prevent NAN for large maxiter...

```

```
        if (emp4 > 1000.0) then
            emp4 = emp4 - 1000.0
        endif
        if (emp4 < -1000.0) then
            emp4 = emp4 + 1000.0
        endif
        if (emp5 > 1000.0) then
            emp5 = emp5 - 1000.0
        endif
        if (emp5 < -1000.0) then
            emp5 = emp5 + 1000.0
        endif

        iter = iter + 1

    end do ! k = klo, MIN(khi,i)
    end do ! do i = 1, nocc
    end do ! do j = 1, nocc
end do ! do klo = 1, nocc, kchunk

150    CONTINUE

tsum = 0.0
tmax = -1.0e10
tmin = 1.0e10
do i = 2, iter
    tsum = tsum + timers(i)
    tmax = MAX(tmax,timers(i))
    tmin = MIN(tmin,timers(i))
end do

tavg = tsum / (iter - 1)
print *, "TOTAL ITER: ", iter
write(*, 110) " TIMING: min=", tmin, ", max=", tmax, ", avg of iters after first=",
tavg, " seconds"
110    format (A, F9.6, A, F9.6, A, F9.6, A)

write(*, 120) " emp4 = ", emp4, " emp5 =", emp5
120    format (A, F15.3, A, F15.3)

print *, "END"

deallocate (f1n)
deallocate (f2n)
deallocate (eorb)
deallocate (Jia)
deallocate (Kia)
deallocate (Tia)
deallocate (Xia)
deallocate (Tkj)
deallocate (Kkj)

deallocate (t1v1)
deallocate (t1v2)
deallocate (timers)

end program
```

While this version is straightforward to program, it is not the most efficient.

## Version 2

In Version 2 of the program, we allocate all 11 arrays in system memory using plain `allocate`, and use the `map` clause to map the arrays to the device:

```
real, allocatable :: f1n(:,:)
real, allocatable :: f2n(:,:)
real, allocatable :: eorb(:)
...
 !$omp target data
    map(to: f1n, f2n, eorb)      &
    map(to: Jia, Kia, Tia, Xia)  &
    map(to: Tkj, Kkj)           &
    map(to: t1v1, t1v2)
```

When an array that is mapped to the device is updated on the host, we use the OpenMP `target update` directive to copy the new values of the array from the host to the device. We place the directive at the highest level in the loop nest as possible, to avoid unnecessary copying of the array.

```
Tkj((khi-klo+1)*lnvv) = Tkj((khi-klo+1)*lnvv) + 1.0
Kkj((khi-klo+1)*lnov) = Kkj((khi-klo+1)*lnov) + 1.0
 !$omp target update to (Tkj, Kkj)
```

The full Version 2 is shown below.

```
include "mkl_omp_offload.f90"

subroutine omp_fbody(f1n,f2n,eorb,
                     ncor,nocc,nvir, emp4,emp5,a,i,j,k,klo,      &
                     Jia, Kia, Tia, Xia, Tkj, Kkj,             &
                     t1v1,t1v2)

use omp_lib
use onemkl_blas_omp_offload_lp64
use iso_fortran_env
implicit none

real, intent(inout) :: emp4,emp5
integer, intent(in) :: ncor,nocc,nvir
integer, intent(in) :: a,i,j,k, klo
real, intent(inout) :: f1n(nvir,nvir)
real, intent(inout) :: f2n(nvir,nvir)
real, intent(in)   :: eorb(*)
real, intent(in)   :: Jia(*), Kia(*), Tia(*), Xia(*)
real, intent(in)   :: Tkj(*), Kkj(*)
real, intent(in)   :: t1v1(nvir),t1v2(nvir)
real   :: emp4i,emp5i
real   :: eaijk,denom
integer :: lnov,lnvv
integer :: b,c
real   :: f1nbc,f1ncb,f2nbc,f2ncb
real   :: t1v1b,t1v2b

lnov=nocc*nvir
lnvv=nvir*nvir
emp4i = 0.0
```

```

emp5i = 0.0

!$omp dispatch
call sgemm('n','t',nvir,nvir,nvir,1.0,Jia,nvir,      &
           Tkj(1+(k-klo)*lnvv),nvir,1.0,f1n,nvir)

!$omp dispatch
call sgemm('n','t',nvir,nvir,nvir,1.0,Kia,nvir,      &
           Tkj(1+(k-klo)*lnvv),nvir,1.0, f2n,nvir)

!$omp dispatch
call sgemm('n','n',nvir,nvir,nocc,-1.0,Tia,nvir,      &
           Kkj(1+(k-klo)*lnov),nocc,1.0, f1n,nvir)

!$omp dispatch
call sgemm('n','n',nvir,nvir,nocc,-1.0,Xia,nvir,      &
           Kkj(1+(k-klo)*lnov),nocc,1.0, f2n,nvir)

eaijk = eorb(a) - ( eorb(ncor+i)+eorb(ncor+j)+eorb(ncor+k) )

!$omp target teams distribute parallel do collapse(2)      &
!$omp      reduction(:emp4i,emp5i)                         &
!$omp      private(f1nbc,f1ncb,f2nbc,f2ncb)             &
!$omp      private(t1v1b,t1v2b)                           &
!$omp      private(denom) firstprivate(eaijk,nvir,ncor,nocc)
do b=1,nvir
  do c=1,nvir
    denom=-1.0/( eorb(ncor+nocc+b)+eorb(ncor+nocc+c)+eaijk )

    f1nbc = f1n(b,c);
    f1ncb = f1n(c,b);
    f2nbc = f2n(b,c);
    f2ncb = f2n(c,b);
    t1v1b = t1v1(b);
    t1v2b = t1v2(b);

    emp4i = emp4i + (denom*t1v1b*f1nbc) + (denom*2*f1ncb)
    emp5i = emp5i + (denom*t1v2b*f2nbc) + (denom*3*f2ncb)
  enddo
enddo
!$omp end target teams distribute parallel do

emp4 = emp4 + emp4i
emp5 = emp5 + emp5i

end

subroutine init_array_1(arr, m)
  implicit none
  real, intent(inout) :: arr(m)
  integer m, i

  do i = 1, m
    arr(i) = 1.0/(100.0 + i-1)
  end do
end subroutine init_array_1

```

```
subroutine init_array_2(arr, m, n)
    implicit none
    real, intent(inout) :: arr(m, n)
    integer m, n, i, j

    do i = 1, m
        do j = 1, n
            arr(i,j) = 1.0/(100.0 + ((i-1) * n) + j)
        end do
    end do
end subroutine init_array_2

program main

use omp_lib
use iso_fortran_env
implicit none

interface
    subroutine omp_fbody(f1n,f2n,eorb,
                         &ncor,nocc,nvir, emp4,emp5,a,i,j,k,klo, &
                         &Jia, Kia, Tia, Xia, Tkj, Kkj,          &
                         &t1v1,t1v2)
        real, intent(inout) :: emp4,emp5
        integer, intent(in) :: ncor,nocc,nvir
        integer, intent(in) :: a,i,j,k, klo
        real, intent(inout) :: f1n(nvir,nvir)
        real, intent(inout) :: f2n(nvir,nvir)
        real, intent(in)   :: eorb(*)
        real, intent(in)   :: Jia(*), Kia(*), Tia(*), Xia(*), Tkj(*), Kkj(*)
        real, intent(in)   :: t1v1(nvir),t1v2(nvir)
    end subroutine omp_fbody
end interface

integer :: ncor, nocc, nvir, maxiter, nkpass
integer :: nbf, lnnv, lnov, kchunk
real, allocatable :: eorb(:)
real, allocatable :: f1n(:, :)
real, allocatable :: f2n(:, :)

real, allocatable :: Jia(:)
real, allocatable :: Kia(:)
real, allocatable :: Tia(:)
real, allocatable :: Xia(:)
real, allocatable :: Tkj(:)
real, allocatable :: Kkj(:)

real, allocatable :: t1v1(:, ),t1v2(:, )
real emp4, emp5
integer :: a, b, c, i, j, k
integer :: klo, khi, iter
double precision, allocatable :: timers(:)
double precision :: t0, t1, tsum, tmax, tmin, tavg

!
! Run parameters
nocc = 256
```

```
nvir = 2048
maxiter = 50
nkpass = 1
ncor = 0

print *, "Run parameters:"
print *, "nocc      =", nocc
print *, "nvir      =", nvir
print *, "maxiter   =", maxiter
print *, "nkpass    =", nkpass
print *, "ncor      =", ncor
print *, ""

!
! Allocate and initialize arrays.

nbf = ncor + nocc + nvir
lnvv = nvir * nvir
lnov = nocc * nvir
kchunk = (nocc - 1)/nkpass + 1

allocate( f1n(1:nvir,1:nvir) )
allocate( f2n(1:nvir,1:nvir) )
allocate( eorb(1:nbf) )
allocate( Jia(1:lnvv) )
allocate( Kia(1:lnvv) )
allocate( Tia(1:lnov*nocc) )
allocate( Xia(1:lnov*nocc) )
allocate( Tkj(1:kchunk*lnvv) )
allocate( Kkj(1:kchunk*lnvv) )
allocate( t1v1(1:lnvv) )
allocate( t1v2(1:lnvv) )

!
call init_array_1(eorb, nbf)
call init_array_1(Jia, lnnv)
call init_array_1(Kia, lnnv)
call init_array_1(Tia, lnov*nocc)
call init_array_1(Xia, lnov*nocc)
call init_array_1(Tkj, kchunk*lnvv)
call init_array_1(Kkj, kchunk*lnov)
call init_array_1(t1v1, lnnv)
call init_array_1(t1v2, lnnv)

call init_array_2(f1n, nvir, nvir)
call init_array_2(f2n, nvir, nvir)

print *, "End of initialization"

allocate (timers(1:maxiter))

emp4=0.0
emp5=0.0
a=1
iter=1

 !$omp target data
 map(to: f1n, f2n, eorb) &
 map(to: Jia, Kia, Tia, Xia) &
 map(to: Tkj, Kkj) &
```

```

map(to: t1v1, t1v2)

do klo = 1, nocc, kchunk
    khi = MIN(nocc, klo+kchunk-1)
    do j = 1, nocc

#if defined(DO_UPDATE_ARRAYS)
!           Update elements of Tkj and KKj.
Tkj((khi-klo+1)*lnvv) = Tkj((khi-klo+1)*lnvv) + 1.0
Kkj((khi-klo+1)*lnov) = Kkj((khi-klo+1)*lnov) + 1.0

 !$omp target update to (Tkj, Kkj)
#endif

do i = 1, nocc

#if defined(DO_UPDATE_ARRAYS)
!           Update elements of Jia, Kia, Tia, Xia arrays.
Jia(lnvv) = Jia(lnvv) + 1.0
Kia(lnvv) = Kia(lnvv) + 1.0
Tia(lnov) = Tia(lnov) + 1.0
Xia(lnov) = Xia(lnov) + 1.0

 !$omp target update to (Jia, Kia, Tia, Xia)
#endif

do k = klo, MIN(khi,i)

#if defined(DO_UPDATE_ARRAYS)
!           Update elements of t1v1 array.
t1v1(:) = Tkj(lnvv-nvir+1:lnvv)

 !$omp target update to (t1v1)
#endif

t0 = omp_get_wtime()

call omp_fbody(f1n,f2n,eorb,
               &
               ncor,nocc,nvir, emp4,emp5,a,i,j,k,klo, &
               Jia, Kia, Tia, Xia, Tkj, Kkj,          &
               t1v1,t1v2)

t1 = omp_get_wtime()
timers(iter) = (t1-t0)
if (iter .eq. maxiter) then
    print *, "Stopping after ", iter, "iterations"
    print *, " "
    goto 150
endif

!
! Prevent NAN for large maxiter...
if (emp4 > 1000.0) then
    emp4 = emp4 - 1000.0
endif
if (emp4 < -1000.0) then
    emp4 = emp4 + 1000.0
endif
if (emp5 > 1000.0) then

```

```

        emp5 = emp5 - 1000.0
    endif
    if (emp5 < -1000.0) then
        emp5 = emp5 + 1000.0
    endif

        iter = iter + 1

    end do ! k = klo, MIN(khi,i)
    end do ! do i = 1, nocc
    end do ! do j = 1, nocc
end do ! do klo = 1, nocc, kchunk

150    CONTINUE
!$omp end target data

tsum = 0.0
tmax = -1.0e10
tmin = 1.0e10
do i = 2, iter
    tsum = tsum + timers(i)
    tmax = MAX(tmax,timers(i))
    tmin = MIN(tmin,timers(i))
end do

tavg = tsum / (iter - 1)
print *, "TOTAL ITER: ", iter
write(*, 110) " TIMING: min=", tmin, ", max=", tmax, ", avg of iters after first=",
tavg, " seconds"
110    format (A, F9.6, A, F9.6, A, F9.6, A)

        write(*, 120) " emp4 = ", emp4, " emp5 =", emp5
120    format (A, F15.3, A, F15.3)

        print *, "END"

deallocate (fln)
deallocate (f2n)
deallocate (eorb)
deallocate (Jia)
deallocate (Kia)
deallocate (Tia)
deallocate (Xia)
deallocate (Tkj)
deallocate (Kkj)

deallocate (t1v1)
deallocate (t1v2)
deallocate (timers)

end program

```

### Version 3

In the third version of the program, we consider which arrays are used on the host only, on the device only, or on both the host and the device. We also consider whether the array values are updated during the execution of the program. This information is used to decide where to allocate the arrays, how to initialize them, and whether to update their values on the device.

Arrays `f1` and `f2` are used as work arrays on the device (to store the results of calls to SGEMM), and they are not accessed on the host. So we allocate them directly on the device.

Since `f1` and `f2` are allocated on the device, we call `init_array_2()` to initialize the arrays in an OpenMP target construct.

The other 9 arrays are accessed on both the host and the device, so we allocate them in Host memory and call `init_array_1()` to initialize the arrays. The arrays are then mapped to the device using the `map` clause.

We use the OpenMP target update to directive to copy the updated values of `Tkj`, `Kkj`, `Jia`, `Kia`, `Tia`, `Xia`, and `t1v1` from the host to the device.

```
!$omp allocators allocate(allocator(omp_target_device_mem_alloc): f1n)
allocate( f1n(1:nvir,1:nvir) )

!$omp allocators allocate(allocator(omp_target_device_mem_alloc): f2n)
allocate( f2n(1:nvir,1:nvir) )

!$omp allocators allocate(allocator(omp_target_host_mem_alloc): eorb)
allocate( eorb(1:nbf) )

!$omp allocators allocate(allocator(omp_target_host_mem_alloc): Jia)
allocate( Jia(1:lnvv) )

!$omp allocators allocate(allocator(omp_target_host_mem_alloc): Kia)
allocate( Kia(1:lnvv) )
...

!$omp target data
    map(to: eorb) &
    map(to: Jia, Kia, Tia, Xia) &
    map(to: Tkj, Kkj) &
    map(to: t1v1, t1v2)
    ...

Tkj((khi-klo+1)*lnvv) = Tkj((khi-klo+1)*lnvv) + 1.0
Kkj((khi-klo+1)*lnov) = Kkj((khi-klo+1)*lnov) + 1.0

!$omp target update to (Tkj, Kkj)
```

The full Version 3 of the program is shown below.

```
include "mkl_omp_offload.f90"

subroutine omp_fbody(f1n,f2n,eorb,
                     ncor,nocc,nvir, emp4,emp5,a,i,j,k,klo, &
                     Jia, Kia, Tia, Xia, Tkj, Kkj,           &
                     t1v1,t1v2)

use omp_lib
use onemkl_blas_omp_offload_lp64
use iso_fortran_env
implicit none

real, intent(inout) :: emp4,emp5
integer, intent(in) :: ncor,nocc,nvir
integer, intent(in) :: a,i,j,k, klo
real, intent(inout) :: f1n(nvir,nvir)
real, intent(inout) :: f2n(nvir,nvir)
```

```

real, intent(in)    :: eorb(*)
real, intent(in)    :: Jia(*), Kia(*), Tia(*), Xia(*)
real, intent(in)    :: Tkj(*), Kkj(*)
real, intent(in)    :: t1v1(nvir),t1v2(nvir)
real    :: emp4i,emp5i
real    :: eaijk,denom
integer :: lnov,lnvv
integer :: b,c
real    :: f1nbc,f1ncb,f2nbc,f2ncb
real    :: t1v1b,t1v2b

lnov=nocc*nvir
lnvv=nvir*nvir
emp4i = 0.0
emp5i = 0.0

 !$omp dispatch
call sgemm('n','t',nvir,nvir,nvir,1.0,Jia,nvir,      &
           Tkj(1+(k-klo)*lnvv),nvir,1.0,f1n,nvir)

 !$omp dispatch
call sgemm('n','t',nvir,nvir,nvir,1.0,Kia,nvir,      &
           Tkj(1+(k-klo)*lnvv),nvir,1.0, f2n,nvir)

 !$omp dispatch
call sgemm('n','n',nvir,nvir,nocc,-1.0,Tia,nvir,      &
           Kkj(1+(k-klo)*lnov),nocc,1.0, f1n,nvir)

 !$omp dispatch
call sgemm('n','n',nvir,nvir,nocc,-1.0,Xia,nvir,      &
           Kkj(1+(k-klo)*lnov),nocc,1.0, f2n,nvir)

eaijk = eorb(a) - ( eorb(ncor+i)+eorb(ncor+j)+eorb(ncor+k) )

 !$omp target teams distribute parallel do collapse(2)      &
 !$omp reduction(+:emp4i,emp5i)                                &
 !$omp private(f1nbc,f1ncb,f2nbc,f2ncb)                      &
 !$omp private(t1v1b,t1v2b)                                     &
 !$omp private(denom) firstprivate(eaijk,nvir,ncor,nocc)
do b=1,nvir
  do c=1,nvir
    denom=-1.0/( eorb(ncor+nocc+b)+eorb(ncor+nocc+c)+eaijk )

    f1nbc = f1n(b,c);
    f1ncb = f1n(c,b);
    f2nbc = f2n(b,c);
    f2ncb = f2n(c,b);
    t1v1b = t1v1(b);
    t1v2b = t1v2(b);

    emp4i = emp4i + (denom*t1v1b*f1nbc) + (denom*2*f1ncb)
    emp5i = emp5i + (denom*t1v2b*f2nbc) + (denom*3*f2ncb)
  enddo
enddo
 !$omp end target teams distribute parallel do

emp4 = emp4 + emp4i
emp5 = emp5 + emp5i

```

```
end

subroutine init_array_1(arr, m)
    implicit none
    real, intent(inout) :: arr(m)
    integer m, i

    do i = 1, m
        arr(i) = 1.0/(100.0 + i-1)
    end do
end subroutine init_array_1

subroutine init_array_2(arr, m, n)
    implicit none
    real, intent(inout) :: arr(m, n)
    integer m, n, i, j

    !$omp target teams distribute parallel do
    do i = 1, m
        do j = 1, n
            arr(i,j) = 1.0/(100.0 + ((i-1) * n) + j)
        end do
    end do
end subroutine init_array_2

program main

use omp_lib
use iso_fortran_env
implicit none

interface
    subroutine omp_fbody(f1n,f2n,eorb,
                         ncor,nocc,nvir, emp4,emp5,a,i,j,k,klo, &
                         Jia, Kia, Tia, Xia, Tkj, Kkj, &
                         t1v1,t1v2)
        real, intent(inout) :: emp4,emp5
        integer, intent(in) :: ncor,nocc,nvir
        integer, intent(in) :: a,i,j,k, klo
        real, intent(inout) :: f1n(nvir,nvir)
        real, intent(inout) :: f2n(nvir,nvir)
        real, intent(in) :: eorb(*)
        real, intent(in) :: Jia(*), Kia(*), Tia(*), Xia(*), Tkj(*), Kkj(*)
        real, intent(in) :: t1v1(nvir),t1v2(nvir)
    end subroutine omp_fbody
end interface

integer :: ncor, nocc, nvir, maxiter, nkpass
integer :: nbf, lnnv, lnov, kchunk
real, allocatable :: eorb(:)
real, allocatable :: f1n(:, :)
real, allocatable :: f2n(:, :)

real, allocatable :: Jia(:)
```

```
real, allocatable :: Kia(:)
real, allocatable :: Tia(:)
real, allocatable :: Xia(:)
real, allocatable :: Tkj(:)
real, allocatable :: Kkj(:)

real, allocatable :: t1v1(:),t1v2(:)
real emp4, emp5
integer :: a, b, c, i, j, k
integer :: klo, khi, iter
double precision, allocatable :: timers(:)
double precision :: t0, t1, tsum, tmax, tmin, tavg

! Run parameters
nocc = 256
nvir = 2048
maxiter = 50
nkpass = 1
ncor = 0

print *, "Run parameters:"
print *, "nocc      =", nocc
print *, "nvir      =", nvir
print *, "maxiter   =", maxiter
print *, "nkpass    =", nkpass
print *, "ncor      =", ncor
print *, " "

! Allocate and initialize arrays.

nbf = ncor + nocc + nvir
lnvv = nvir * nvir
lnov = nocc * nvir
kchunk = (nocc - 1)/nkpass + 1

!$omp allocators allocate(allocator(omp_target_device_mem_alloc): f1n)
allocate( f1n(1:nvir,1:nvir) )

!$omp allocators allocate(allocator(omp_target_device_mem_alloc): f2n)
allocate( f2n(1:nvir,1:nvir) )

!$omp allocators allocate(allocator(omp_target_host_mem_alloc): eorb)
allocate( eorb(1:nbf) )

!$omp allocators allocate(allocator(omp_target_host_mem_alloc): Jia)
allocate( Jia(1:lnvv) )

!$omp allocators allocate(allocator(omp_target_host_mem_alloc): Kia)
allocate( Kia(1:lnvv) )

!$omp allocators allocate(allocator(omp_target_host_mem_alloc): Tia)
allocate( Tia(1:lnov*nocc) )

!$omp allocators allocate(allocator(omp_target_host_mem_alloc): Xia)
allocate( Xia(1:lnov*nocc) )

!$omp allocators allocate(allocator(omp_target_host_mem_alloc): Tkj)
allocate( Tkj(1:kchunk*lnvv) )
```

```

 !$omp allocators allocate(allocator(omp_target_host_mem_alloc): Kkj)
 allocate( Kkj(1:kchunk*lnvv) )

 !$omp allocators allocate(allocator(omp_target_host_mem_alloc): t1v1)
 allocate( t1v1(1:lnvv) )

 !$omp allocators allocate(allocator(omp_target_host_mem_alloc): t1v2)
 allocate( t1v2(1:lnvv) )
!

call init_array_1(eorb, nbf)
call init_array_1(Jia, lnvv)
call init_array_1(Kia, lnvv)
call init_array_1(Tia, lnov*nocc)
call init_array_1(Xia, lnov*nocc)
call init_array_1(Tkj, kchunk*lnvv)
call init_array_1(Kkj, kchunk*lnov)
call init_array_1(t1v1, lnvv)
call init_array_1(t1v2, lnvv)

call init_array_2(f1n, nvir, nvir)
call init_array_2(f2n, nvir, nvir)

print *, "End of initialization"

allocate (timers(1:maxiter))

emp4=0.0
emp5=0.0
a=1
iter=1

 !$omp target data
    map(to: eorb) &
    map(to: Jia, Kia, Tia, Xia) &
    map(to: Tkj, Kkj) &
    map(to: t1v1, t1v2)

do klo = 1, nocc, kchunk
    khi = MIN(nocc, klo+kchunk-1)
    do j = 1, nocc

#if defined(DO_UPDATE_ARRAYS)
!
    Update elements of Tkj and KKj.
    Tkj((khi-klo+1)*lnvv) = Tkj((khi-klo+1)*lnvv) + 1.0
    Kkj((khi-klo+1)*lnov) = Kkj((khi-klo+1)*lnov) + 1.0

    !$omp target update to (Tkj, Kkj)
#endif

        do i = 1, nocc

#if defined(DO_UPDATE_ARRAYS)
!
    Update elements of Jia, Kia, Tia, Xia arrays.
    Jia(lnvv) = Jia(lnvv) + 1.0
    Kia(lnvv) = Kia(lnvv) + 1.0
    Tia(lnov) = Tia(lnov) + 1.0
    Xia(lnov) = Xia(lnov) + 1.0

```

```

 !$omp target update to (Jia, Kia, Tia, Xia)
#endif

      do k = klo, MIN(khi,i)

#if defined(DO_UPDATE_ARRAYS)
!
!           Update elements of t1v1 array.
t1v1(:) = Tkj(lnvv-nvir+1:lnvv)

!$omp target update to (t1v1)
#endif

      t0 = omp_get_wtime()

      call omp_fbody(f1n,f2n,eorb,
                     &
                     ncor,nocc,nvir, emp4,emp5,a,i,j,k,klo, &
                     Jia, Kia, Tia, Xia, Tkj, Kkj,             &
                     t1v1,t1v2)

      t1 = omp_get_wtime()
      timers(iter) = (t1-t0)
      if (iter .eq. maxiter) then
          print *, "Stopping after ", iter, "iterations"
          print *, " "
          goto 150
      endif

!
! Prevent NAN for large maxiter...
if (emp4 > 1000.0) then
    emp4 = emp4 - 1000.0
endif
if (emp4 < -1000.0) then
    emp4 = emp4 + 1000.0
endif
if (emp5 > 1000.0) then
    emp5 = emp5 - 1000.0
endif
if (emp5 < -1000.0) then
    emp5 = emp5 + 1000.0
endif

      iter = iter + 1

      end do ! k = klo, MIN(khi,i)
      end do ! do i = 1, nocc
      end do ! do j = 1, nocc
      end do ! do klo = 1, nocc, kchunk

150    CONTINUE
 !$omp end target data

      tsum = 0.0
      tmax = -1.0e10
      tmin = 1.0e10
      do i = 2, iter
          tsum = tsum + timers(i)
          tmax = MAX(tmax,timers(i))

```

```

        tmin = MIN(tmin,timers(i))
end do

tavg = tsum / (iter - 1)
print *, "TOTAL ITER: ", iter
write(*, 110) " TIMING: min=", tmin, ", max=", tmax, ", avg of iters after first=",
tavg, " seconds"
110  format (A, F9.6, A, F9.6, A, F9.6, A)
      write(*, 120) " emp4 = ", emp4, " emp5 =", emp5
120  format (A, F15.3, A, F15.3)

print *, "END"

deallocate (f1n)
deallocate (f2n)
deallocate (eorb)
deallocate (Jia)
deallocate (Kia)
deallocate (Tia)
deallocate (Xia)
deallocate (Tkj)
deallocate (Kkj)

deallocate (t1v1)
deallocate (t1v2)
deallocate (timers)

end program

```

## Performance Comparison

The following commands were used to compile, link, and run the various versions. In the following, substitute the source filename for TEST.f.

```

Compile:
ifx -O3 -fiopenmp -fopenmp-targets=spir64 -DMKL -qmkl -fpp -free -D DO_UPDATE_ARRAYS TEST.f -c -
o TEST.o

Link:
ifx -fiopenmp -fopenmp-targets=spir64 -qmkl -fsycl -L${MKLROOT}/lib/intel64 \
-liomp5 -lsycl -lOpenCL -lstdc++ -lpthread -lm -ldl -lmkl_sycl \
-D DO_UPDATE_ARRAYS TEST.o -o TEST.exe

Run:
LIBOMPTARGET_LEVEL_ZERO_COMMAND_BATCH=copy OMP_TARGET_OFFLOAD=MANDATORY \
ZE_AFFINITY_MASK=0 LIBOMPTARGET_DEBUG=0 LIBOMPTARGET_PLUGIN=level0 TEST.exe

```

We compared the performance of the three versions on the particular GPU used (1-stack only).

The average iteration time for the various versions was as follows.

```

Version 1 (test-SharedMem.f): 0.115163 seconds
Version 2 (test-Map-UpdateTo.f): 0.002012 seconds
Version 3 (test-DeviceMem-Map-UpdateTo.f): 0.001978 seconds

```

## Clauses: `is_device_ptr`, `use_device_ptr`, `has_device_addr`, `use_device_addr`

The OpenMP clauses `is_device_ptr`, `use_device_ptr`, `has_device_addr`, and `use_device_addr` can be used to convey information about variables referenced in `target`, `target data`, or `dispatch` constructs. These clauses are described as follows.

### `is_device_ptr`

The `is_device_ptr` clause appears on a `target` or `dispatch` directive. It indicates that the list items are device pointers. So each list item is privatized inside the construct and the new list item is initialized to the device address to which the original list item refers.

- In C, each list item should be of type pointer or array.
- In C++, each list item should be of type pointer, array, reference to pointer, or reference to array.
- In Fortran, each list item should be of type `C_PTR`.

The following C/C++ example illustrates the use of the `is_device_ptr` clause. The `omp_target_alloc_device` routine allocates memory on the device and returns a device pointer for that memory which is saved in the host variable `arr_device`. On the `target` directive, we use the `is_device_ptr(arr_device)` clause to indicate that `arr_device` points to device memory. So inside the `target` construct `arr_device` is privatized and initialized to the device address to which `arr_device` refers.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <omp.h>

#define N 100

int main(void)
{
    int *arr_host = NULL;
    int *arr_device = NULL;

    arr_host = (int *) malloc(N * sizeof(int));
    arr_device = (int *) omp_target_alloc_device(N * sizeof(int),
                                                omp_get_default_device());

#pragma omp target is_device_ptr(arr_device) map(from: arr_host[0:N])
{
    for (int i = 0; i < N; ++i) {
        arr_device[i] = i;
        arr_host[i] = arr_device[i];
    }
}

printf ("%d, %d, %d \n", arr_host[0], arr_host[N/2], arr_host[N-1]);
}
```

### `use_device_ptr`

The `use_device_ptr` clause appears on a `target data` directive. It indicates that each list item is a pointer to an object that has corresponding storage on the device or is accessible on the device.

If a list item is a pointer to an object that is mapped to the device, then references to the list item in the construct are converted to references to a device pointer that is local to the construct and that refers to the device address of the corresponding object.

If the list item does not point to a mapped object, it must contain a valid device address, and the list item references are converted to references to a local device pointer that refers to this device address.

Each list item must be a pointer for which the value is the address of an object that has corresponding storage in the device data environment or is accessible on the target device.

In C, each list item should be of type pointer or array.

In C++, each list item should be of type pointer, array, reference to pointer, or reference to array.

In Fortran, each list item should be of type C\_PTR.

The following C/C++ example illustrates the use of the `use_device_ptr` clause. The `omp_target_alloc_device` routine is called three times to allocate memory on the device. The addresses of the memory allocated is saved in the pointer variables A, B, and C on the host. We use the `use_device_ptr(A, B, C)` clause on the `target data` directive to indicate that A, B, and C contain valid device addresses.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <omp.h>

#define length 65536

int main(void)
{
    int device_id = omp_get_default_device();
    size_t bytes = length*sizeof(double);
    double * __restrict A;
    double * __restrict B;
    double * __restrict C;
    double scalar = 3.0;
    double ar;
    double br;
    double cr;
    double asum;

    // Allocate arrays in device memory

    A = (double *) omp_target_alloc_device(bytes, device_id);
    if (A == NULL){
        printf(" ERROR: Cannot allocate space for A using omp_target_alloc_device().\n");
        exit(1);
    }

    B = (double *) omp_target_alloc_device(bytes, device_id);
    if (B == NULL){
        printf(" ERROR: Cannot allocate space for B using omp_target_alloc_device().\n");
        exit(1);
    }

    C = (double *) omp_target_alloc_device(bytes, device_id);
    if (C == NULL){
        printf(" ERROR: Cannot allocate space for C using omp_target_alloc_device().\n");
        exit(1);
    }

    #pragma omp target data use_device_ptr(A,B,C)
    {
        // Initialize the arrays
```

```
#pragma omp target teams distribute parallel for
for (size_t i=0; i<length; i++) {
    A[i] = 2.0;
    B[i] = 2.0;
    C[i] = 0.0;
}

// Perform the computation

#pragma omp target teams distribute parallel for
for (size_t i=0; i<length; i++) {
    C[i] += A[i] + scalar * B[i];
}

// Validate and output results

ar = 2.0;
br = 2.0;
cr = 0.0;
for (int i=0; i<length; i++) {
    cr += ar + scalar * br;
}

asum = 0.0;
#pragma omp target teams distribute parallel for reduction(+:asum)
for (size_t i=0; i<length; i++) {
    asum += fabs(C[i]);
}

} // end target data

omp_target_free(A, device_id);
omp_target_free(B, device_id);
omp_target_free(C, device_id);

double epsilon=1.e-8;
if (fabs(cr - asum)/asum > epsilon) {
    printf("Failed Validation on output array\n"
          "      Expected checksum: %lf\n"
          "      Observed checksum: %lf\n"
          "ERROR: solution did not validate\n", cr, asum);
    return 1;
} else {
    printf("Solution validates. Checksum = %lf\n", asum);
}

return 0;
}
```

#### **has\_device\_addr**

The `has_device_addr` clause appears on a `target` directive. It indicates that the list items already have valid device addresses, and therefore may be directly accessed from the device.

Each list item must have a valid device address for the device data environment. It can be on any type, including an array section.

The `has_device_addr` clause is especially useful in Fortran, because it can be used with list items of any type (not just `C_PTR`) to indicate that the list items have device addresses.

The following Fortran example illustrates the use of the `has_device_addr` clause. In the example, the three arrays `A`, `B`, and `C` are allocated on the device. When the arrays are referenced in a `target` region, we use the `has_device_addr(A, B, C)` clause to indicate that `A`, `B`, and `C` already have device addresses.

```
program main
    use iso_fortran_env
    use omp_lib
    implicit none

    integer, parameter :: iterations=1000
    integer, parameter :: length=64*1024*1024
    real(kind=REAL64), parameter :: epsilon=1.D-8
    real(kind=REAL64), allocatable :: A(:)
    real(kind=REAL64), allocatable :: B(:)
    real(kind=REAL64), allocatable :: C(:)
    real(kind=REAL64) :: scalar=3.0
    real(kind=REAL64) :: ar, br, cr, asum
    real(kind=REAL64) :: nstream_time, avgtime
    integer :: i, iter

    !
    ! Allocate arrays in device memory

    !$omp allocators allocate(allocator(omp_target_device_mem_alloc): A)
    allocate(A(length))

    !$omp allocators allocate(allocator(omp_target_device_mem_alloc): B)
    allocate(B(length))

    !$omp allocators allocate(allocator(omp_target_device_mem_alloc): C)
    allocate(C(length))

    !
    ! Initialize the arrays

    !$omp target teams distribute parallel do has_device_addr(A, B, C)
    do i = 1, length
        A(i) = 2.0
        B(i) = 2.0
        C(i) = 0.0
    end do

    !
    ! Perform the computation

    nstream_time = omp_get_wtime()
    do iter = 1, iterations
        !$omp target teams distribute parallel do has_device_addr(A, B, C)
        do i = 1, length
            C(i) = C(i) + A(i) + scalar * B(i)
        end do
    end do
    nstream_time = omp_get_wtime() - nstream_time

    !
    ! Validate and output results
```

```

ar = 2.0
br = 2.0
cr = 0.0
do iter = 1, iterations
    do i = 1, length
        cr = cr + ar + scalar * br
    end do
end do

asum = 0.0
!$omp target teams distribute parallel do reduction(+:asum) has_device_addr(C)
do i = 1, length
    asum = asum + abs(C(i))
end do

if (abs(cr - asum)/asum > epsilon) then
    print *, "Failed Validation on output array:", "Expected =", cr, "Observed =", asum
else
    avgtime = nstream_time/iterations
    print *, "Solution validates:", "Checksum =", asum, "Avg time (s) =", avgtime
endif

deallocate(A)
deallocate(B)
deallocate(C)

end program main

```

### **use\_device\_addr**

The `use_device_addr` clause appears on a `target data` directive. It indicates that each list item already has corresponding storage on the device or is accessible on the device.

If a list item is mapped, then references to the list item in the construct are converted to references to the corresponding list item. If a list item is not mapped, it is assumed to be accessible on the device.

A list item may be an array section.

Just like `has_device_addr`, the `use_device_addr` clause is especially useful in Fortran, because it can be used with list items of any type (not just `C_PTR`) to indicate that the list items have device addresses.

The following Fortran example illustrates the use of the `use_device_addr` clause. In the example, `array_d` is mapped to the device with the `alloc` map-type, so storage is allocated for `array_d` on the device and no data transfer between the host and the device occurs. We use the `use_device_addr(array_d)` clause on the `target data` directive to indicate that `array_d` has corresponding storage on the device.

```

program target_use_device_addr

use omp_lib
use iso_fortran_env, only : real64
implicit none

integer, parameter :: N1 = 1024
real(kind=real64), parameter :: aval = real(42, real64)
real(kind=real64), allocatable :: array_d(:), array_h(:)
integer :: i,err

! Allocate host data

```

```

allocate(array_h(N1))

!$omp target data map (from:array_h(1:N1)) map(alloc:array_d(1:N1))
!$omp target data use_device_addr(array_d)
 !$omp target
    do i=1, N1
        array_d(i) = aval
        array_h(i) = array_d(i)
    end do
 !$omp end target
 !$omp end target data
 !$omp end target data

 ! Check result
 write (*,*) array_h(1), array_h(N1)
 if (any(array_h /= aval)) then
     err = 1
 else
     err = 0
 end if

 deallocate(array_h)
 if (err == 1) then
     stop 1
 else
     stop 0
 end if

end program target_use_device_addr

```

The following table summarizes the properties of the clauses described in this section.

Clause	On which directive	Type of list item	Description
is_device_ptr	target, dispatch	C/C++: Pointer, array, or reference Fortran: C_PTR	Indicates that list item is a device pointer (has valid device address).
use_device_ptr	target data	C/C++: Pointer, array, or reference Fortran: C_PTR	Indicates that list item is a pointer to an object that has corresponding storage on device or is accessible on device.
has_device_addr	target	Any type (may be array section)	Indicates that list item has a valid device address.
use_device_addr	target data	Any type (may be array section)	Indicates that list item has corresponding storage on device or is accessible on the device.

## Prefetching

User-guided data prefetching is a useful technique for hiding latency arising from lower-level cache misses and main memory accesses. OpenMP offload for Intel® GPUs now enables this feature using the `prefetch` pragma, with syntax as follows:

### C OpenMP prefetch:

```
#pragma ompx prefetch data([prefetch-hint-modifier:],arrsect, [,arrsect] ) [ if
(condition) ]
```

### Fortran OpenMP prefetch:

```
!$omp prefetch data( [prefetch-hint-modifier:] arrsect [, arrsect] ) [if (condition)]
```

The prefetch pragma above is an Intel® extension, and works for Intel® Data Center GPU Max Series and later products. The main aspects of the pragma are:

- *Prefetch-hint*: The destination for the prefetched data is specified using the optional `prefetch-hint-modifier`. Valid values are 0 (No-op), 2 (prefetch to L2 only) and 4 (prefetch to L1 and L2). If the value is not specified, the default value is 0.
- *Array section*: A contiguous array section `arrsect` is specified using the OpenMP syntax `[lower-bound : length]` in C, and `(lower-bound : upper-bound)` in Fortran. For example, to prefetch four elements starting at `a[10]` or `a(10)`, use `a[10:4]` in C or `a(10:13)` in Fortran. Note, at the time of this writing, the compiler generates only single element prefetch requests. The examples above are therefore equivalent to `a[10:1]` in C, or `a(10:10)` in Fortran.
- *Default prefetch size*: Even if a single array element is requested to be prefetched, the hardware will prefetch an entire cache line that contains that element. In Intel® Data Center GPU Max Series, the size of a cache line is 64 bytes.
- *Faulting*: Prefetch instructions in Intel® Data Center GPU Max Series are faulting, which means accesses to invalid addresses can cause a segmentation fault. The optional `if` condition in the pragma can be used for guarding against out-of-bounds accesses.
- *Non-blocking*: The prefetch pragma does not block, it does not wait for the prefetch to complete.

### Prefetch in C OpenMP

The following example shows a simplified 1-dimension version of an N-body force kernel. The outer for-loop iterates over the particles for which the forces are calculated. The inner loops iterate over the interacting particles, in batches of `TILE_SIZE` particles. We can prefetch the next stack of particles during the computations of the current stack of particles. Prefetch always brings in 64 bytes of data as described above. So we need to prefetch only one out of every 16 single-precision floating point values, which is achieved by using `if ( (next_tile % 16) == 0 )`. Using this masking condition may not always help, see additional notes after the code snippet below. The prefetch hint used is 4 (prefetch to L1 and L2 cache). Only the offloaded kernel is shown below.

```
#define WORKGROUP_SIZE 1024
#define PREFETCH_HINT 4 // 4 = prefetch to L1 and L3; 2 = prefetch to L3
#define TILE_SIZE 64

void nbody_1d_gpu(float *c, float *a, float *b, int n1, int n2) {
#pragma omp target teams distribute parallel for thread_limit(WORKGROUP_SIZE)
    for (int i = 0; i < n1; i++) {
        const float ma0 = 0.269327f, ma1 = -0.0750978f, ma2 = 0.0114808f;
        const float ma3 = -0.00109313f, ma4 = 0.0000605491f, ma5 = -0.00000147177f;
        const float eps = 0.01f;

        float dx = 0.0;
        float bb[TILE_SIZE];
        for (int j = 0; j < n2; j += TILE_SIZE) {
            // load tile from b
            for (int u = 0; u < TILE_SIZE; ++u) {
                bb[u] = b[j + u];
            }
#endif PREFETCH
            int next_tile = j + TILE_SIZE + u;
            if ((next_tile % 16) == 0) {
#pragma ompx prefetch data(PREFETCH_HINT : b[next_tile]) if (next_tile < n2)
            }
#endif
        }
    }
}
```

```
#pragma unroll(TILE_SIZE)
    for (int u = 0; u < TILE_SIZE; ++u) {
        float delta = bb[u] - a[i];
        float r2 = delta * delta;
        float s0 = r2 + eps;
        float s1 = 1.0f / sqrtf(s0);
        float f =
            (s1 * s1 * s1) -
            (ma0 + r2 * (ma1 + r2 * (ma2 + r2 * (ma3 + r2 * (ma4 + ma5))))) ;
        dx += f * delta;
    }
}
c[i] = dx * 0.23f;
}
}
```

The condition `if ( (next_tile % 16) == 0 )` can save on the prefetch overhead when the array index is not vectorized. In the example above, only the index `i` is vectorized, so when we prefetch `b[]` that is indexed using `j`, it helps to issue a prefetch only once every 16 elements. On the other hand, if we were to prefetch an array over the index `i`, then the prefetch is vectorized and therefore the masking condition may not offer any benefits. The user will need to experimentally determine the best approach for their application.

### Compilation command:

Without prefetch:

```
icpx -O3 -g -fiopenmp -fopenmp-targets=spir64 -mcmodel=medium nbody_c.cpp -o test_c
```

With prefetch:

```
icpx -O3 -g -fiopenmp -fopenmp-targets=spir64 -mcmodel=medium -DPREFETCH nbody_c.cpp -o test_c
```

### Run command:

```
LIBOMPTARGET_LEVEL0_COMPILATION_OPTIONS="-cl-strict-aliasing -cl-fast-relaxed-math" \
ZE_AFFINITY_MASK=0 LIBOMPTARGET_PLUGIN_PROFILE=T,usec IGC_ForceOCLSIMDWidth=16 ./test_c
```

The default SIMD width is chosen to be 16 or 32 automatically by the backend device compiler (Intel® Graphics Compiler or IGC) on Intel® Data Center GPU Max Series by compiler heuristics that take into account factors such as register pressure in the kernel. One can use the IGC environment variable `IGC_ForceOCLSIMDWidth=16` to request the IGC compiler to force a SIMD width of 16. SIMD16 gave a better performance for the above kernel. In the run command, we have also enabled OpenMP's built-in profiler using `LIBOMPTARGET_PLUGIN_PROFILE=T,usec`. The output from the run without prefetch was as follows below.

```
Obtained output = 222700231.430
Expected output = 222700339.016

Total time = 205.4 milliseconds
=====
=====
LIBOMPTARGET_PLUGIN_PROFILE(LEVEL0) for OMP DEVICE(0) Intel(R) Graphics [0x0bd6], Thread 0
-----
-----
Kernel 0 : __omp_offloading_46_3c0d785c_z12nbody_1d_gpuPfs_S_ii_115
Kernel 1 : __omp_offloading_46_3c0d785c_z15clean_cache_gpuPdi_169
Kernel 2 : __omp_offloading_46_3c0d785c_z4main_198
-----
-----
: Host Time (usec) Device Time (usec)
```

Name		:	Total	Average	Min	Max	Total	Average
Min	Max	Count						
<hr/>								
Compiling		:	598283.05	598283.05	598283.05	598283.05	0.00	0.00
0.00	0.00	1.00						
DataAlloc		:	9578.23	798.19	0.00	8728.03	0.00	0.00
0.00	0.00	12.00						
DataRead (Device to Host)	:	77.01	77.01	77.01	77.01	5.68	5.68	
5.68	5.68	1.00						
DataWrite (Host to Device)	:	713.11	356.55	179.05	534.06	15.76	7.88	
5.04	10.72	2.00						
Kernel 0		:	205292.22	2052.92	2033.95	2089.98	203572.32	2035.72
1984.96	2073.12	100.00						
Kernel 1		:	109194.28	1091.94	1076.94	1681.09	107051.52	1070.52
1062.40	1107.04	100.00						
Kernel 2		:	1746.89	1746.89	1746.89	1746.89	3.84	3.84
3.84	3.84	1.00						
Linking		:	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	1.00						
OffloadEntriesInit		:	2647.88	2647.88	2647.88	2647.88	0.00	0.00
0.00	0.00	1.00						
<hr/>								
<hr/>								

From the output above, the average device time for the GPU kernel execution (Kernel 0) is 2036 microseconds. If we run the binary with prefetch enabled, the average kernel device time observed was 1841 microseconds, as shown below:

Name		:	Host Time (usec)		Device Time (usec)				
Min	Max	Count	:	Total	Average	Min	Max	Total	Average
<hr/>									
Compiling		:	499351.98	499351.98	499351.98	499351.98	0.00	0.00	
0.00	0.00	1.00							
DataAlloc		:	9609.94	800.83	0.00	8740.19	0.00	0.00	
0.00	0.00	12.00							
DataRead (Device to Host)	:	77.01	77.01	77.01	77.01	4.96	4.96		
4.96	4.96	1.00							
DataWrite (Host to Device)	:	722.17	361.08	185.01	537.16	16.40	8.20		
5.44	10.96	2.00							
Kernel 0		:	185793.88	1857.94	1839.88	1919.03	184075.20	1840.75	
1824.00	1874.56	100.00							
<hr/>									
<hr/>									

Kernel 1			:	109442.95	1094.43	1076.94	1590.01	107334.56	1073.35
1062.40	1115.68	100.00							
Kernel 2			:	1821.99	1821.99	1821.99	1821.99	3.84	3.84
3.84	3.84	1.00							
Linking			:	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	1.00							
OffloadEntriesInit			:	2493.14	2493.14	2493.14	2493.14	0.00	0.00
0.00	0.00	1.00							
<hr/>									
<hr/>									

Please note that the achieved performance depends on the hardware and the software stack used, so users may see different performance numbers at their end.

## Prefetch in Fortran OpenMP

The same nbody1d kernel is shown in Fortran below. The prefetch pragma is inserted in the same location as before, with prefetch hint of value 4, and again prefetching only one out of every 16 elements.

```
#define WORKGROUP_SIZE 1024
#define PREFETCH_HINT 4      ! 4 = prefetch to L1 and L3;  2 = prefetch to L3
#define TILE_SIZE 64

    module gpu_kernels
    contains
        subroutine nbody_1d_gpu(c, a, b, n1, n2)
        implicit none
        integer n1, n2
        real a(0:n1-1), b(0:n2-1), c(0:n1-1)
        real dx, bb(0:TILE_SIZE-1), delta, r2, s0, s1, f
        integer i,j,u,next
        real ma0, ma1, ma2, ma3, ma4, ma5, eps
        parameter (ma0=0.269327, ma1=-0.0750978, ma2=0.0114808)
        parameter (ma3=-0.00109313, ma4=0.0000605491, ma5=-0.00000147177)
        parameter (eps=0.01)

!$omp target teams distribute parallel do thread_limit(WORKGROUP_SIZE)
!$omp& private(i,dx,j,u,bb,next,delta,r2,s0,s1,f)
        do i = 0, n1-1
            dx = 0.0
            do j = 0, n2-1, TILE_SIZE
                ! load tile from b
                do u = 0, TILE_SIZE-1
                    bb(u) = b(j+u)
                #ifdef PREFETCH
                    next = j + TILE_SIZE + u
                    if (mod(next,16).eq.0) then
!$omp prefetch data(PREFETCH_HINT:b(next:next))if(next<n2)
                        endif
                #endif
                enddo
                ! compute
                !DIR$ unroll(TILE_SIZE)
                do u = 0, TILE_SIZE-1
                    delta = bb(u) - a(i)
                    r2 = delta*delta
                    s0 = r2 + eps
                    s1 = 1.0 / sqrt(s0)
                    f = (s1*s1*s1)-(ma0+r2*(ma1+r2*(ma2+r2*(ma3+r2*(ma4+ma5)))))
```

```

        dx = dx + f*delta
    enddo
enddo
c(i) = dx*0.23
enddo
end subroutine

```

**Compilation command:****Without prefetch:**

```
ifx -O3 -g -fopenmp -fopenmp-targets=spir64 -fpconstant -fpp -ffast-math \
-fno-sycl-instrument-device-code -mcmmodel=medium nbody_f.f -o test_f
```

**With prefetch:**

```
ifx -O3 -g -fopenmp -fopenmp-targets=spir64 -fpconstant -fpp -ffast-math \
-fno-sycl-instrument-device-code -mcmmodel=medium -DPREFETCH nbody_f.f -o test_f
```

**Run command:**

```
LIBOMPTARGET_LEVEL0_COMPILATION_OPTIONS="-cl-strict-aliasing -cl-fast-relaxed-math" \
ZE_AFFINITY_MASK=0 LIBOMPTARGET_PLUGIN_PROFILE=T,usec IGC_ForceOCLSIMDWidth=16 ./test_f
```

The output is not shown here since it looks like the output of the C example. The average kernel time without and with prefetch were respectively, 2017 us and 1823 us. Again, please note that users may see different performance numbers, depending on the actual hardware and the software stack used.

**Prefetch in C OpenMP SIMD**

OpenMP offload also supports a SIMD programming model wherein all computations are specified in terms of EU threads that comprise 16 or 32 SIMD lanes in Intel® Data Center GPU Max Series. Correspondingly, even the `thread_limit()` clause in OpenMP takes on a modified meaning, and now specifies the number of EU threads per work-group. The OpenMP SIMD version of the `nbody1d` kernel is listed below. We need to explicitly specify the SIMD width, which is `VECLEN=16`. At the time of this writing, the prefetch pragma is recommended to be used outside the scope of the `simd` clause, which means only one SIMD lane will issue a prefetch instruction. In this example, 1 out of 16 lanes will carry out prefetch, which is exactly what we need – so we no longer need `if ( (next_tile % 16) == 0 )` that we had used in the previous examples above.

```

#define WORKGROUP_SIZE 1024
#define PREFETCH_HINT 4 // 4 = prefetch to L1 and L3; 2 = prefetch to L3
#define TILE_SIZE 64

void nbody_1d_gpu(float *c, float *a, float *b, int n1, int n2) {
#pragma omp target teams distribute parallel for thread_limit(WORKGROUP_SIZE / \
VECLEN)
    for (int i = 0; i < n1; i += VECLEN) {
        const float ma0 = 0.269327f, ma1 = -0.0750978f, ma2 = 0.0114808f;
        const float ma3 = -0.00109313f, ma4 = 0.0000605491f, ma5 = -0.00000147177f;
        const float eps = 0.01f;

        float dx[VECLEN];
        float aa[VECLEN], bb[TILE_SIZE];
#pragma omp simd simdlen(VECLEN)
#pragma unroll(0)
        for (int v = 0; v < VECLEN; ++v) {
            dx[v] = 0.0f;
            aa[v] = a[i + v];
        }
    }
}

```

```

for (int j = 0; j < n2; j += TILE_SIZE) {
    // load tile from b
    for (int u = 0; u < TILE_SIZE; u += VECLEN) {
#pragma omp simd simdlen(VECLEN)
#pragma unroll(0)
        for (int v = 0; v < VECLEN; ++v)
            bb[u + v] = b[j + u + v];
#endif PREFETCH
        int next_tile = j + TILE_SIZE + u;
#pragma ompx prefetch data(PREFETCH_HINT : b[next_tile]) if (next_tile < n2)
#endif
    }
    // compute current tile
#pragma omp simd simdlen(VECLEN)
#pragma unroll(0)
    for (int v = 0; v < VECLEN; ++v) {
#pragma unroll(TILE_SIZE)
        for (int u = 0; u < TILE_SIZE; ++u) {
            float delta = bb[u] - aa[v];
            float r2 = delta * delta;
            float s0 = r2 + eps;
            float s1 = 1.0f / sqrtf(s0);
            float f =
                (s1 * s1 * s1) -
                (ma0 + r2 * (ma1 + r2 * (ma2 + r2 * (ma3 + r2 * (ma4 + ma5))))) +
            dx[v] += f * delta;
        }
    }
#endif VECLEN
#pragma unroll(0)
    for (int v = 0; v < VECLEN; ++v) {
        c[i + v] = dx[v] * 0.23f;
    }
}

```

## **Compilation command:**

We need to use an additional compilation switch `-fopenmp-target-simd` to enable the SIMD programming model. The compilation command is therefore as follows:

## Without prefetch:

```
icpx -O3 -g -fopenmp -fopenmp-targets=spir64 -mcmmodel=medium \
-fopenmp-target-simd nbody c simd.cpp -o test c simd
```

With prefetch:

```
icpx -O3 -g -fopenmp -fopenmp-targets=spir64 -mcmodel=medium -DPREFETCH \
-fopenmp-target-simd nbody c simd.cpp -o test c simd
```

## Run command:

```
LIBOMPTARGET_LEVEL0_COMPILATION_OPTIONS="-fstrict-aliasing -ffast-relaxed-math" \
    ZE_AFFINITY_MASK=0 LIBOMPTARGET_PLUGIN PROFILE=T,usec ./test c simd
```

Notice that we no longer need the environment variable `IGC_ForceOCL SIMDWidth=16`, because the SIMD width has been explicitly specified in the OpenMP code.

The output looks like the previous examples, so it is not shown. The average kernel time without and with prefetch are respectively, 2008 us and 1810 us. As noted earlier, users may see different performance numbers, depending on the actual hardware and the software stack used.

## Multi-GPU and Multi-Stack Architecture and Programming

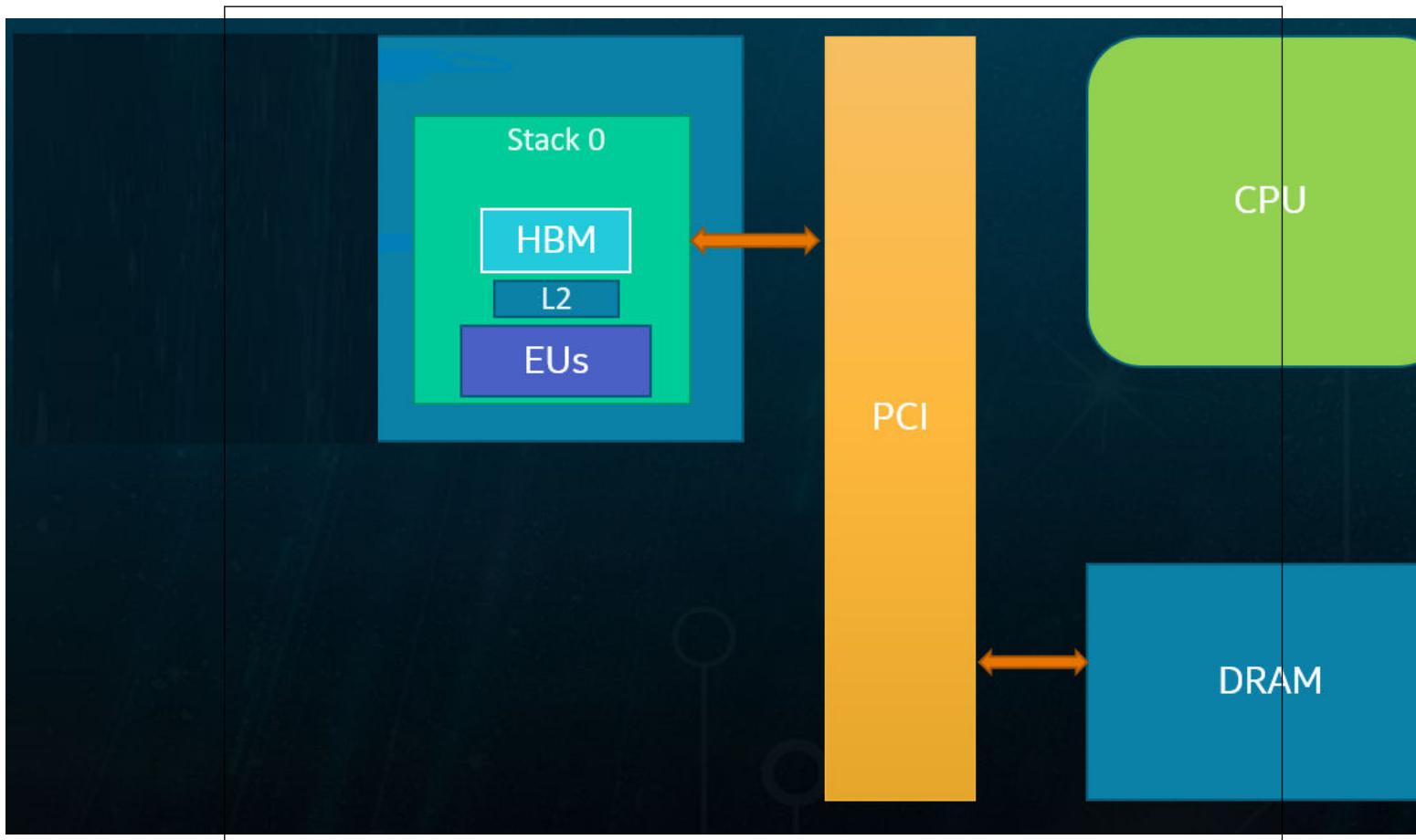
Intel® Data Center GPU Max Series uses a multi-stack GPU architecture, where each GPU contains 1 or 2 stacks. The GPU architecture and products enable multi-GPU and multi-stack computing.

In this chapter, we introduce the following topics:

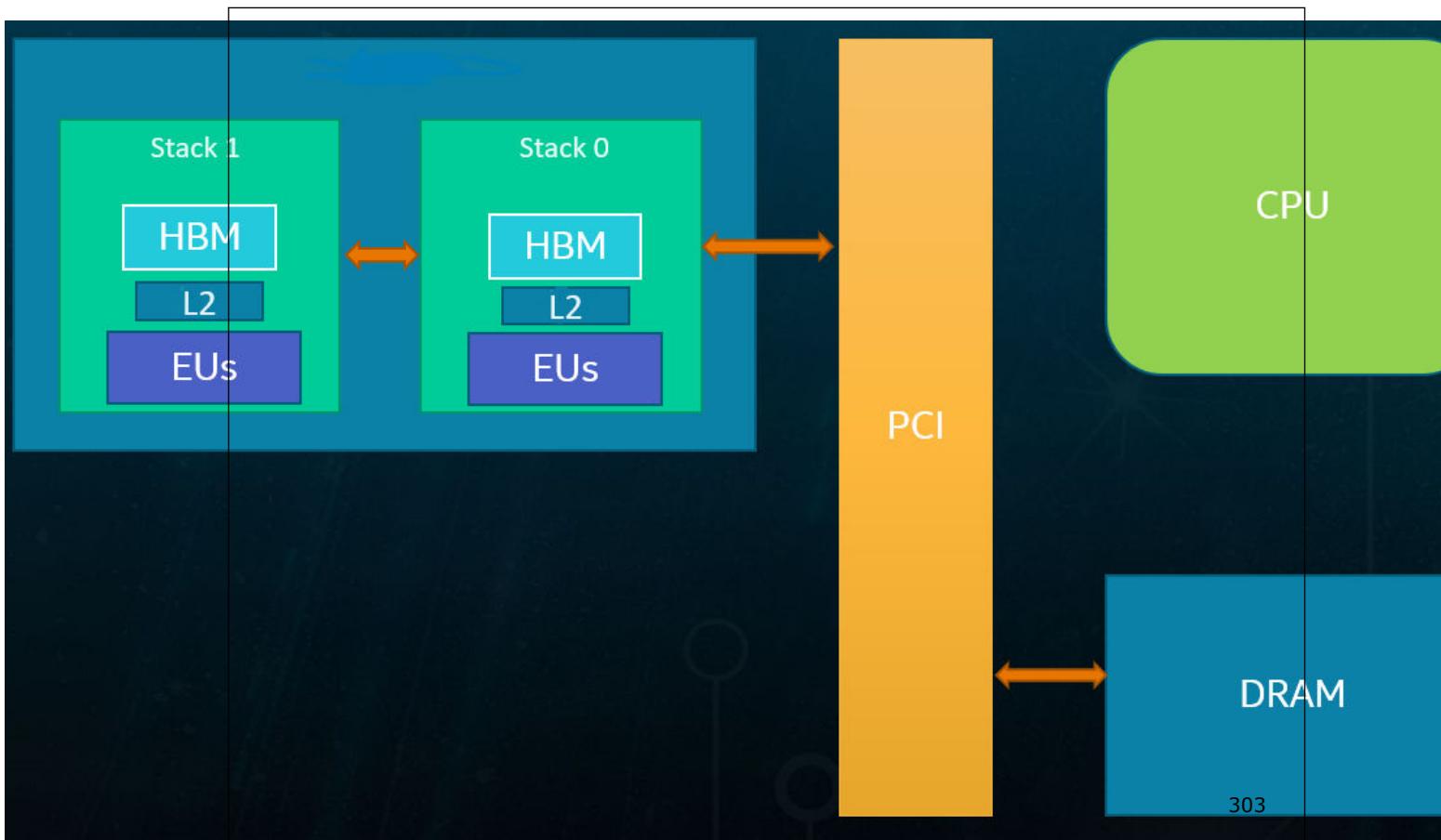
- [Multi-Stack GPU Architecture](#)
- [Exposing the Device Hierarchy](#)
- [FLAT Mode Programming](#)
- [COMPOSITE Mode Programming](#)
- [Using Intel® oneAPI Math Kernel Library \(oneMKL\)](#)
- [Using Intel® MPI Library](#)
- [Advanced Topics](#)
- [Terminology](#)

### Multi-Stack GPU Architecture

Intel® Data Center GPU Max Series use a multi-stack GPU architecture with 1 or 2 stacks.



Intel® Iris® Xe GPU Multi-Stack Architecture



The above figure illustrates 1-stack and 2-stack Intel® Data Center GPU Max Series products, each with its own dedicated resources:

Vector Engines (VEs)	Computation units belong to the stack
High Bandwidth Memory (HBM)	HBM directly connected to the stack
Level 2 Cache (L2)	Level 2 cache belonging to the stack

For general applications, the most common mode is to use each stack as a device (see next section, Exposing the Device Hierarchy). Intel GPU driver, as well as SYCL and OpenMP parallel language runtimes work together to dispatch kernels to the stack(s) in the GPU.

Stacks are connected with fast interconnect that allows efficient communication between stacks. The following operations are possible:

Any stack is capable of reading and writing to any HBM memory in the same GPU card. For example, stack 0 may read the local HBM memory of stack 1. In this case, the interconnect between stack 0 and stack 1 is used for communication.

Each stack is an independent entity. The stack can execute workloads on its own.

Because access to a stack's local HBM does not involve inter-stack interconnect, it is more efficient than cross-stack HBM access, with lower latency and lower inter-stack bandwidth consumption. Advanced developers can take advantage of memory locality to achieve higher performance.

The default for each stack is to use a single Compute Command Streamer (CCS) that includes all the hardware computing resources on that stack. Most applications will work well in this mode.

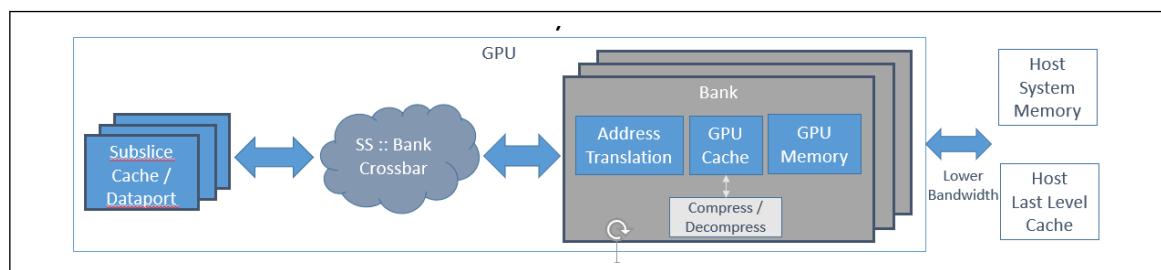
It is also possible to statically partition each stack, via the environment variable ZEX\_NUMBER\_OF\_CCS, into a 2-CCS mode or 4-CCS mode configuration, and treat each CCS as an entity to which kernels can be offloaded. For more information, refer to [Advanced Topics](#).

## GPU Memory System

The memory for a general-purpose engine is partitioned into host-side memory and device-side memory as shown in the following figure, 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 Xe-cores (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 cache lines. Also supports read/write of fast-clear surfaces. Improves bandwidth but also adds latency.

## GPU Memory System



GPU Memory accesses measured at VE:

- Sustained fabric bandwidth ~90% of peak
- GPU cache hit ~150 cycles, cache miss ~300 cycles. TLB miss adds 50-150 cycles
- GPU cache line read after write to same cache line adds ~30 cycles

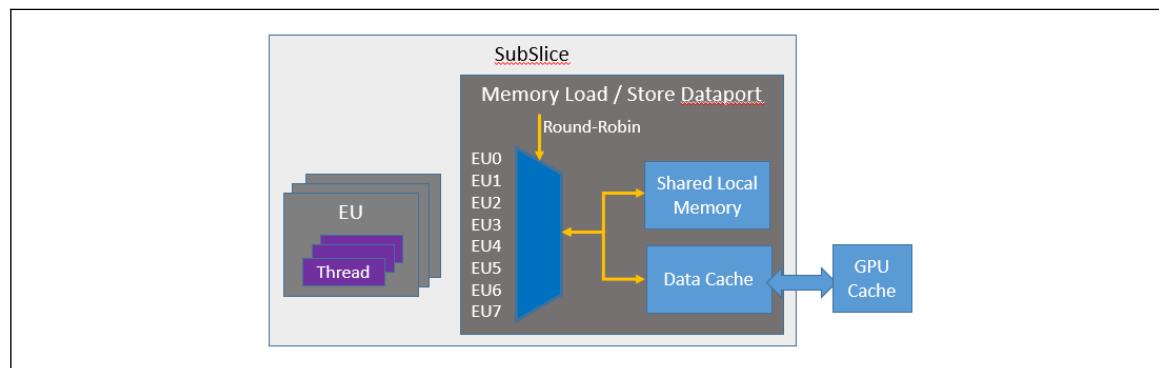
Stacks accessing device memory on a different stack utilize a new GAM-to-GAM High bandwidth interface ("GT Link") between stacks. The bandwidth of this interface closely matches the memory bandwidth that can be handled by the device memory sub-system of any single stack.

Loads/Stores/Atomics in VE Threads

VE threads make memory accesses by sending messages to a data-port, the load instruction sends address and receives data, The store instructions send address and data. All VE in X<sup>e</sup>-core share one Memory Load/Store data-port as shown in the figure below.

- Inside X<sup>e</sup>-core: ~128-256 Bytes per cycle
- Outside X<sup>e</sup>-core: ~64 Bytes per cycle
- Read bandwidth sometimes higher than write bandwidth

### VE 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:

#### Memory latency table

Access type	Latency
Shared local memory	~30 cycles
X <sup>e</sup> -core 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 the same thread may complete out-of-order, Read/Write ordering is not maintained between threads nor VEs nor X<sup>e</sup>-cores, so code needs to use atomic and/or fence operations 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 L2 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), and 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.

The following SYCL 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 VE 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.

```
#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;
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_;
};

#ifdef 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::no_init);
    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) {
    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

    Timer timer;

    q.submit([&](auto &h) {
        sycl::accessor buf_acc(inbuf, h, sycl::write_only, sycl::no_init);

        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::atomic_ref<int, sycl::memory_order::relaxed,
                                sycl::memory_scope::device,
                                sycl::access::address_space::global_space>(buf_acc[i]);
                    #else
                    sycl::atomic_ref<int, sycl::memory_order::acq_rel,
                                sycl::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_v, 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);
}

```

In real workloads with atomics, users need to understand memory access behaviors and data set sizes when selecting an atomic operation to achieve optimal bandwidth.

## Exposing the Device Hierarchy

A multi-stack GPU card can be exposed as a single root device, or each stack can be exposed as a root device. This can be controlled via the environment variable `ZE_FLAT_DEVICE_HIERARCHY`. The allowed values for `ZE_FLAT_DEVICE_HIERARCHY` are FLAT, COMPOSITE, or COMBINED.

Our focus in this Guide is on FLAT and COMPOSITE modes.

Note that, in a system with one stack per GPU card, FLAT and COMPOSITE are the same.

### `ZE_FLAT_DEVICE_HIERARCHY=FLAT (Default)`

The FLAT mode is the default mode if `ZE_FLAT_DEVICE_HIERARCHY` is not set. In FLAT mode, each stack is exposed as a root device. The recommendation is to use FLAT mode. The FLAT mode performs well for most applications.

In FLAT mode, the driver and language runtime provide tools that expose each stack as a root device that can be programmed independently of all the other stacks.

In FLAT mode, offloading is done using explicit scaling.

On a single or multiple GPU card system, the user can use all the stacks in all the GPU cards in FLAT mode, and offload to all the stacks (devices) simultaneously.

In OpenMP, the `device` clause on the `target` construct can be used to specify to which stack (device) the kernel should be offloaded.

In SYCL, `platform::get_devices()` can be called to get the stacks (devices) exposed.

For more information about the FLAT mode, refer to the [FLAT Mode Programming](#) section.

### `ZE_FLAT_DEVICE_HIERARCHY=COMPOSITE`

In COMPOSITE mode, each GPU card is exposed as a root device. If the card contains more than one stack, then the stacks on the GPU card are exposed as subdevices.

In COMPOSITE mode, offloading can be done using either explicit or implicit scaling.

Note that in earlier GPU drivers, the default was COMPOSITE mode and implicit scaling. Now the default is FLAT mode.

#### **Explicit Scaling in COMPOSITE Mode:**

In COMPOSITE mode, the driver and language runtime provide tools that expose each GPU card as a root device and the stacks as subdevices that can be programmed independently.

In OpenMP, the `device` and `subdevice` clauses on the `target` construct can be used to specify to which stack (subdevice) the kernel should be offloaded. (Note that the `subdevice` clause is an Intel extension to OpenMP.)

In SYCL, the `device::create_sub_devices()` can be called to get the subdevices or the stacks on each card device.

For more information about explicit scaling in COMPOSITE mode, refer to the [Explicit Scaling](#) section.

### Implicit Scaling in COMPOSITE Mode:

In COMPOSITE mode, if the program offloads to a device that is the entire card, then the driver and language runtime are, by default, responsible for work distribution and multi-stack memory placement.

The recommendation is to use explicit scaling. However, if the memory requirement is more than what is available in a single stack, then implicit scaling may be used.

For more information about implicit scaling in COMPOSITE mode, refer to the [Implicit Scaling](#) section.

## MPI Considerations

In an MPI application, each MPI rank may be configured to run on a GPU card or a GPU stack. Each rank can use OpenMP and SYCL to use the assigned GPU cards or stacks.

The table below shows common device configurations for MPI + OpenMP applications.

### Configurations for MPI + OpenMP (N Cards with 2 Stacks Each)

FLAT or COMPOSITE	Device Exposed	MPI Rank Assignment	OpenMP Devices(s) View	Implicit Scaling?	Recommended?
FLAT	Stack	1 rank per stack, 2*N ranks in total	1 stack as device0	No	Yes
COMPOSITE	Card	1 rank per stack, 2*N ranks in total	1 stack as device0	No	For expert users
COMPOSITE	Card	1 rank per card, N ranks in total	2 stacks as device0 and device1	No	Yes
COMPOSITE	Card	1 rank per card, N ranks in total	1 card as device0	Yes	If single stack memory is not sufficient

## Obtaining System and Debugging Information

The following two schemes can be used to obtain information about the system and devices.

- Before you run an application, it is recommended that you run the `sycl-ls` command on the command line to find out which devices are available on the platform. This information is especially useful when doing performance measurements.

Note that `sycl-ls` shows devices seen or managed by all backends. For example, running on a system with a single GPU card with 2 stacks in total, `sycl-ls` shows that there are 2 devices (corresponding to the 2 stacks) managed by either the Level Zero or OpenCL backend.

```
$ sycl-ls
[level_zero:gpu][level_zero:0] ... Intel(R) Data Center GPU Max 1550 1.3
[level_zero:gpu][level_zero:1] ... Intel(R) Data Center GPU Max 1550 1.3
[opencl:gpu][opencl:0] ... Intel(R) Data Center GPU Max 1550 OpenCL 3.0 NEO
[opencl:gpu][opencl:1] ... Intel(R) Data Center GPU Max 1550 OpenCL 3.0 NEO
```

- Set the environment variable LIBOMPTARGET\_DEBUG to 1 so the runtime would display debugging information, including information about which devices were found and used. Note that LIBOMPTARGET\_DEBUG is OpenMP-specific (it does not apply to SYCL). See example in the [FLAT Mode Example - OpenMP](#) section.

## Environment Variables to Control Device Exposure

The following environment variables can be used to control the hardware or devices that are exposed to the application.

- ZE\_FLAT\_DEVICE\_HIERARCHY=FLAT or COMPOSITE (default is FLAT). See [Device Hierarchy in Level Zero Specification Documentation](#).
- ONEAPI\_DEVICE\_SELECTOR. This environment variable is OpenMP-specific (does not apply in SYCL). The environment variable controls what hardware is exposed to the application. For details, see [ONEAPI\\_DEVICE\\_SELECTOR in \\_oneAPI DPC++ Compiler documentation](#).
- ZE\_AFFINITY\_MASK. This environment variable control what hardware is exposed by the Level-Zero User-Mode Driver (UMD). For details, see [Affinity Mask in Level Zero Specification Documentation](#).
- LIBOMPTARGET\_DEVICES=DEVICE or SUBDEVICE or SUBSUBDEVICE. This environment variable is OpenMP-specific (does not apply in SYCL). It can be used to map an OpenMP "device" to a GPU card (device), a stack (subdevice), or a Compute Command Streamer (subsubdevice). See [Compiling and Running an OpenMP Application in oneAPI GPU Optimization Guide](#).
- ZEX\_NUMBER\_OF\_CCS. See example in the [Advanced Topics](#) section.

## References

1. [Device Hierarchy in Level Zero Specification Documentation](#)
2. [Affinity Mask in Level Zero Specification Documentation](#)
3. [ONEAPI\\_DEVICE\\_SELECTOR in \\_oneAPI DPC++ Compiler documentation](#)
4. [Compiling and Running an OpenMP Application in oneAPI GPU Optimization Guide](#)

## FLAT Mode Programming

As mentioned previously, the FLAT mode is the default mode on Intel® Data Center GPU Max Series. In FLAT mode, each stack is exposed as a root device. In this section, we present SYCL and OpenMP examples to demonstrate offloading in FLAT mode.

### Memory in FLAT Mode

Each stack has its own memory. A kernel offloaded to a stack will run on that stack and use the memory allocated on that stack.

A kernel running on a stack can access memory on other stacks in the same GPU card. However, accessing memory on a stack other than the stack it is running on will be slower.

- [FLAT Mode Example - SYCL](#)
- [FLAT Mode Example - OpenMP](#)

### FLAT Mode Example - SYCL

In this section, we use a simple vector addition example to show how to scale the performance using multiple devices in FLAT mode.

#### Offloading to a single device (stack)

A first look at adding 2 vectors using one device or stack:

```
float *da;
float *db;
float *dc;
```

```

da = (float *)sycl::malloc_device<float>(gsize, q);
db = (float *)sycl::malloc_device<float>(gsize, q);
dc = (float *)sycl::malloc_device<float>(gsize, q);
q.memcpy(da, ha, gsize);
q.memcpy(db, hb, gsize);

q.wait();

std::cout << "Offloading work to 1 device" << std::endl;

for (int i = 0; i < 16; i++) {
    q.parallel_for(sycl::nd_range<1>(gsize, 1024), [=] (auto idx) {
        int ind = idx.get_global_id();
        dc[ind] = da[ind] + db[ind];
    });
}

q.wait();

std::cout << "Offloaded work completed" << std::endl;

q.memcpy(hc, dc, gsize);

```

The above example adds two float vectors `ha` and `hb` and then saves the result in vector `hc`.

The example first allocates device memory for the 3 vectors, then copies the data from the host to the device before launching the kernels on the device to do vector addition. After the computation on the device completes, the result in vector `dc` on the device is copied to the vector `hc` on the host.

#### Compilation and run commands:

```
$ icpx -fsycl flat_sycl_vec_add_single_device.cpp -o flat_sycl_vec_add_single_device
$ ./flat_sycl_vec_add_single_device
```

#### Offloading to multiple devices (stacks)

We can scale up the performance of the above example if multiple devices are available.

The following example starts with enumerating all devices on the platform to make sure that at least 2 devices are available.

```

auto plat = sycl::platform(sycl::gpu_selector_v);
auto devs = plat.get_devices();
auto ctxt = sycl::context(devs);

if (devs.size() < 2) {
    std::cerr << "No 2 GPU devices found" << std::endl;
    return -1;
}

std::cout << devs.size() << " GPU devices are found and 2 will be used" << std::endl;
sycl::queue q[2];
q[0] = sycl::queue(ctxt, devs[0], {sycl::property::queue::in_order()});
q[1] = sycl::queue(ctxt, devs[1], {sycl::property::queue::in_order()});

```

Next, the example allocates the vectors `ha` and `hb` on the host and partitions each vector into 2 parts. Then the first halves of the vectors `ha` and `hb` are copied to the first device, and the second halves are copied to the second device.

```

constexpr size_t gsize = 1024 * 1024 * 1024L;
float *ha = (float *)malloc(gsize * sizeof(float));
float *hb = (float *)malloc(gsize * sizeof(float)));

```

```

float *hc = (float *) (malloc(gsize * sizeof(float)));

for (size_t i = 0; i < gsize; i++) {
    ha[i] = float(i);
    hb[i] = float(i + gsize);
}

float *da[2];
float *db[2];
float *dc[2];

size_t lsize = gsize / 2;

da[0] = (float *)sycl::malloc_device<float>(lsize, q[0]);
db[0] = (float *)sycl::malloc_device<float>(lsize, q[0]);
dc[0] = (float *)sycl::malloc_device<float>(lsize, q[0]);
q[0].memcpy(da[0], ha, lsize);
q[0].memcpy(db[0], hb, lsize);

da[1] = (float *)sycl::malloc_device<float>((lsize + gsize % 2), q[1]);
db[1] = (float *)sycl::malloc_device<float>((lsize + gsize % 2), q[1]);
dc[1] = (float *)sycl::malloc_device<float>((lsize + gsize % 2), q[1]);
q[1].memcpy(da[1], ha + lsize, lsize + gsize % 2);
q[1].memcpy(db[1], hb + lsize, lsize + gsize % 2);

q[0].wait();
q[1].wait();

```

Once the data is available on the two devices, the vector addition kernels are launched on each device and the devices execute the kernels in parallel. After the computations on both devices complete, the results are copied from both the devices to the host.

```

for (int i = 0; i < 16; i++) {
    q[0].parallel_for(sycl::nd_range<1>(lsize, 1024), [=](auto idx) {
        int ind = idx.get_global_id();
        dc[0][ind] = da[0][ind] + db[0][ind];
    });
    q[1].parallel_for(sycl::nd_range<1>(lsize + gsize % 2, 1024), [=](auto idx) {
        int ind = idx.get_global_id();
        dc[1][ind] = da[1][ind] + db[1][ind];
    });
}

q[0].wait();
q[1].wait();

std::cout << "Offloaded work completed" << std::endl;

q[0].memcpy(hc, dc[0], lsize);
q[1].memcpy(hc + lsize, dc[1], lsize + gsize % 2);

q[0].wait();
q[1].wait();

```

### Compilation and run commands:

```

$ icpx -fsycl flat_sycl_vec_add.cpp -o flat_sycl_vec_add
$ ./flat_sycl_vec_add

```

Note that this example uses 2 devices. It can easily be extended to use more than 2 devices if more than 2 devices are available. We leave this as an exercise.

## FLAT Mode Example - OpenMP

As previously mentioned, in FLAT mode, the stacks are exposed as devices.

### Offloading to a single device (stack)

In this scheme, the default root device which is device 0 is used to offload. See code example below:

```
int device_id = omp_get_default_device();

#pragma omp target teams distribute parallel for device(device_id) map(...)
for (int i = 0, i < N; i++) {
    ...
}
```

### Offloading to multiple devices (stacks)

In this scheme, we have multiple root devices (stacks) on which the code will run; the stacks may belong to one or more GPU cards. See code example below:

```
int num_devices = omp_get_num_devices();

#pragma omp parallel for
for (int device_id = 0; device_id < num_devices; device_id++) {

    #pragma omp target teams distribute parallel for device(device_id) map(...)
        for (int i = lb(device_id); I < ub(device_id); i++) {
            ...
        }
}
```

We present below a full OpenMP program that offloads to multiple devices (stacks) in FLAT mode.

## OpenMP Example

In the following program, `flat_openmp_01.cpp`, the array A is initialized on the device. First, we determine the number of devices (stacks) available, and then use the devices (stacks) to initialize different chunks of the array. The OpenMP `device` clause on the `target` pragma is used to specify which stack to use for a particular chunk. (If no `device` clause is specified, then the code will run on stack 0.)

`omp_get_num_devices()` returns the total number of devices (stacks) that are available. For example, on a 4-card system with 2 stacks each, the routine will return 8.

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

#define SIZE 320

int num_devices = omp_get_num_devices();
int chunksize = SIZE/num_devices;

int main(void)
{
    int *A;
    A = new int[sizeof(int) * SIZE];

    printf ("num_devices = %d\n", num_devices);
```

```

for (int i = 0; i < SIZE; i++)
    A[i] = -9;

#pragma omp parallel for
for (int id = 0; id < num_devices; id++) {
    #pragma omp target teams distribute parallel for device(id) \
        map(tofrom: A[id * chunksize : chunksize])
    for (int i = id * chunksize; i < (id + 1) * chunksize; i++) {
        A[i] = i;
    }
}

for (int i = 0; i < SIZE; i++)
    if (A[i] != i)
        printf ("Error in: %d\n", A[i]);
    else
        printf ("%d\n", A[i]);
}

```

### Compilation command:

```
$ icpx -fiopenmp -fopenmp-targets=spir64 flat_openmp_01.cpp
```

### Run command:

```
$ OMP_TARGET_OFFLOAD=MANDATORY ./a.out
```

### Notes:

- OMP\_TARGET\_OFFLOAD=MANDATORY is used to make sure that the target region will run on the GPU. The program will fail if a GPU is not found.
- There is no need to specify ZE\_FLAT\_DEVICE\_HIERARCHY=FLAT with the run command, since FLAT mode is the default.

### Running on a system with a single GPU card (2 stacks in total):

sycl-ls shows that there are 2 devices (corresponding to the 2 stacks):

```
$ sycl-ls
[level_zero:gpu][level_zero:0] ... Intel(R) Data Center GPU Max 1550 1.3
[level_zero:gpu][level_zero:1] ... Intel(R) Data Center GPU Max 1550 1.3
[opencl:gpu][opencl:0] ... Intel(R) Data Center GPU Max 1550 OpenCL 3.0 NEO
[opencl:gpu][opencl:1] ... Intel(R) Data Center GPU Max 1550 OpenCL 3.0 NEO
```

We add LIBOMPTARGET\_DEBUG=1 to the run command to get libomptarget.so debug information.

```
$ OMP_TARGET_OFFLOAD=MANDATORY LIBOMPTARGET_DEBUG=1 ./a.out >& libomptarget_debug.log
```

We see the following in libomptarget\_debug.log, showing that 2 devices (corresponding to the 2 stacks) have been found.

```
Target LEVEL_ZERO RTL --> Found a GPU device, Name = Intel(R) Data Center GPU Max 1550
Target LEVEL_ZERO RTL --> Found 2 root devices, 2 total devices.
Target LEVEL_ZERO RTL --> List of devices (DeviceID[.SubID[.CCSID]])
Target LEVEL_ZERO RTL --> -- 0
Target LEVEL_ZERO RTL --> -- 1
```

### Running on a system with 4 GPU cards (8 stacks in total)

`sycl-ls` shows that there are 8 devices (corresponding to the 8 stacks):

```
$ sycl-ls
[level_zero:gpu][level_zero:0] ... Intel(R) Data Center GPU Max 1550 1.3
[level_zero:gpu][level_zero:1] ... Intel(R) Data Center GPU Max 1550 1.3
[level_zero:gpu][level_zero:2] ... Intel(R) Data Center GPU Max 1550 1.3
[level_zero:gpu][level_zero:3] ... Intel(R) Data Center GPU Max 1550 1.3
[level_zero:gpu][level_zero:4] ... Intel(R) Data Center GPU Max 1550 1.3
[level_zero:gpu][level_zero:5] ... Intel(R) Data Center GPU Max 1550 1.3
[level_zero:gpu][level_zero:6] ... Intel(R) Data Center GPU Max 1550 1.3
[level_zero:gpu][level_zero:7] ... Intel(R) Data Center GPU Max 1550 1.3
[opencl:gpu][opencl:0] ... Intel(R) Data Center GPU Max 1550 OpenCL 3.0 NEO
[opencl:gpu][opencl:1] ... Intel(R) Data Center GPU Max 1550 OpenCL 3.0 NEO
[opencl:gpu][opencl:2] ... Intel(R) Data Center GPU Max 1550 OpenCL 3.0 NEO
[opencl:gpu][opencl:3] ... Intel(R) Data Center GPU Max 1550 OpenCL 3.0 NEO
[opencl:gpu][opencl:4] ... Intel(R) Data Center GPU Max 1550 OpenCL 3.0 NEO
[opencl:gpu][opencl:5] ... Intel(R) Data Center GPU Max 1550 OpenCL 3.0 NEO
[opencl:gpu][opencl:6] ... Intel(R) Data Center GPU Max 1550 OpenCL 3.0 NEO
[opencl:gpu][opencl:7] ... Intel(R) Data Center GPU Max 1550 OpenCL 3.0 NEO
```

We add `LIBOMPTARGET_DEBUG=1` to the run command to get `libomptarget.so` debug information.

```
$ OMP_TARGET_OFFLOAD=MANDATORY LIBOMPTARGET_DEBUG=1 ./a.out >& libomptarget_debug.log
```

We see the following in `libomptarget_debug.log`, showing that 8 devices (corresponding to the 8 stacks) have been found:

```
Target LEVEL_ZERO RTL --> Found a GPU device, Name = Intel(R) Data Center GPU Max 1550
Target LEVEL_ZERO RTL --> Found 8 root devices, 8 total devices.
Target LEVEL_ZERO RTL --> List of devices (DeviceID[.SubID[.CCSID]])
Target LEVEL_ZERO RTL --> -- 0
Target LEVEL_ZERO RTL --> -- 1
Target LEVEL_ZERO RTL --> -- 2
Target LEVEL_ZERO RTL --> -- 3
Target LEVEL_ZERO RTL --> -- 4
Target LEVEL_ZERO RTL --> -- 5
Target LEVEL_ZERO RTL --> -- 6
Target LEVEL_ZERO RTL --> -- 7
```

## COMPOSITE Mode Programming

As mentioned earlier, in COMPOSITE mode, each GPU card is exposed as a root device. If the card contains more than one stack, then the stacks on the GPU card are exposed as subdevices.

In COMPOSITE mode, offloading can be done using either explicit or implicit scaling. In the following sections we describe implicit and explicit scaling in more detail.

- [Explicit Scaling](#)
- [Implicit Scaling](#)

### Explicit Scaling

In explicit scaling, the programmer takes direct control over work group distribution and memory placement.

In this section, we cover:

- [Explicit Scaling - SYCL](#)
- [Explicit Scaling - OpenMP](#)
- [Explicit Scaling Summary](#)

## Explicit Scaling - SYCL

In this section we describe explicit scaling in SYCL in COMPOSITE mode provide usage examples.

### Unified shared memory (USM)

Memory allocated against a root device is accessible by all of its subdevices (stacks). So if you are operating on a context with multiple subdevices 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 would need an explicit copy out to the host if it necessary to see data on the host. Refer to section [Unified Shared Memory Allocations](#) for the details on the three types of USM allocations.

### subdevice

Intel® Data Center GPU Max 1350 or 1550 has 2 stacks. The root device, corresponding to the whole GPU, can be partitioned to 2 subdevices, each subdevice corresponding to a physical stack.

```
try {
    vector<device> SubDevices = RootDevice.create_sub_devices<
        cl::sycl::info::partition_property::partition_by_affinity_domain>(
        cl::sycl::info::partition_affinity_domain::numa);
}
```

In SYCL, each call to `create_sub_devices` will return exactly the same subdevices.

Note that the `partition_by_affinity_domain` is the only partitioning supported for Intel GPUs. Similarly, `next_partitionable` and `numa` are the only partitioning properties supported (both doing the same thing).

To control what subdevices are exposed, one can use `ONEAPI_DEVICE_SELECTOR` described in [ONEAPI\\_DEVICE\\_SELECTOR in \\_oneAPI DPC++ Compiler documentation](#).

To control what subdevices are exposed by the Level-Zero User-Mode Driver (UMD) one can use `ZE_AFFINITY_MASK` environment variable described in [Affinity Mask in Level Zero Specification Documentation](#).

In COMPOSITE mode, each subdevice (stack) can be further decomposed to subsubdevices (Compute Command Streamers or CCSs). For more information about subsubdevices, refer to [Advanced Topics](#) section.

### Context

A context is used for resources isolation and sharing. A SYCL context may consist of one or multiple devices. Both root devices and subdevices 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 for each of the root devices in the context. For a context that consists of multiple subdevices of the same root device only a single build (to that root device) is needed.

### Buffers

SYCL buffers are created against a context and are mapped to the Level-Zero USM allocation discussed above. Current mapping is as follows.

- For an integrated device, the allocations are made on the host, and are accessible by the host and the device without any copying.
- Memory buffers for a context with subdevices 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 the context. The synchronization with the host is performed by the SYCL runtime with map/unmap doing implicit copies when necessary.
- Memory buffers for a context with devices from different root devices in it are allocated on the host (thus made accessible to all devices).

## Queues

A SYCL queue is always attached to a single device in a possibly multi-device context. Here are some typical scenarios, in the order of most performant to least performant:

### Context associated with a single subdevice

Creating a context with a single subdevice in it and the queue is attached to that subdevice (stack): In this scheme, the execution/visibility is limited to the single subdevice only, and expected to offer the best performance per stack. See code example:

```
try {
    vector<device> SubDevices = ...;
    for (auto &D : SubDevices) {
        // Each queue is in its own context, no data sharing across them.
        auto Q = queue(D);
        Q.submit([&](handler &cgh) { ... });
    }
}
```

### Context associated with multiple subdevices

Creating a context with multiple subdevices (multiple stacks) of the same root device: In this scheme, queues are attached to the subdevices, effectively implementing “explicit scaling”. In this scheme, the root device should not be passed to such a context for better performance. See code example below:

```
try {
    vector<device> SubDevices = ...;
    auto C = context(SubDevices);
    for (auto &D : SubDevices) {
        // All queues share the same context, data can be shared across
        // queues.
        auto Q = queue(C, D);
        Q.submit([&](handler &cgh) { ... });
    }
}
```

### Context associated with a single 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 subdevices/stacks via “implicit scaling” by the GPU driver, which is the most simple way to enable multi-stack hardware but does not offer the possibility to target specific stacks. See code example below:

```
try {
    // The queue is attached to the root-device, driver distributes to
    // sub - devices, if any.
    auto D = device(gpu_selector{});
    auto Q = queue(D);
    Q.submit([&](handler &cgh) { ... });
}
```

### Context associated with multiple root devices

Creating Contexts with multiple root devices (multi-card): This scheme, the most nonrestrictive context with queues attached to different root devices, offers more sharing possibilities at the cost of slow access through host memory or explicit copies needed. See a code example:

```
try {
    auto P = platform(gpu_selector{});
    auto RootDevices = P.get_devices();
```

```

auto C = context(RootDevices);
for (auto &D : RootDevices) {
    // Context has multiple root-devices, data can be shared across
    // multi - card(requires explicit copying)
    auto Q = queue(C, D);
    Q.submit([&](handler &cgh) { ... });
}
}

```

Depending on the chosen explicit subdevices usage described and the algorithm used, make sure to do proper memory allocation/synchronization. The following program is a full example using explicit subdevices - one queue per stack:

```

#include <CL/sycl.hpp>
#include <algorithm>
#include <cassert>
#include <cfloat>
#include <iostream>
#include <string>
using namespace sycl;

constexpr int num_runs = 10;
constexpr size_t scalar = 3;

cl_ulong triad(size_t array_size) {

    cl_ulong min_time_ns0 = DBL_MAX;
    cl_ulong min_time_ns1 = DBL_MAX;

    device dev = device(gpu_selector_v);

    std::vector<device> subdev = {};
    subdev = dev.create_sub_devices<sycl::info::partition_property::partition_by_affinity_domain>(sycl::info::partition_affinity_domain::numa);

    queue q[2] = {queue(subdev[0], property::queue::enable_profiling{}),
                  queue(subdev[1], property::queue::enable_profiling{})};

    std::cout << "Running on device: " <<
    q[0].get_device().get_info<info::device::name>() << "\n";
    std::cout << "Running on device: " <<
    q[1].get_device().get_info<info::device::name>() << "\n";

    double *A0 = malloc_shared<double>(array_size/2 * sizeof(double), q[0]);
    double *B0 = malloc_shared<double>(array_size/2 * sizeof(double), q[0]);
    double *C0 = malloc_shared<double>(array_size/2 * sizeof(double), q[0]);

    double *A1 = malloc_shared<double>(array_size/2 * sizeof(double), q[1]);
    double *B1 = malloc_shared<double>(array_size/2 * sizeof(double), q[1]);
    double *C1 = malloc_shared<double>(array_size/2 * sizeof(double), q[1]);

    for (int i = 0; i < array_size/2; i++) {
        A0[i] = 1.0; B0[i] = 2.0; C0[i] = 0.0;
        A1[i] = 1.0; B1[i] = 2.0; C1[i] = 0.0;
    }

    for (int i = 0; i < num_runs; i++) {
        auto q0_event = q[0].submit([&](handler& h) {

```

```
    h.parallel_for(array_size/2, [=](id<1> idx) {
        C0[idx] = A0[idx] + B0[idx] * scalar;
    });
});

auto q1_event = q[1].submit([&](handler& h) {
    h.parallel_for(array_size/2, [=](id<1> idx) {
        C1[idx] = A1[idx] + B1[idx] * scalar;
    });
});

q[0].wait();
q[1].wait();

cl_ulong exec_time_ns0 =
    q0_event.get_profiling_info<info::event_profiling::command_end>() -
    q0_event.get_profiling_info<info::event_profiling::command_start>();

std::cout << "Tile-0 Execution time (iteration " << i << ") [sec]: "
    << (double)exec_time_ns0 * 1.0E-9 << "\n";
min_time_ns0 = std::min(min_time_ns0, exec_time_ns0);

cl_ulong exec_time_ns1 =
    q1_event.get_profiling_info<info::event_profiling::command_end>() -
    q1_event.get_profiling_info<info::event_profiling::command_start>();

std::cout << "Tile-1 Execution time (iteration " << i << ") [sec]: "
    << (double)exec_time_ns1 * 1.0E-9 << "\n";
min_time_ns1 = std::min(min_time_ns1, exec_time_ns1);
}

// Check correctness
bool error = false;
for ( int i = 0; i < array_size/2; i++) {
    if ((C0[i] != A0[i] + scalar * B0[i]) || (C1[i] != A1[i] + scalar * B1[i])) {
        std::cout << "\nResult incorrect (element " << i << " is " << C0[i] << ")\n";
        error = true;
    }
}

sycl::free(A0, q[0]);
sycl::free(B0, q[0]);
sycl::free(C0, q[0]);

sycl::free(A1, q[1]);
sycl::free(B1, q[1]);
sycl::free(C1, q[1]);

if (error) return -1;

std::cout << "Results are correct!\n\n";
return std::max(min_time_ns0, min_time_ns1);
}

int main(int argc, char *argv[]) {

    size_t array_size;
    if (argc > 1) {
```

```

array_size = std::stoi(argv[1]);
}
else {
    std::cout << "Run as ./<progname> <arraysize in elements>\n";
    return 1;
}
std::cout << "Running with stream size of " << array_size
<< " elements (" << (array_size * sizeof(double))/(double)1024/1024 << "MB)\n";

cl_ulong min_time = triad(array_size);

if (min_time == -1) return 1;
size_t triad_bytes = 3 * sizeof(double) * array_size;
std::cout << "Triad Bytes: " << triad_bytes << "\n";
std::cout << "Time in sec (fastest run): " << min_time * 1.0E-9 << "\n";
double triad_bandwidth = 1.0E-09 * triad_bytes/(min_time*1.0E-9);
std::cout << "Bandwidth of fastest run in GB/s: " << triad_bandwidth << "\n";
return 0;
}

```

The build command using Ahead-Of-Time or AOT compilation is:

```
icpx -fsycl -fsycl-targets=spir64_gen -O2 -ffast-math -Xs "-device pvc" explicit-subdevice.cpp -o run.exe
```

## References

1. [Affinity Mask in Level Zero Specification Documentation](#)
2. [ONEAPI\\_DEVICE\\_SELECTOR in \\_oneAPI DPC++ Compiler documentation](#)

### *Explicit Scaling - OpenMP*

In this section we describe explicit scaling in OpenMP in COMPOSITE mode provide usage examples.

Remember to set the environment variable ZE\_FLAT\_DEVICE\_HIERARCHY=COMPOSITE to enable COMPOSITE mode.

### **Unified Shared Memory (USM)**

Three OpenMP APIs as Intel extensions for USM memory allocations have been added:

`omp_target_alloc_host`, `omp_target_alloc_device`, and `omp_target_alloc_shared`.

Please refer to [OpenMP USM Allocation API](#) section for details.

### Offloading to multiple subdevices

In this scheme, we have multiple subdevices on which the code will run, and queues are attached to the subdevices. This effectively results in “explicit scaling”. See code example below:

```

#define DEVKIND 0 // Stack

int root_id = omp_get_default_device();

#pragma omp parallel for
for (int id = 0; id < NUM_SUBDEVICES; ++id) {

#pragma omp target teams distribute parallel for device(root_id)
    subdevice(DEVKIND, id) map(...)
    for (int i = lb(id), i < ub(id); i++) {

```

```

    ...
}
}
```

In COMPOSITE mode, each subdevice (stack) can be further decomposed to subsubdevices (Compute Command Streamers or CCSs). For more information about subsubdevices, refer to [Advanced Topics](#) section.

#### Offloading to a single root device

In this scheme, we have a single root device and a queue attached to the root device. The work will be automatically distributed across all subdevices/stacks via “implicit scaling” by the GPU driver. This is the most simple way to enable multi-stack utilization, without targeting specific stacks. See code example below:

```

int root_id = omp_get_default_device();

#pragma omp target teams distribute parallel for device(root_id) map(...)
for (int i = 0, i < N; i++) {
    ...
}
```

#### Offloading to multiple root devices

In this scheme, we have multiple root devices, where each root device is a GPU card. The queues are attached to the root devices, which offers more sharing possibilities but at the cost of slow access through host memory or explicit copying of data. See code example:

```

int num_devices = omp_get_num_devices();

#pragma omp parallel for
for (int root_id = 0; root_id < num_devices; root_id++) {

#pragma omp target teams distribute parallel for device(root_id) map(...)
    for (int i = lb(root_id); I < ub(root_id); i++) {
        ...
    }
}
```

#### Program: Offloading to subdevices (stacks) in COMPOSITE mode

Depending on the chosen devices or subdevices used, as well as the algorithm used, be sure to do proper memory allocation/synchronization. The following is a full OpenMP program that offloads to multiple subdevices (stacks) in COMPOSITE mode.

```

#include <assert.h>
#include <iostream>
#include <omp.h>
#include <stdint.h>
#ifndef NUM_SUBDEVICES
#define NUM_SUBDEVICES 1
#endif

#ifndef DEVKIND
#define DEVKIND 0 // Stack
#endif

template <int num_subdevices> struct mptr {
    float *p[num_subdevices];
};

int main(int argc, char **argv) {
```

```

constexpr int SIZE = 8e6;
constexpr int SIMD_SIZE = 32;
constexpr std::size_t TOTAL_SIZE = SIZE * SIMD_SIZE;
constexpr int num_subdevices = NUM_SUBDEVICES;

mptr<num_subdevices> device_ptr_a;
mptr<num_subdevices> device_ptr_b;
mptr<num_subdevices> device_ptr_c;

const int default_device = omp_get_default_device();
std::cout << "default_device = " << default_device << std::endl;

for (int sdev = 0; sdev < num_subdevices; ++sdev) {
    device_ptr_a.p[sdev] =
        static_cast<float*>(malloc(TOTAL_SIZE * sizeof(float)));
    device_ptr_b.p[sdev] =
        static_cast<float*>(malloc(TOTAL_SIZE * sizeof(float)));
    device_ptr_c.p[sdev] =
        static_cast<float*>(malloc(TOTAL_SIZE * sizeof(float)));

#pragma omp target enter data map(alloc
                           : device_ptr_a.p[sdev] [0:TOTAL_SIZE])
                           \
                           device(default_device) subdevice(DEVKIND, sdev)

#pragma omp target enter data map(alloc
                           : device_ptr_b.p[sdev] [0:TOTAL_SIZE])
                           \
                           device(default_device) subdevice(DEVKIND, sdev)

#pragma omp target enter data map(alloc
                           : device_ptr_c.p[sdev] [0:TOTAL_SIZE])
                           \
                           device(default_device) subdevice(DEVKIND, sdev)
} // for (int sdev ...

std::cout << "memory footprint per GPU = "
      << 3 * (std::size_t)(TOTAL_SIZE) * sizeof(float) * 1E-9 << " GB"
      << std::endl;

#pragma omp parallel for
for (int sdev = 0; sdev < num_subdevices; ++sdev) {
    float *a = device_ptr_a.p[sdev];
    float *b = device_ptr_b.p[sdev];

#pragma omp target teams distribute parallel for device(default_device)
                           \
                           subdevice(DEVKIND, sdev)
    for (int i = 0; i < TOTAL_SIZE; ++i) {
        a[i] = i + 0.5;
        b[i] = i - 0.5;
    }
}

const int no_max_rep = 200;
double time = 0.0;
for (int irep = 0; irep < no_max_rep + 1; ++irep) {
    if (irep == 1)
        time = omp_get_wtime();

#pragma omp parallel for num_threads(num_subdevices)
    for (int sdev = 0; sdev < num_subdevices; ++sdev) {

```

```

float *a = device_ptr_a.p[sdev];
float *b = device_ptr_b.p[sdev];
float *c = device_ptr_c.p[sdev];

#pragma omp target teams distribute parallel for device(default_device) \
    subdevice(DEVKIND, sdev)
    for (int i = 0; i < TOTAL_SIZE; ++i) {
        c[i] = a[i] + b[i];
    }
}

time = omp_get_wtime() - time;
time = time / no_max_rep;

const std::size_t streamed_bytes =
    3 * (std::size_t)(TOTAL_SIZE)*num_subdevices * sizeof(float);
std::cout << "bandwidth = " << (streamed_bytes / time) * 1E-9 << " GB/s"
    << std::endl;
std::cout << "time = " << time << " s" << std::endl;
std::cout.precision(10);

for (int sdev = 0; sdev < num_subdevices; ++sdev) {
#pragma omp target update from(device_ptr_c.p[sdev][:TOTAL_SIZE]) \
    device(default_device) subdevice(DEVKIND, sdev)
    std::cout << "-GPU: device id = : " << sdev << std::endl;
    std::cout << "target result:" << std::endl;
    std::cout << "c[" << 0 << "] = " << device_ptr_c.p[sdev][0] << std::endl;
    std::cout << "c[" << SIMD_SIZE - 1
        << "] = " << device_ptr_c.p[sdev][SIMD_SIZE - 1] << std::endl;
    std::cout << "c[" << TOTAL_SIZE / 2
        << "] = " << device_ptr_c.p[sdev][TOTAL_SIZE / 2] << std::endl;
    std::cout << "c[" << TOTAL_SIZE - 1
        << "] = " << device_ptr_c.p[sdev][TOTAL_SIZE - 1] << std::endl;
}
}

for (int sdev = 0; sdev < num_subdevices; ++sdev) {
    for (int i = 0; i < TOTAL_SIZE; ++i) {
        assert((int)(device_ptr_c.p[sdev][i]) ==
            (int)(device_ptr_c.p[sdev][i] +
                device_ptr_a.p[sdev][i] * device_ptr_b.p[sdev][i]));
    }
}

for (int sdev = 0; sdev < num_subdevices; ++sdev) {
#pragma omp target exit data map(release
            : device_ptr_a.p[sdev][:TOTAL_SIZE]) \
    device(default_device) subdevice(DEVKIND, sdev)
#pragma omp target exit data map(release
            : device_ptr_b.p[sdev][:TOTAL_SIZE]) \
    device(default_device) subdevice(DEVKIND, sdev)
#pragma omp target exit data map(release
            : device_ptr_a.p[sdev][:TOTAL_SIZE])
    device(default_device) subdevice(DEVKIND, sdev)
}
}

```

Compilation command:

```
$ icpx -ffp-contract=fast -O2 -ffast-math -DNUM_SUBDEVICES=2 \
-fiopenmp -fopenmp-targets=spir64 openmp_explicit_subdevice.cpp
```

Run command:

```
$ ZE_FLAT_DEVICE_HIERARCHY=COMPOSITE OMP_TARGET_OFFLOAD=MANDATORY ./a.out
```

This OpenMP program achieves linear scaling  $\sim 2x$  on an Intel<sup>®</sup> Data Center GPU Max system.

### *Explicit Scaling Summary*

Performance tuning for a multi-stack GPU imposes a tedious process given the parallelism granularity is at a finer level. However, the fundamentals are similar to CPU performance tuning. To understand performance scaling dominators, one needs to pay attention to:

- VE utilization efficiency - how kernels utilize the execution resources of different stacks
- Data placement - how allocations are spread across the HBM of different stacks
- 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`)
- Explicit and implicit scaling
- Contexts and queues
- Environment variables and program language APIs or constructs

## **Implicit Scaling**

As mentioned above, in COMPOSITE mode, the driver and language runtime provide tools that expose each GPU card as a root device. In this mode, a root device is composed of multiple sub-devices, also known as stacks. The stacks form a shared memory space which allows treating a root device as a monolithic device without the requirement of explicit communication between stacks.

This section covers multi-stack programming principles using implicit scaling in COMPOSITE mode.

- [Introduction](#)
- [Work Scheduling and Memory Distribution](#)
- [STREAM Example](#)
- [Programming Principles](#)

### *Introduction*

Implicit scaling applies to multiple stacks inside a single GPU card only.

In COMPOSITE mode, if the program offloads to a device that is the entire card, then the driver and language runtime are, by default, responsible for work distribution and multi-stack memory placement.

For implicit scaling, no change in application code is required. An OpenMP/SYCL kernel submitted to a device will utilize all the stacks on that device. Similarly, memory allocated on the device will be accessible across all the stacks. The driver behavior is described in [Work Scheduling and Memory Distribution](#).

Notes on implicit scaling:

- Set **`ZE_FLAT_DEVICE_HIERARCHY=COMPOSITE`** to allow implicit scaling.
- Implicit scaling should not be combined with SYCL/OpenMP sub-device semantics.
- Do not use sub-device syntax in **`ZE_AFFINITY_MASK`**. That is, instead of exposing stack 0 in root device 0 (**`ZE_AFFINITY_MASK=0.0`**), you must expose the entire root device to the driver via **`ZE_AFFINITY_MASK=0`** or by unsetting **`ZE_AFFINITY_MASK`**.

## Performance Expectations

In implicit scaling, the resources of all the stacks are exposed to a kernel. When using a root device with 2 stacks, a kernel can achieve 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 the workload fits in the increased cache capacity.

Each stack is equivalent to a NUMA domain and therefore memory access pattern and memory allocation are crucial to achieving optimal implicit scaling performance. Workloads with a concept of locality are expected to work best with this programming model as cross-stack memory accesses are naturally minimized. Note that compute-bound kernels are not impacted by NUMA domains, thus they are expected to easily scale to multiple stacks with implicit scaling. If the algorithm has a lot of cross-stack memory accesses, the performance will be impacted negatively. Minimize cross-stack memory accesses by exploiting locality in algorithm.

MPI applications are more efficient with implicit scaling compared to an explicit scaling approach. A single MPI rank can utilize the entire root device which eliminates explicit synchronization and communication between stacks. Implicit scaling automatically overlaps local memory accesses and cross-stack memory accesses in a single kernel launch.

Implicit scaling improves kernel execution time only. Serial bottlenecks will not speed up. Applications will observe no speedup with implicit scaling if a 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 is reduced with implicit scaling. Note that only stack 0 has PCIe connection to the host. On Intel® Data Center GPU Max with implicit scaling enabled, kernel launch latency increases by about 3 microseconds.

## Work Scheduling and Memory Distribution

The root device driver uses deterministic heuristics to distribute work-groups and memory pages to all stacks when implicit scaling is used. These heuristics are described in the next two sections.

## Memory Coloring

Any allocation in SYCL/OpenMP that corresponds to a shared or device allocation is colored across all stacks, meaning that allocation is divided into number-of-stacks chunks and distributed round-robin between the stacks. Consider this root device allocation:

OpenMP:

```
int *a = (int*)omp_target_alloc( sizeof(int)*N, device_id );
```

SYCL:

```
int *a = sycl::malloc_device<int>(N, q);
```

For a 2-stack root device, the first half, (elements a[0] to a[N/2-1]), is physically allocated on stack 0. The remaining half, (elements a[N/2] to a[N-1]), is located on stack 1. In future, we will introduce memory allocation APIs that allow user-defined memory coloring.

Note:

- The memory coloring described above is applied at page size-granularity. An allocation containing three pages has two pages resident on stack 0.
- Allocations smaller than or equal to page-size are resident on stack 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 the object data to be distributed among all stacks.

## Static Partitioning

Scheduling of work-groups to stacks is deterministic and referred to as static partitioning. The partitioning follows a simple rule: the slowest moving dimension is divided in number-of-stacks chunks and distributed round-robin between stacks. Let's look the following at 1-dimensional kernel launch on the root device:

OpenMP:

```
#pragma omp target teams distribute parallel for simd
for (int i = 0; i < N; ++i)
{
    //
}
```

SYCL:

```
q.parallel_for(N, [=](auto i) {
    //
});
```

Since there is only a single dimension it is, automatically the slowest moving dimension and partitioned between stacks by the driver. For a 2-stack root device, iterations 0 to  $N/2-1$  are executed on stack 0. The remaining iterations  $N/2$  to  $N-1$  are executed on stack 1.

For OpenMP, the slowest moving dimension is the outermost loop when the `collapse` clause is used. For SYCL, the slowest moving dimension is the first element of global range. For example, consider this 3D kernel launch:

OpenMP:

```
#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)
        {
            //
        }
    }
}
```

SYCL:

```
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 moving dimension is  $z$  and is partitioned between the stacks. That is, for a 2-stack root device, all iterations from  $z = 0$  to  $z = nz/2 - 1$  are executed on stack 0. The remaining iterations from  $z = nz/2$  to  $z = nz-1$  are executed on stack 1.

In case the slowest moving dimension cannot be divided evenly between the stacks and there is a load imbalance that is larger than 5%, the driver will partition the 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 the table below (assuming local range {1,1,16}):

#### Work-Group Partition to Stacks

<b>nz</b>	<b>ny</b>	<b>nx</b>	<b>Partitioned Dimension</b>
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 SYCL, the partitioned dimension can change. For example, for global range {38,512,512} with local range {2,1,8} the driver would partition the y-dimension, while for local range {1,1,16} the driver would partition the z-dimension.

OpenMP can only have a 1-dimensional local range which is created from the innermost loop, and thus does not impact static partitioning heuristics. OpenMP kernels created with a collapse level larger than 3 correspond to a 1-dimensional kernel with all the for loops linearized. The linearized loop will be partitioned following 1D kernel launch heuristics.

Notes:

- Static partitioning happens at work-group granularity. This implies that all work-items in a work-group are scheduled to the same stack.
- A kernel with a single work-group is resident on stack 0 only.

#### STREAM Example

For a given kernel:

OpenMP:

```
int *a = (int *)omp_target_alloc(sizeof(int) * N, device_id);

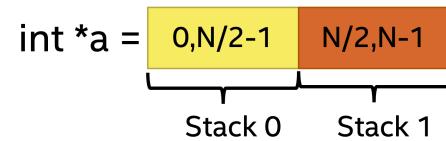
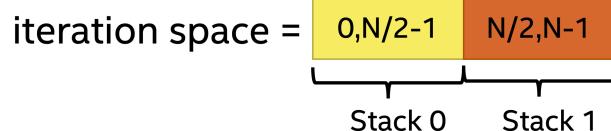
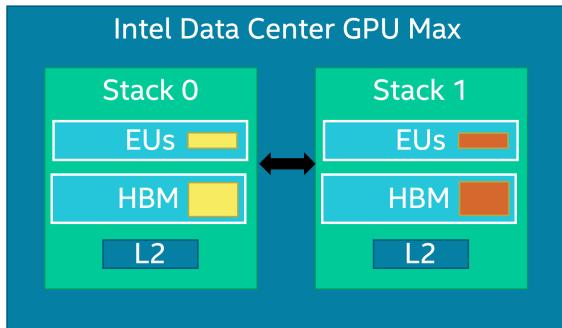
#pragma omp target teams distribute parallel for simd
for (int i = 0; i < N; ++i)
{
    a[i] = i;
}
```

SYCL:

```
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 this section, we demonstrate implicit scaling performance for STREAM benchmark using 1D and 3D kernel launches on Intel® Data Center GPU Max.

## STREAM

Consider the STREAM benchmark written in OpenMP. The main kernel is on line 44-48:

```
// Code for STREAM:
#include <iostream>
#include <omp.h>
#include <cstdint>

// compile via:
// icpx -O2 -fopenmp -fopenmp-targets=spir64 ./stream.cpp

int main()
{
    constexpr int64_t N = 256 * 1e6;
    constexpr int64_t bytes = N * sizeof(int64_t);

    int64_t *a = static_cast<int64_t *>(malloc(bytes));
    int64_t *b = static_cast<int64_t *>(malloc(bytes));
    int64_t *c = static_cast<int64_t *>(malloc(bytes));

    #pragma omp target enter data map(alloc:a[0:N])
    #pragma omp target enter data map(alloc:b[0:N])
    #pragma omp target enter data map(alloc:c[0:N])

    for (int i = 0; i < N; ++i)
    {
        a[i] = i + 1;
        b[i] = i - 1;
    }

    #pragma omp target update to(a[0:N])
    #pragma omp target update to(b[0:N])

    const int no_max_rep = 100;
    double time;
    for (int irep = 0; irep < no_max_rep + 10; ++irep)
    {
        if (irep == 10)
            time = omp_get_wtime();

        #pragma omp target teams distribute parallel for simd
    }
}
```

```

        for (int i = 0; i < N; ++i)
        {
            c[i] = a[i] + b[i];
        }
    }
    time = omp_get_wtime() - time;
    time = time / no_max_rep;

#pragma omp target update from(c[0:N])

for (int i = 0; i < N; ++i)
{
    if (c[i] != 2 * i)
    {
        std::cout << "wrong results!" << std::endl;
        exit(1);
    }
}

const int64_t streamed_bytes = 3 * N * sizeof(int64_t);

std::cout << "bandwidth = " << (streamed_bytes / time) * 1E-9
<< " GB/s" << std::endl;
}

```

In COMPOSITE mode, the benchmark runs on the entire root-device (GPU card with 2 stacks) by implicit scaling. No code changes are required. The heuristics of static partitioning and memory coloring guarantee that each stack accesses local memory only. On a 2-stack Intel® Data Center GPU Max system we measure 2x speed-up for STREAM compared to a single stack. Measured bandwidth is reported in table below.

#### Measured Bandwidth with 1D Kernel Launch

Array Size [MB]	1-stack Bandwidth [GB/s]	Implicit Scaling (2-stack) Bandwidth [GB/s]	Implicit Scaling Speed-up over 1-stack
512	1056	2074	1.96x
1024	1059	2127	2x
2048	1063	2113	1.99x

#### 3D STREAM

The STREAM benchmark can be modified to use 3D kernel launch via the `collapse` clause in OpenMP. The intent here is to show performance in case driver heuristics are used to partition the 3D kernel launches between the stacks. The kernel is on line 59-70:

```

// Code for 3D STREAM
#include <iostream>
#include <omp.h>
#include <cassert>

// compile via:
// icpx -O2 -fopenmp -fopenmp-targets=spir64 ./stream_3D.cpp

int main(int argc, char **argv)
{
    const int device_id = omp_get_default_device();

```

```
const int desired_total_size = 32 * 512 * 16384;
const std::size_t bytes = desired_total_size * sizeof(int64_t);

std::cout << "memory footprint = " << 3 * bytes * 1E-9 << " GB"
<< std::endl;

int64_t *a = static_cast<int64_t*>(omp_target_alloc_device(bytes, device_id));
int64_t *b = static_cast<int64_t*>(omp_target_alloc_device(bytes, device_id));
int64_t *c = static_cast<int64_t*>(omp_target_alloc_device(bytes, device_id));

const int min = 64;
const int max = 32768;

for (int lx = min; lx < max; lx *= 2)
{
    for (int ly = min; ly < max; ly *= 2)
    {
        for (int lz = min; lz < max; lz *= 2)
        {
            const int total_size = lx * ly * lz;
            if (total_size != desired_total_size)
                continue;

            std::cout << "lx=" << lx << " ly=" << ly << " lz="
            << lz << ", ";

            #pragma omp target teams distribute parallel for simd
            for (int i = 0; i < total_size; ++i)
            {
                a[i] = i + 1;
                b[i] = i - 1;
                c[i] = 0;
            }

            const int no_max_rep = 40;
            const int warmup = 10;
            double time;
            for (int irep = 0; irep < no_max_rep + warmup; ++irep)
            {
                if (irep == warmup) time = omp_get_wtime();

                #pragma omp target teams distribute parallel for simd collapse(3)
                for (int iz = 0; iz < lz; ++iz)
                {
                    for (int iy = 0; iy < ly; ++iy)
                    {
                        for (int ix = 0; ix < lx; ++ix)
                        {
                            const int index = ix + iy * lx + iz * lx * ly;
                            c[index] = a[index] + b[index];
                        }
                    }
                }
            }
            time = omp_get_wtime() - time;
            time = time / no_max_rep;

            const int64_t streamed_bytes = 3 * total_size * sizeof(int64_t);
```

```

        std::cout << "bandwidth = " << (streamed_bytes / time) * 1E-9
        << " GB/s" << std::endl;

#pragma omp target teams distribute parallel for simd
for (int i = 0; i < total_size; ++i)
{
    assert(c[i] == 2 * i);
}
}

omp_target_free(a, device_id);
omp_target_free(b, device_id);
omp_target_free(c, device_id);
}

```

Note that the inner-most loop has stride-1 memory access pattern. If the z- or y-loop were the innermost loop, performance would decrease due to the generation of scatter loads and stores leading to poor cache line utilization. On a 2-stack Intel® Data Center GPU Max with 2 Gigabytes array size, we measure the performance shown below.

### Measured Bandwidth with 3D Kernel Launch

<b>nx</b>	<b>ny</b>	<b>nz</b>	<b>1-stack Bandwidth [GB/s]</b>	<b>Implicit Scaling Bandwidth [GB/s]</b>	<b>Implicit Scaling Speed-up over 1-stack</b>
64	256	16834	1040	2100	2.01x
16834	64	256	1040	2077	1.99x
256	16834	64	1037	2079	2x

As described in [Static Partitioning](#), for these loop bounds the driver partitions the slowest moving dimension, i.e. the z-dimension, between both stacks. This guarantees that each stack accesses local memory only, leading to close to 2x speed-up with implicit scaling compared to using a single stack.

### Programming Principles

To achieve good performance with implicit scaling, cross-stack memory accesses must be minimized, but it is not required to eliminate all cross-stack accesses. A certain amount of cross-stack traffic can be handled by stack-to-stack interconnect if performed concurrently with local memory accesses. For a memory bandwidth bound workload the amount of acceptable cross-stack accesses is determined by the ratio of local memory bandwidth and cross-stack bandwidth (see [Cross-Stack Traffic](#)).

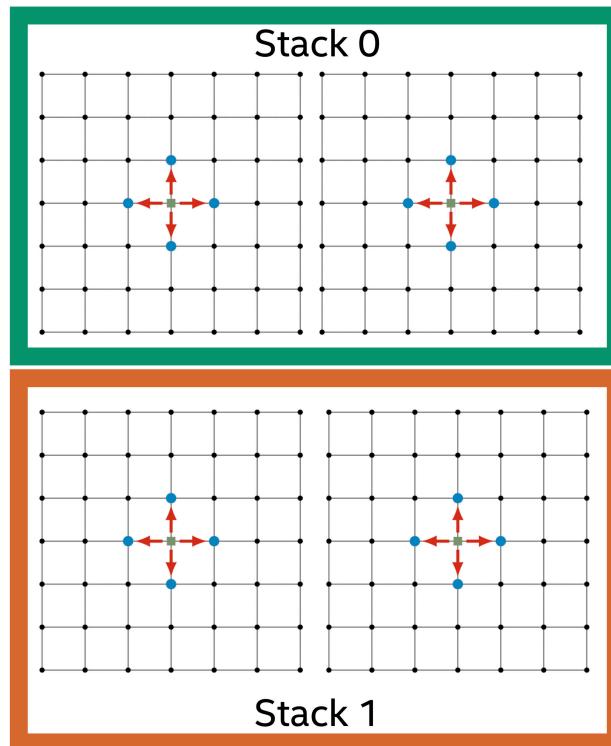
The following principles should be embraced by workloads that use implicit scaling:

- The kernel must have enough work-items to utilize both stacks.

- The minimal number of work-items needed to utilize both stacks is  $< \text{number of VEs} > * < \text{hardware-threads per VE} > * < \text{SIMD width} >$ , where VE refers to Vector Engine or Execution Unit.
- 2-stack Intel® Data Center GPU Max with 1024 VE and SIMD32 requires at least 262,144 work-items.

- Device time must dominate runtime to observe whole application scaling.
- Minimize cross-stack memory accesses by exploiting locality in algorithm.
- The slowest moving dimension should be large to avoid stack load imbalance.
- Cross-stack memory accesses and local memory accesses should be interleaved.
- Avoid stride-1 memory accesses in slowest moving dimension for 2D and 3D kernel launches.
- If the memory access pattern changes dynamically over time, a sorting step should be performed every Nth iteration to minimize cross-stack 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-stack traffic. To illustrate this concept, we use a stencil kernel as an example. A stencil operates on a grid which can be divided into blocks where the majority of stencil computations within a block use stack local data. Only stencil operations that are at the border of the block require data from another block, i.e. on another stack. The amount of these cross-stack/cross-border accesses are suppressed by halo to local volume ratio. This concept is illustrated below.



## Cross-Stack Traffic

As mentioned in the previous section, it is crucial to minimize cross-stack traffic. To guide how much traffic can be tolerated without significantly impacting application performance, we can benchmark the STREAM kernel with varying amounts of cross-stack traffic and compare to stack-local STREAM performance. The worst case is 100% cross-stack traffic. This is generated by reversing the loop order in STREAM kernel (see [STREAM](#)):

```
#pragma omp target teams distribute parallel for simd
for (int i = N - 1; i >= 0; --i)
{
    c[i] = a[i] + b[i];
}
```

Here, each stack has 100% cross-stack memory traffic as work-groups on stack-0 access array elements N-1 to N/2 which are located in stack-1 memory. This kernel essentially benchmarks stack-to-stack bi-directional bandwidth. This approach can be generalized to interpolate between 0% cross-stack accesses and 100% cross-stack accesses by the modified STREAM below:

```
// Code for cross stack stream
#include <iostream>
#include <omp.h>

// compile via:
// icpx -O2 -fopenmp -fopenmp-targets=spir64 ./stream_cross_stack.cpp
// run via:
// ZE_AFFINITY_MASK=0 ./a.out

template <int cross_stack_fraction>
void cross_stack_stream() {

    constexpr int64_t size = 256*1e6;
    constexpr int64_t bytes = size * sizeof(int64_t);

    int64_t *a = static_cast<int64_t*>(malloc( bytes ));
    int64_t *b = static_cast<int64_t*>(malloc( bytes ));
    int64_t *c = static_cast<int64_t*>(malloc( bytes ));
    #pragma omp target enter data map( alloc:a[0:size] )
    #pragma omp target enter data map( alloc:b[0:size] )
    #pragma omp target enter data map( alloc:c[0:size] )

    for ( int i = 0; i < size; ++i ) {

        a[i] = i + 1;
        b[i] = i - 1;
        c[i] = 0;
    }

    #pragma omp target update to( a[0:size] )
    #pragma omp target update to( b[0:size] )
    #pragma omp target update to( c[0:size] )

    const int num_max_rep = 100;

    double time;

    for ( int irep = 0; irep < num_max_rep+10; ++irep ) {

        if ( irep == 10 ) time = omp_get_wtime();

        #pragma omp target teams distribute parallel for simd
        for ( int j = 0; j < size; ++j ) {

            const int cache_line_id = j / 16;

            int i;

            if ( (cache_line_id%cross_stack_fraction) == 0 ) {

                i = (j+size/2)%size;
            }
            else {

```

```

        i = j;
    }

    c[i] = a[i] + b[i];
}
time = omp_get_wtime() - time;
time = time/num_max_rep;

#pragma omp target update from( c[0:size] )

for ( int i = 0; i < size; ++i ) {

    if ( c[i] != 2*i ) {

        std::cout << "wrong results!" << std::endl;
        exit(1);
    }
}

const int64_t streamed_bytes = 3 * size * sizeof(int64_t);

std::cout << "cross_stack_percent = " << (1/(double)cross_stack_fraction)*100
      << "%, bandwidth = " << (streamed_bytes/time) * 1E-9 << " GB/s" << std::endl;
}

int main() {

    cross_stack_stream< 1>();
    cross_stack_stream< 2>();
    cross_stack_stream< 4>();
    cross_stack_stream< 8>();
    cross_stack_stream<16>();
    cross_stack_stream<32>();
}

```

The kernel on line 48-65 accesses every `cross_stack_fraction`'th cache line cross-stack by offsetting array access with  $(j+N/2)\%N$ . For `cross_stack_fraction==1`, we generate 100% cross-stack memory accesses. By doubling `cross_stack_fraction` we decrease cross-stack traffic by a factor of 2. Note that this kernel is written such that cross-stack and local memory accesses are interleaved within work-groups to maximize hardware utilization. Measured performance on 2-stack Intel® Data Center GPU Max with 2 Gigabytes array size can be seen below:

#### Measured Bandwidth with Cross-Stack Accesses

<b>Partial cross-stack STREAM bandwidth [GB/s]</b>	<b>cross_stack_fraction</b>	<b>% of cross-stack accesses</b>	<b>% of max local 2-stack STREAM bandwidth</b>
355	1	100%	17%
696	2	50%	33%
1223	4	25%	58%
1450	8	12.5%	69%
1848	16	6.25%	87%

<b>Partial cross-stack STREAM bandwidth [GB/s]</b>	<b>cross_stack_fraction</b>	<b>% of cross-stack accesses</b>	<b>% of max local 2- stack STREAM bandwidth</b>
2108	32	3.125%	99%

As seen in the above table, applications should try to limit cross-stack traffic to be less than 10% of all memory traffic to avoid a significant drop in sustained memory bandwidth. For STREAM with of 12.5% cross-stack accesses we measure about 69% of the bandwidth of a local STREAM benchmark. These numbers can be used to estimate the impact of cross-stack memory accesses on application kernel execution time.

## Using Intel® oneAPI Math Kernel Library (oneMKL)

- Scaling Performance with Intel® oneAPI Math Kernel Library(oneMKL) in SYCL
- Scaling Performance with Intel® oneAPI Math Kernel Library(oneMKL) in OpenMP

### Scaling Performance with Intel® oneAPI Math Kernel Library(oneMKL) in SYCL

This section describes how to use Intel® oneAPI Math Kernel Library (oneMKL) on a platform with multiple devices to scale up performance in FLAT mode using a SYCL example.

The code below shows full example of how to split GEMM computation across 2 devices. This example demonstrates scaling for {S,D,C,Z}GEMM APIs.

```
#include "mkl.h"
#include "oneapi/mkl/blas.hpp"
#include <sycl/sycl.hpp>

// Helper functions
template <typename fp> fp set_fp_value(fp arg1, fp arg2 = 0.0) { return arg1; }

template <typename fp>
std::complex<fp> set_fp_value(std::complex<fp> arg1,
                               std::complex<fp> arg2 = 0.0) {
    return std::complex<fp>(arg1.real(), arg2.real());
}

template <typename fp>
void init_matrix(fp *M, oneapi::mkl::transpose trans, int64_t m, int64_t n,
                 int64_t ld) {
    if (trans == oneapi::mkl::transpose::nontrans) {
        for (int64_t j = 0; j < n; j++)
            for (int64_t i = 0; i < m; i++)
                M[i + j * ld] = fp(i + j);
    } else {
        for (int64_t i = 0; i < m; i++)
            for (int64_t j = 0; j < n; j++)
                M[j + i * ld] = fp(i + j);
    }
}

// GEMM operation is performed as shown below.
// C = alpha * op(A) * op(B) + beta * C

template <typename fp> void run_gemm(const sycl::device &dev) {
    // Initialize data for GEMM
    oneapi::mkl::transpose transA = oneapi::mkl::transpose::nontrans;
    oneapi::mkl::transpose transB = oneapi::mkl::transpose::nontrans;
```

```
// Matrix data sizes
int64_t m = 1024;
int64_t n = 1024;
int64_t k = 1024;

// Leading dimensions of matrices
int64_t ldA = 1024;
int64_t ldB = 1024;
int64_t ldC = 1024;

// Set scalar fp values
fp alpha = set_fp_value(2.0, -0.5);
fp beta = set_fp_value(3.0, -1.5);

// Create devices for multi-device run using
// the same platform as input device
auto devices = dev.get_platform().get_devices();
sycl::queue device0_queue;
sycl::queue device1_queue;

int64_t nb_device = devices.size();

// nb_device = 1      Example will use 1 device
// nb_device = 2      Example will use 2 devices with explicit scaling
// nb_device > 2     Example will use only 2 devices with explicit scaling

// Catch asynchronous exceptions
auto exception_handler = [] (sycl::exception_list exceptions) {
    for (std::exception_ptr const &e : exceptions) {
        try {
            std::rethrow_exception(e);
        } catch (sycl::exception const &e) {
            std::cout << "Caught asynchronous SYCL exception during GEMM:\n"
                  << e.what() << std::endl;
        }
    }
};

// Create context and execution queue
devices.push_back(dev);
sycl::context cxt(devices);
sycl::queue main_queue(cxt, dev, exception_handler);

// If two devices are detected, create queue for each device
if (nb_device > 1) {
    device0_queue = sycl::queue(cxt, devices[0], exception_handler);
    device1_queue = sycl::queue(cxt, devices[1], exception_handler);
}

// Allocate and initialize arrays for matrices
int64_t sizea, sizeb;
if (transA == oneapi::mkl::transpose::nontrans)
    sizea = ldA * k;
else
    sizea = ldA * m;
if (transB == oneapi::mkl::transpose::nontrans)
    sizeb = ldB * n;
```

```
else
    sizeb = ldB * k;
int64_t sizec = ldc * n;

auto A_host = sycl::malloc_host<fp>(sizea, main_queue);
auto B_host = sycl::malloc_host<fp>(sizeb, main_queue);
auto C_host = sycl::malloc_host<fp>(sizec, main_queue);

init_matrix(A_host, transA, m, k, lda);
init_matrix(B_host, transB, k, n, ldb);
init_matrix(C_host, oneapi::mkl::transpose::nontrans, m, n, ldc);

// Copy A/B/C from host to device(s)
// When two devices are detected,
// GEMM operation is split between devices in n direction
// All A matrix is copied to both devices.
// B and C matrices are split between devices, so only half of B and C are
// copied to each device.

fp *A_dev0, *A_dev1, *B_dev0, *B_dev1, *C_dev0, *C_dev1, *A_dev, *B_dev,
    *C_dev;

if (nb_device > 1) {
    A_dev0 = sycl::malloc_device<fp>(sizea, device0_queue);
    A_dev1 = sycl::malloc_device<fp>(sizea, device1_queue);
    B_dev0 = sycl::malloc_device<fp>(sizeb / 2, device0_queue);
    B_dev1 = sycl::malloc_device<fp>(sizeb / 2, device1_queue);
    C_dev0 = sycl::malloc_device<fp>(sizec / 2, device0_queue);
    C_dev1 = sycl::malloc_device<fp>(sizec / 2, device1_queue);
    // Entire A matrix is copied to both devices
    device0_queue.memcpy(A_dev0, A_host, sizea * sizeof(fp));
    device1_queue.memcpy(A_dev1, A_host, sizea * sizeof(fp));
    // Half of B and C are copied to each device
    device0_queue.memcpy(B_dev0, B_host, (sizeb / 2) * sizeof(fp));
    device1_queue.memcpy(B_dev1, B_host + ldB * n / 2,
        (sizeb / 2) * sizeof(fp));
    device0_queue.memcpy(C_dev0, C_host, (sizec / 2) * sizeof(fp));
    device1_queue.memcpy(C_dev1, C_host + ldc * n / 2,
        (sizec / 2) * sizeof(fp));
    device0_queue.wait();
    device1_queue.wait();
} else {
    A_dev = sycl::malloc_device<fp>(sizea, main_queue);
    B_dev = sycl::malloc_device<fp>(sizeb, main_queue);
    C_dev = sycl::malloc_device<fp>(sizec, main_queue);
    main_queue.memcpy(A_dev, A_host, sizea * sizeof(fp));
    main_queue.memcpy(B_dev, B_host, sizeb * sizeof(fp));
    main_queue.memcpy(C_dev, C_host, sizec * sizeof(fp));
    main_queue.wait();
}

// Call oneMKL GEMM API
// When two devices are detected, GEMM call is launched on each device
sycl::event gemm_done0;
sycl::event gemm_done1;
std::vector<sycl::event> gemm_dependencies;
try {
    if (nb_device > 1) {
```

```
// Split B and C for each device
int64_t n_half = n / 2;
gemm_done0 = oneapi::mkl::blas::gemm(
    device0_queue, transA, transB, m, n_half, k, alpha, A_dev0, lda,
    B_dev0, ldb, beta, C_dev0, ldc, gemm_dependencies);
gemm_done1 = oneapi::mkl::blas::gemm(
    device1_queue, transA, transB, m, n_half, k, alpha, A_dev1, lda,
    B_dev1, ldb, beta, C_dev1, ldc, gemm_dependencies);
} else {
    gemm_done0 = oneapi::mkl::blas::gemm(main_queue, transA, transB, m, n, k,
                                         alpha, A_dev, lda, B_dev, ldb, beta,
                                         C_dev, ldc, gemm_dependencies);
}
} catch (sycl::exception const &e) {
    std::cout << "\tCaught synchronous SYCL exception during GEMM:\n"
        << e.what() << std::endl;
}

// Wait for GEMM calls to finish
gemm_done0.wait();
if (nb_device > 1)
    gemm_done1.wait();

// Copy C from device(s) to host
if (nb_device > 1) {
    device0_queue.memcpy(C_host, C_dev0, (sizec / 2) * sizeof(fp));
    device1_queue.memcpy(C_host + ldc * n / 2, C_dev1,
                         (sizec / 2) * sizeof(fp));
    device0_queue.wait();
    device1_queue.wait();
} else {
    main_queue.memcpy(C_host, C_dev, sizec * sizeof(fp));
    main_queue.wait();
}

// Clean-up
free(A_host, ctxt);
free(B_host, ctxt);
free(C_host, ctxt);
if (nb_device > 1) {
    sycl::free(A_dev0, device0_queue);
    sycl::free(A_dev1, device1_queue);
    sycl::free(B_dev0, device0_queue);
    sycl::free(B_dev1, device1_queue);
    sycl::free(C_dev0, device0_queue);
    sycl::free(C_dev1, device1_queue);
} else {
    sycl::free(A_dev, main_queue);
    sycl::free(B_dev, main_queue);
    sycl::free(C_dev, main_queue);
}

if (nb_device > 1)
    std::cout << "\tGEMM operation is complete on 2 devices" << std::endl;
else
    std::cout << "\tGEMM operation is complete on 1 device" << std::endl;
}
```

```
// Main entry point for example.
// GEMM example is run on GPU for 4 data types.

int main(int argc, char **argv) {
    // Create GPU device
    sycl::device dev = sycl::device(sycl::gpu_selector_v);

    std::cout << "Running with single precision real data type:" << std::endl;
    run_gemm<float>(dev);

    std::cout << "Running with double precision real data type:" << std::endl;
    run_gemm<double>(dev);

    std::cout << "Running with single precision complex data type:" << std::endl;
    run_gemm<std::complex<float>>(dev);

    std::cout << "Running with double precision complex data type:" << std::endl;
    run_gemm<std::complex<double>>(dev);
    return 0;
}
```

In this example, first scalar and dimension parameters for GEMM are initialized.

```
// Initialize data for GEMM
oneapi::mkl::transpose transA = oneapi::mkl::transpose::nontrans;
oneapi::mkl::transpose transB = oneapi::mkl::transpose::nontrans;

// Matrix data sizes
int64_t m = 1024;
int64_t n = 1024;
int64_t k = 1024;

// Leading dimensions of matrices
int64_t ldA = 1024;
int64_t ldB = 1024;
int64_t ldC = 1024;

// Set scalar fp values
fp alpha = set_fp_value(2.0, -0.5);
fp beta = set_fp_value(3.0, -1.5);
```

Next, if available, multiple devices are created from root device and stored in SYCL vector. If the platform does not have multiple devices, root device will be used for computation. Note that, the example also demonstrates how to handle asynchronous exceptions.

```
// Create devices for multi-device run using
// the same platform as input device
auto devices = dev.get_platform().get_devices();
sycl::queue device0_queue;
sycl::queue device1_queue;

int64_t nb_device = devices.size();

// nb_device = 1      Example will use 1 device
// nb_device = 2      Example will use 2 devices with explicit scaling
// nb_device > 2      Example will use only 2 devices with explicit scaling

// Catch asynchronous exceptions
auto exception_handler = [] (sycl::exception_list exceptions) {
```

```

for (std::exception_ptr const &e : exceptions) {
    try {
        std::rethrow_exception(e);
    } catch (sycl::exception const &e) {
        std::cout << "Caught asynchronous SYCL exception during GEMM:\n"
              << e.what() << std::endl;
    }
}
};


```

Context and queues are then created. If multiple devices are detected earlier, a separate queue is created for each device.

```

// Create context and execution queue
devices.push_back(dev);
sycl::context cxt(devices);
sycl::queue main_queue(cxt, dev, exception_handler);

// If two devices are detected, create queue for each device
if (nb_device > 1) {
    device0_queue = sycl::queue(cxt, devices[0], exception_handler);
    device1_queue = sycl::queue(cxt, devices[1], exception_handler);
}


```

Before oneMKL calls are invoked, input and output matrices have to be allocated, initialized and transferred from host to device. It is important to note that, when 2 devices are detected, first half of GEMM computation is done on first device using entire A matrix and first half of B matrix and the other half of GEMM computation is done on second device using entire A matrix and other half of B matrix. Thus, matrices needs to be copied to each device accordingly.

```

// Allocate and initialize arrays for matrices
int64_t sizea, sizeb;
if (transA == oneapi::mkl::transpose::nontrans)
    sizea = ldA * k;
else
    sizea = ldA * m;
if (transB == oneapi::mkl::transpose::nontrans)
    sizeb = ldB * n;
else
    sizeb = ldB * k;
int64_t sizec = ldc * n;

auto A_host = sycl::malloc_host<fp>(sizea, main_queue);
auto B_host = sycl::malloc_host<fp>(sizeb, main_queue);
auto C_host = sycl::malloc_host<fp>(sizec, main_queue);

init_matrix(A_host, transA, m, k, ldA);
init_matrix(B_host, transB, k, n, ldB);
init_matrix(C_host, oneapi::mkl::transpose::nontrans, m, n, ldc);

// Copy A/B/C from host to device(s)
// When two devices are detected,
// GEMM operation is split between devices in n direction
// All A matrix is copied to both devices.
// B and C matrices are split between devices, so only half of B and C are
// copied to each device.

fp *A_dev0, *A_dev1, *B_dev0, *B_dev1, *C_dev0, *C_dev1, *A_dev, *B_dev,
```

```

*C_dev;

if (nb_device > 1) {
    A_dev0 = sycl::malloc_device<fp>(sizea, device0_queue);
    A_dev1 = sycl::malloc_device<fp>(sizea, device1_queue);
    B_dev0 = sycl::malloc_device<fp>(sizeb / 2, device0_queue);
    B_dev1 = sycl::malloc_device<fp>(sizeb / 2, device1_queue);
    C_dev0 = sycl::malloc_device<fp>(sizec / 2, device0_queue);
    C_dev1 = sycl::malloc_device<fp>(sizec / 2, device1_queue);
    // Entire A matrix is copied to both devices
    device0_queue.memcpy(A_dev0, A_host, sizea * sizeof(fp));
    device1_queue.memcpy(A_dev1, A_host, sizea * sizeof(fp));
    // Half of B and C are copied to each device
    device0_queue.memcpy(B_dev0, B_host, (sizeb / 2) * sizeof(fp));
    device1_queue.memcpy(B_dev1, B_host + ldB * n / 2,
                         (sizeb / 2) * sizeof(fp));
    device0_queue.memcpy(C_dev0, C_host, (sizec / 2) * sizeof(fp));
    device1_queue.memcpy(C_dev1, C_host + ldC * n / 2,
                         (sizec / 2) * sizeof(fp));
    device0_queue.wait();
    device1_queue.wait();
} else {
    A_dev = sycl::malloc_device<fp>(sizea, main_queue);
    B_dev = sycl::malloc_device<fp>(sizeb, main_queue);
    C_dev = sycl::malloc_device<fp>(sizec, main_queue);
    main_queue.memcpy(A_dev, A_host, sizea * sizeof(fp));
    main_queue.memcpy(B_dev, B_host, sizeb * sizeof(fp));
    main_queue.memcpy(C_dev, C_host, sizec * sizeof(fp));
    main_queue.wait();
}

```

Finally, oneMKL GEMM APIs are called with corresponding device(s) queues and updated C matrix is transferred to host once the GEMM calls are done.

```

// Call oneMKL GEMM API
// When two devices are detected, GEMM call is launched on each device
sycl::event gemm_done0;
sycl::event gemm_done1;
std::vector<sycl::event> gemm_dependencies;
try {
    if (nb_device > 1) {
        // Split B and C for each device
        int64_t n_half = n / 2;
        gemm_done0 = oneapi::mkl::blas::gemm(
            device0_queue, transA, transB, m, n_half, k, alpha, A_dev0, lda,
            B_dev0, ldb, beta, C_dev0, ldc, gemm_dependencies);
        gemm_done1 = oneapi::mkl::blas::gemm(
            device1_queue, transA, transB, m, n_half, k, alpha, A_dev1, lda,
            B_dev1, ldb, beta, C_dev1, ldc, gemm_dependencies);
    } else {
        gemm_done0 = oneapi::mkl::blas::gemm(main_queue, transA, transB, m, n, k,
                                              alpha, A_dev, lda, B_dev, ldb, beta,
                                              C_dev, ldc, gemm_dependencies);
    }
} catch (sycl::exception const &e) {
    std::cout << "\t\tCaught synchronous SYCL exception during GEMM:\n"
          << e.what() << std::endl;
}

```

```

// Wait for GEMM calls to finish
gemm_done0.wait();
if (nb_device > 1)
    gemm_done1.wait();

// Copy C from device(s) to host
if (nb_device > 1) {
    device0_queue.memcpy(C_host, C_dev0, (sizec / 2) * sizeof(fp));
    device1_queue.memcpy(C_host + 1dC * n / 2, C_dev1,
                         (sizec / 2) * sizeof(fp));
    device0_queue.wait();
    device1_queue.wait();
} else {
    main_queue.memcpy(C_host, C_dev, sizec * sizeof(fp));
    main_queue.wait();
}

```

To be able to run this example, the below build command can be used after setting \$MKLROOT variable.

```

icpx -fsycl -qmkl=sequential -DMKL_ILP64 -L${MKLROOT}/lib -lsycl -lOpenCL -lpthread -lm -ldl -o
onemkl_sycl_gemm onemkl_sycl_gemm.cpp
export LD_LIBRARY_PATH=${MKLROOT}/lib/:${LD_LIBRARY_PATH}
./onemkl_dpcpp_gemm

```

For large problem sizes (m=n=k=8192), close to 2X performance scaling is expected on two devices.

### Scaling Performance with Intel® oneAPI Math Kernel Library(oneMKL) in OpenMP

This section describes how to use Intel® oneAPI Math Kernel Library (oneMKL) on a platform with multiple devices to scale up performance in FLAT mode using an OpenMP offload example.

The code below shows full example of how to split AXPY computation across 2 devices. This example demonstrates scaling for `cblas_daxpy` API.

```

#include "mkl.h"
#include "mkl_omp_offload.h"
#include <omp.h>
#include <stdio.h>

int main() {

    double *x, *y, *y_ref, alpha;
    MKL_INT n, incx, incy, i;

    // Initialize data for AXPY
    alpha = 1.0;
    n = 8192;
    incx = 1;
    incy = 1;

    // Allocate and initialize arrays for vectors
    x = (double *)mkl_malloc(n * sizeof(double), 128);
    y = (double *)mkl_malloc(n * sizeof(double), 128);
    if ((x == NULL) || (y == NULL)) {
        printf("Error in vector allocation\n");
        return 1;
    }
    for (i = 0; i < n; i++) {

```

```
x[i] = rand() / (double)RAND_MAX - .5;
y[i] = rand() / (double)RAND_MAX - .5;
}

printf("First 10 elements of the output vector Y before AXPY:\n");
for (i = 0; i < 10; i++) {
    printf("%lf ", y[i]);
}
printf("\n\n");

// Detect number of available devices
int nb_device = omp_get_num_devices();

// Copy data to device and perform computation
if (omp_get_num_devices() > 1) {
    printf("2 devices are detected. AXPY operation is divided into 2 to take "
           "advantage of explicit scaling\n");
    printf("Copy x[0..%lld] and y[0..%lld] to device 0\n", n / 2 - 1,
           n / 2 - 1);
#pragma omp target data map(to
                           : x [0:n / 2]) map(tofrom
                           : y [0:n / 2]) device(0)
{
    printf("Copy x[%lld..%lld] and y[%lld..%lld] to device 1\n", n / 2, n - 1,
           n / 2, n - 1);
#pragma omp target data map(to
                           : x [n / 2:n - n / 2]) map(tofrom
                           : y [n / 2:n - n / 2]) \
device(1)
{
    double *x1 = &x[n / 2];
    double *y1 = &y[n / 2];
#pragma omp dispatch device(0) nowait
    cblas_daxpy(n / 2, alpha, x, incx, y, incy);
#pragma omp dispatch device(1)
    cblas_daxpy(n / 2, alpha, x1, incx, y1, incy);
#pragma omp taskwait
}
}
} else {
    printf("1 device is detected. Entire AXPY operation is performed on that "
           "device\n");
    printf("Copy x[0..%lld] and y[0..%lld] to device 0\n", n - 1, n - 1);
#pragma omp target data map(to : x [0:n]) map(tofrom : y [0:n]) device(0)
{
#pragma omp dispatch device(0)
    cblas_daxpy(n, alpha, x, incx, y, incy);
}
}
// End of computation

printf("\nFirst 10 elements of the output vector Y after AXPY:\n");
for (i = 0; i < 10; i++) {
    printf("%lf ", y[i]);
}
printf("\n");

mkl_free(x);
```

```
mkl_free(y);
return 0;
}
```

In this example, first AXPY parameters and vectors are allocated and initialized.

```
// Initialize data for AXPY
alpha = 1.0;
n = 8192;
incx = 1;
incy = 1;

// Allocate and initialize arrays for vectors
x = (double *)mkl_malloc(n * sizeof(double), 128);
y = (double *)mkl_malloc(n * sizeof(double), 128);
if ((x == NULL) || (y == NULL)) {
    printf("Error in vector allocation\n");
    return 1;
}
for (i = 0; i < n; i++) {
    x[i] = rand() / (double)RAND_MAX - .5;
    y[i] = rand() / (double)RAND_MAX - .5;
}

printf("First 10 elements of the output vector Y before AXPY:\n");
for (i = 0; i < 10; i++) {
    printf("%lf ", y[i]);
}
printf("\n\n");
```

Next, number of available devices is detected using `omp_get_num_devices` API.

```
// Detect number of available devices
int nb_device = omp_get_num_devices();
```

Finally, data is transferred and oneMKL `cblas_daxpy` calls are offloaded to GPU using OpenMP. Note that, if 2 devices are detected, half of `x` and `y` vectors are copied to each device and AXPY operation is divided into 2 to take advantage of scaling.

```
// Copy data to device and perform computation
if (omp_get_num_devices() > 1) {
    printf("2 devices are detected. AXPY operation is divided into 2 to take "
           "advantage of explicit scaling\n");
    printf("Copy x[0..%lld] and y[0..%lld] to device 0\n", n / 2 - 1,
           n / 2 - 1);
#pragma omp target data map(to
                           : x [0:n / 2]) map(tofrom
                           : y [0:n / 2]) device(0)
    {
        printf("Copy x[%lld..%lld] and y[%lld..%lld] to device 1\n", n / 2, n - 1,
               n / 2, n - 1);
#pragma omp target data map(to
                           : x [n / 2:n - n / 2]) map(tofrom
                           : y [n / 2:n - n / 2]) device(1)
        {
            double *x1 = &x[n / 2];
            double *y1 = &y[n / 2];
```

```

#pragma omp dispatch device(0) nowait
    cblas_daxpy(n / 2, alpha, x, incx, y, incy);
#pragma omp dispatch device(1)
    cblas_daxpy(n / 2, alpha, x1, incx, y1, incy);
#pragma omp taskwait
}
}
} else {
    printf("1 device is detected. Entire AXPY operation is performed on that "
        "device\n");
    printf("Copy x[0..%lld] and y[0..%lld] to device 0\n", n - 1, n - 1);
#pragma omp target data map(to : x [0:n]) map(tofrom : y [0:n]) device(0)
{
#pragma omp dispatch device(0)
    cblas_daxpy(n, alpha, x, incx, y, incy);
}
}

```

To be able to run this example, the below build command can be used after setting \$MKLROOT variable.

```

icpx -fiopenmp -fopenmp-targets=spir64 -m64 -DMKL_ILP64 -qmkl-ilp64=parallel -lstdc++ -o
onemkl_openmp_axpy onemkl_openmp_axpy.cpp
export LD_LIBRARY_PATH=${MKLROOT}/lib/:${LD_LIBRARY_PATH}
./onemkl_openmp_axpy

```

For large problem sizes (m=150000000), close to 2X performance scaling is expected on two devices.

## Using Intel® MPI Library

GPUs have become common in HPC clusters and the Intel® MPI Library now supports scale-up and scale-out executions of large applications on such heterogeneous machines.

GPU support was first introduced in Intel® MPI Library 2019 U8 ([Intel® MPI Library Release Notes](#)). Several enhancements have since been added in this direction. Unless otherwise stated, all the tests in this section used Intel® MPI Library 2021.8.

- [Running MPI Applications on GPUs](#)
- [Intra-Device and Inter-Device Data Transfers for MPI+OpenMP Programs](#)

## References

### 1. Intel® MPI Library Release Notes

## Running MPI Applications on GPUs

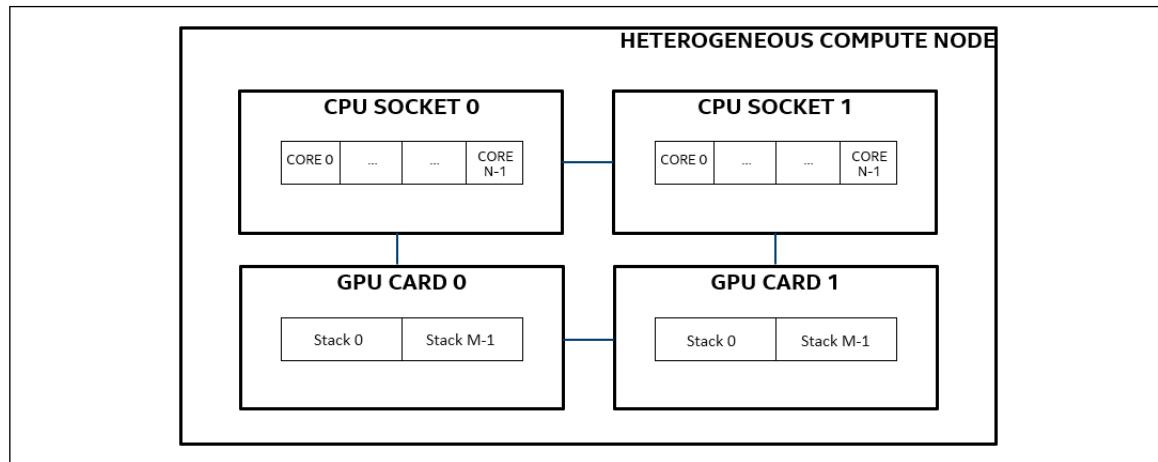
This section covers the following GPU-specific details of Intel® MPI Library:

- Topology Detection
- Execution Models
- GPU Pinning
- Multi-Stack, Multi-GPU and Multi-Node Support
- GPU Benchmarks in Intel® MPI Benchmarks
- Profiling MPI Applications on GPUs
- Recommendations for better performance

## Topology Detection

The figure below illustrates a typical GPU compute node, with two CPU sockets. Each CPU socket has N cores and is connected to a single GPU card with M stacks, making a total of  $2N$  cores and  $2M$  stacks per node.

### Typical GPU compute node



Variations of the above topology are possible. Use the `cpuinfo -g` command to see details:

```
$ cpuinfo -g

Intel(R) processor family information utility, Version 2021.11 Build
20231005 (id: 74c4a23)

Copyright (C) 2005-2021 Intel Corporation. All rights reserved.

===== Processor composition =====
Processor name : Intel(R) Xeon(R) Gold 8480+
Packages(sockets) : 2
Cores : 112
Processors(CPUs) : 224
Cores per package : 56
Threads per core : 2
```

In this example, we have a dual socket node with 56 cores per socket, making a total of 112 physical cores per node. To further understand the GPU topology, one may use Intel® MPI Library's debug output. To activate this, run your application with two additional environment variables, i.e. `I_MPI_DEBUG = 3` or above and `I_MPI_OFFLOAD = 1`.

```
[0] MPI startup(): ===== GPU topology on host1 =====
[0] MPI startup(): NUMA Id GPU Id Stacks Ranks on this NUMA
[0] MPI startup(): 0      0      (0,1)  0,1
[0] MPI startup(): 1      1      (2,3)  2,3
```

The above truncated output shows that there are 2 GPUs and 2 stacks per GPU. Stacks 0 and 1 belong to GPU 0, while stacks 2 and 3 belong to GPU 1.

## Execution Models

Developers may use the Intel® MPI Library in a GPU environment in many ways. Some of the most common scenarios are described below.

Naïve

In this model, the MPI functions are executed on the host (CPU for all practical purposes) alone and the underlying MPI library is not required to be device (here, GPU) aware. As shown in the figure below, developers are responsible for taking care of data movements between the host and device (use of `to:` and `tofrom:`). In this model, applications making frequent MPI communications might suffer from excessive data movements between the host and device. However, this approach might still be useful for very small message sizes.

### Naïve execution model using OpenMP

```
// Mark data to be copied from host to device and device to host
#pragma omp target data map(to: rank, num_values) map(tofrom:values[0:num_values])
{
    // Compute on GPU
    #pragma omp target parallel for
    for (i = 0; i < num_values; ++i) {
        values[i] = values[i] + rank + 1;
        printf("[Within GPU:] values[%d]=%d \n", i, values[i]);
    }
    //Send device buffer to rank 0 after copy back from GPU
    MPI_Send(values, num_values, MPI_INT, dest_rank, tag, MPI_COMM_WORLD);
}
```

Please note that `MPI_Send` appears outside the scope of both offload pragmas in the previous figure. Both SYCL and OpenMP offload constructs may be used for the required data movements between the host and device.

### GPU Aware MPI

In this model, the underlying MPI library is aware of GPU buffers, hence developers are relieved from the task of explicit data movement between the host and device. However, to pass a pointer of an offloadable memory region to MPI, developers may need to use specific compiler directives or get it from corresponding acceleration runtime API. Apart from moving the data as required, The Intel® MPI Library also optimizes data movement between the host and device to hide the data transfer costs as much as possible. The MPI calls still run on the host. A new environment variable called `I_MPI_OFFLOAD` was introduced for GPU buffer support. As shown in the following figure, `MPI_Send` appears outside the scope of `omp target parallel for` pragma but inside the scope of the target pragma responsible for data movement. Also note that the pointer `values` is marked using the `use_device_ptr` and `is_device_ptr` keywords to enable `values` pointer to be correctly processed by the Intel® MPI Library. GPU aware features of Intel® MPI Library can be used from within both SYCL and OpenMP-offload based applications.

### GPU aware execution model using OpenMP Offload

```
// Mark data to be copied from host to device
#pragma omp target data map(to: rank, values[0:num_values], num_values) use_device_ptr(values)
{
    // Compute on GPU
    #pragma omp target parallel for is_device_ptr(values)
    for (i = 0; i < num_values; ++i) {
        values[i] = values[i] + rank + 1;
        printf("[Within GPU:] values[%d]=%d \n", i, values[i]);
    }

    // Send device buffer to rank 0 without the use of from:values
    MPI_Send(values, num_values, MPI_INT, dest_rank, tag, MPI_COMM_WORLD);
}
```

### Build command

```
$ mpiicx -qopenmp -fopenmp-targets=spir64 test.c -o test
```

where, `-fopenmp` enables a middle-end that supports the transformation of OpenMP in LLVM; and `-fopenmp-targets=spir64` enables the compiler to generate a `x86 + SPIR64` fat binary for the GPU device binary generation.

## Run command

```
$ export I_MPI_OFFLOAD=1
$ mpiexec.hydra -n 2 ./test
```

`I_MPI_OFFLOAD` controls handling of device buffers in MPI functions. A value of 1 enables handling of device buffers under the assumption that `libz3_loader.so` is already loaded. If the library is not already loaded (which is unexpected), then GPU buffer support will not be enabled. To enforce Intel® MPI Library to load `libz3_loader.so` `I_MPI_OFFLOAD` should be set to 2.

## Kernel Based Collectives

Unlike the *GPU aware MPI* model, this model supports execution of MPI scale-up kernels by the device directly. This approach helps eliminate data transfers between the host and device that were made owing to the execution of MPI calls on the host. MPI-like implementations like Intel® oneAPI Collective Communications Library ([Intel® oneCCL](#)) already support such functionalities for collective calls.

## GPU Pinning

For a heterogeneous (CPU + GPU) MPI application, apart from allocating CPU cores to MPI ranks, it is also necessary to allocate available GPU resources to ranks. In the context of the Intel® MPI Library, by default the base unit of work on Intel's discrete GPU is assumed to be a stack. Many environment variables have been introduced to achieve user desired pinning schemes. By default the Intel® MPI Library enforces a balanced pinning scheme, i.e., if the number of ranks per node are equal to the number of stacks, then 1 rank is assigned per stack. In case of oversubscription, a round robin scheme is used for the best possible balanced distribution. The Intel® MPI Library needs GPU topology awareness to enforce GPU pinning and this is achieved with the help of the [Intel® oneAPI Level Zero](#). The following environment variable enables the use of Level Zero for topology detection,

```
$ export I_MPI_OFFLOAD_TOPOLIB=level_zero
```

Since Intel® MPI Library 2021 U6, setting `I_MPI_OFFLOAD > 0` automatically sets `I_MPI_OFFLOAD_TOPOLIB` to `level_zero`. Further, in scenarios where GPU pinning needs to be disabled completely, the `I_MPI_OFFLOAD_PIN` environment variable is available to switch this support as needed.

At debug level 3 (i.e. `I_MPI_DEBUG=3`), Intel® MPI Library also prints the GPU pinning scheme (in addition to GPU topology) used at runtime, for the first host only. For topology and debug information from all hosts/nodes, use a debug level of 120 or above.

```
$ export I_MPI_DEBUG=3
```

Following is a sample debug output from an application running with 4 ranks on a node with four stacks:

```
[0] MPI startup(): ===== GPU topology on host1 =====
[0] MPI startup(): NUMA Id GPU Id Stacks Ranks on this NUMA
[0] MPI startup(): 0      0      (0,1)  0,1
[0] MPI startup(): 1      1      (2,3)  2,3

[0] MPI startup(): ===== GPU pinning on host1 =====
[0] MPI startup(): Rank Pin stack
[0] MPI startup(): 0      {0}
[0] MPI startup(): 1      {1}
[0] MPI startup(): 2      {2}
[0] MPI startup(): 3      {3}
```

The information under “GPU topology on host1” section should not be treated as GPU pinning information (the reason becomes more evident in the example below, where MPI ranks are pinned to GPU 0 only). Rather this section presents how the ranks are placed on NUMA nodes. For a node with just 1 NUMA domain per socket, NUMA IDs can be treated as CPU sockets. Under this assumption, one can interpret the GPU topology as follows:

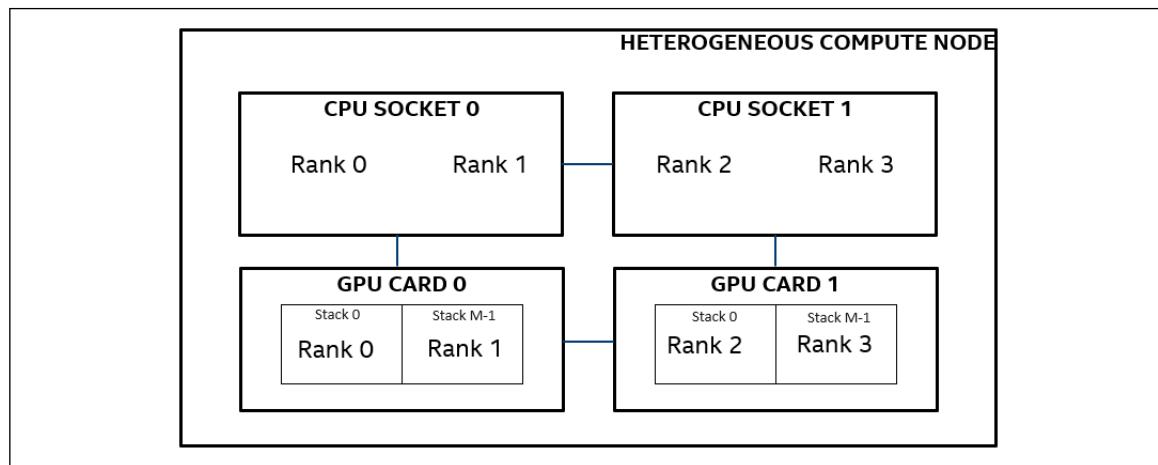
- GPU 0 is connected to CPU socket 0.
- GPU 0 contains stacks 0 and 1.
- Ranks 0 and 1 are placed on CPU socket 0.

Similarly,

- GPU 1 is connected to CPU socket 1.
- GPU 1 contains stacks 2 and 3.
- Ranks 2 and 3 are placed on CPU socket 1.

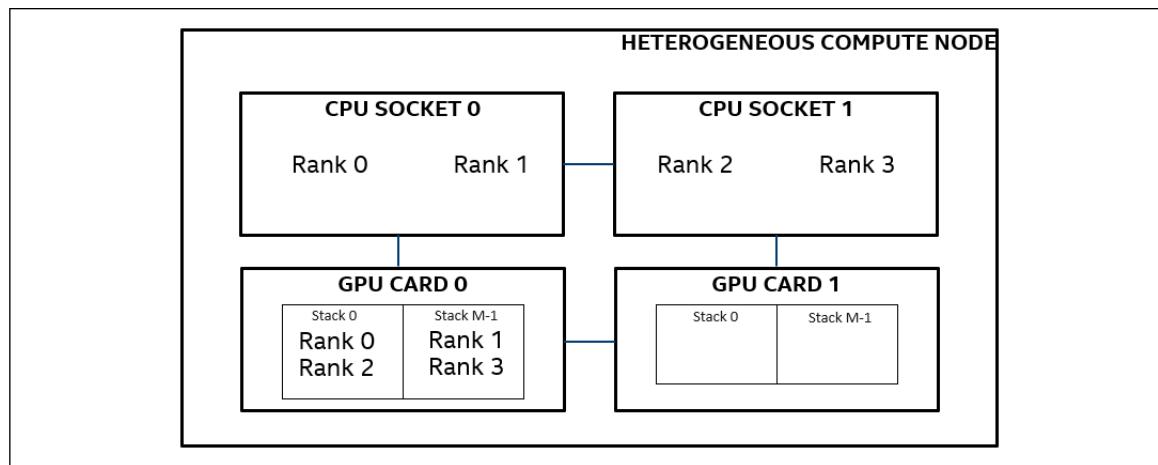
The information under “GPU Pinning on host1” clarifies the rank to GPU stack pinning. Here, each rank is pinned to a unique stack. The same is also presented in the following figure.

#### A pictorial representation of the default pinning scheme



For more clarity, let's consider another example. For the node depicted in the figure above, let's assume that one would like to restrict execution to GPU 0 alone. This can be achieved using the following environment variable:

#### A pictorial representation of a user defined pinning scheme



```
$ export I_MPI_OFFLOAD_DEVICES=0
```

As shown in the debug level 3 output (below), ranks 0, 1 and ranks 2, 3 reside on CPU sockets 0 and 1 respectively. However, since `I_MPI_OFFLOAD_DEVICES` was set to 0, all the ranks were pinned in a round robin manner to stacks of GPU 0 only. Please refer to above figure for a pictorial representation of the same.

```
[0] MPI startup(): ===== GPU topology on host1 =====
[0] MPI startup(): NUMA Id GPU Id Stacks Ranks on this NUMA
[0] MPI startup(): 0      0      (0,1)  0,1
[0] MPI startup(): 1      1      (2,3)  2,3
[0] MPI startup(): ===== GPU pinning on host1 =====
[0] MPI startup(): Rank Pin stack
[0] MPI startup(): 0      {0}
[0] MPI startup(): 1      {1}
[0] MPI startup(): 2      {0}
[0] MPI startup(): 3      {1}
```

Like the `I_MPI_PIN_PROCESSOR_LIST` environment variable available for defining a custom pinning scheme on CPU cores, a similar variable `I_MPI_OFFLOAD_DEVICE_LIST` is also available for GPU pinning. For more fine-grained pinning control, more variables like `I_MPI_OFFLOAD_CELL`, `I_MPI_OFFLOAD_DOMAIN_SIZE`, `I_MPI_OFFLOAD_DOMAIN`, etc. are available. Please refer to [GPU Pinning](#) for more details.

## Multi-Stack, Multi-GPU and Multi-Node support

Modern heterogeneous (CPU + GPU) clusters present various GPU-specific scaling possibilities:

1. Stack scaling: scaling across stacks within the same GPU card (intra-GPU, inter-stack).
2. Scale-up: Scaling across multiple GPU cards connected to the same node (intra-node, inter-GPU).
3. Scale-out: Scaling across GPU cards connected to different nodes (inter-GPU, inter-node).

All of the above scenarios are transparently handled by the Intel® MPI Library and application developers are not required to make additional source changes for this purpose. Sample command lines for each scenario are presented below.

**Baseline: All ranks run on stack 0**

```
$ I_MPI_OFFLOAD_DEVICE_LIST=0 I_MPI_DEBUG=3 I_MPI_OFFLOAD=1
mpieexec.hydra -n 2 ./mpi-binary

[0] MPI startup(): ===== GPU topology on host1 =====
[0] MPI startup(): NUMA Id GPU Id Stacks Ranks on this NUMA
[0] MPI startup(): 0 0 (0,1) 0
[0] MPI startup(): 1 1 (2,3) 1
[0] MPI startup(): ===== GPU pinning on host1 =====
[0] MPI startup(): Rank Pin stack
[0] MPI startup(): 0 {0}
[0] MPI startup(): 1 {0}
```

**Stack scaling: 1 rank per stack on GPU 0**

```
$ I_MPI_OFFLOAD_DEVICE_LIST=0,1 I_MPI_DEBUG=3
I_MPI_OFFLOAD=1 mpieexec.hydra -n 2 ./mpi-binary

[0] MPI startup(): ===== GPU topology on host1 =====
[0] MPI startup(): NUMA Id GPU Id Stacks Ranks on this NUMA
[0] MPI startup(): 0 0 (0,1) 0
[0] MPI startup(): 1 1 (2,3) 1
[0] MPI startup(): ===== GPU pinning on host1 =====
[0] MPI startup(): Rank Pin stack
[0] MPI startup(): 0 {0}
[0] MPI startup(): 1 {1}
```

Scale-up: 1 rank per stack on GPU 0 and GPU 1

```
$ I_MPI_DEBUG=3 I_MPI_OFFLOAD=1
mpiexec.hydra -n 4 ./mpi-binary

[0] MPI startup(): ===== GPU topology on host1 =====
[0] MPI startup(): NUMA Id GPU Id Stacks Ranks on this NUMA
[0] MPI startup(): 0 0 (0,1) 0,1
[0] MPI startup(): 1 1 (2,3) 2,3
[0] MPI startup(): ===== GPU pinning on host1 =====
[0] MPI startup(): Rank Pin stack
[0] MPI startup(): 0 {0}
[0] MPI startup(): 1 {1}
[0] MPI startup(): 2 {2}
[0] MPI startup(): 3 {3}
```

Scale-out: 1 rank per stack on GPUs 0, 1 of host1, GPUs 2, 3 of host2

```
$ I_MPI_DEBUG=3 I_MPI_OFFLOAD=1
mpiexec.hydra -n 8 -ppn 4 -hosts host1,host2 ./mpi-binary

[0] MPI startup(): ===== GPU topology on host1 =====
[0] MPI startup(): NUMA Id GPU Id Stacks Ranks on this NUMA
[0] MPI startup(): 0 0 (0,1) 0,1
[0] MPI startup(): 1 1 (2,3) 2,3
[0] MPI startup(): ===== GPU pinning on host1 =====
[0] MPI startup(): Rank Pin stack
[0] MPI startup(): 0 {0}
[0] MPI startup(): 1 {1}
[0] MPI startup(): 2 {2}
[0] MPI startup(): 3 {3}
```

Similar pinning occurred on host2 as well (not shown here).

## Considerations in FLAT mode

As described in the earlier sections on device hierarchy, the current GPU drivers offer **FLAT** mode as default. Using the `ZE_FLAT_DEVICE_HIERARCHY` environment variable it is possible to toggle between **FLAT** and **COMPOSITE** modes. In the **FLAT** mode, the node topology presented by Intel® MPI Library is not in alignment with other tools like `sycl-ls` or `clinfo`. This is not a bug but simply a design choice within Intel® MPI Library which selects GPU stack as the base unit of compute and always sees the true node topology, independent of the selected modes. This results in uniform distribution of ranks across available stacks, which maximizes overall system utilization.

Following is an example from a system with 4 Intel(R) Data Center GPU Max 1550 GPUs in **FLAT** mode, where `sycl-ls` enumerates 8 GPU devices, while Intel® MPI Library enumerates 4 GPU devices.

```
$ sycl-ls | grep GPU
[ext_oneyapi_level_zero:gpu:0] Intel(R) Level-Zero, Intel(R) Data Center GPU Max 1550 1.3
[1.3.27191]
[ext_oneyapi_level_zero:gpu:1] Intel(R) Level-Zero, Intel(R) Data Center GPU Max 1550 1.3
[1.3.27191]
[ext_oneyapi_level_zero:gpu:2] Intel(R) Level-Zero, Intel(R) Data Center GPU Max 1550 1.3
[1.3.27191]
[ext_oneyapi_level_zero:gpu:3] Intel(R) Level-Zero, Intel(R) Data Center GPU Max 1550 1.3
[1.3.27191]
[ext_oneyapi_level_zero:gpu:4] Intel(R) Level-Zero, Intel(R) Data Center GPU Max 1550 1.3
[1.3.27191]
[ext_oneyapi_level_zero:gpu:5] Intel(R) Level-Zero, Intel(R) Data Center GPU Max 1550 1.3
[1.3.27191]
```

```
[ext_oneyapi_level_zero:gpu:6] Intel(R) Level-Zero, Intel(R) Data Center GPU Max 1550 1.3
[1.3.27191]
[ext_oneyapi_level_zero:gpu:7] Intel(R) Level-Zero, Intel(R) Data Center GPU Max 1550 1.3
[1.3.27191]

$ I_MPI_DEBUG=3 I_MPI_OFFLOAD=1 mpirun -n 8 IMB-MPI1-GPU
[0] MPI startup(): ===== GPU topology on host1 =====
[0] MPI startup(): NUMA nodes : 2
[0] MPI startup(): GPUs      : 4
[0] MPI startup(): Tiles     : 8
[0] MPI startup(): NUMA Id    GPU Id       Tiles          Ranks on this NUMA
[0] MPI startup(): 0         0,1           (0,1)(2,3)    0,1,2,3
[0] MPI startup(): 1         2,3           (4,5)(6,7)    4,5,6,7
```

In FLAT mode, `sycl-ls` enumerates 8 GPUs, i.e. each stack as a separate device (`gpu:0` to `gpu:7`), while Intel® MPI Library shows 4 GPUs or 8 Stacks (Tiles). In COMPOSITE mode, there is no discrepancy between the outputs from Intel® MPI Library and `sycl-ls` or `clinfo`.

Please also note that the behavior of Intel® MPI Library, with regards to topology detection, pinning and performance does not change in FLAT and COMPOSITE modes.

## A note on environment variables

In the workflow we have used so far, multiple components like Level Zero backend, compiler runtime, Intel® MPI Library runtime, etc. are in action at the same time. Each component provides a number of environment variables (knobs) to control the runtime environment based on user needs. Sometimes it is possible that conflicting knob settings from different components get made. To deal with such situations, certain rules are set in place, which ensures that one will have precedence over the other. A common scenario that one may come across, is using conflicting affinity knobs from Level Zero backend (for e.g. `ZE_AFFINITY_MASK`) and Intel® MPI Library (for e.g. `I_MPI_OFFLOAD_CELL_LIST`). In such a scenario, the Level Zero knobs will have a greater priority over Intel® MPI Library knobs. It is also possible to have scenarios where conflicting environment variables are hard to resolve, and a runtime error gets generated in such cases. It is therefore a good practice to consult to the product documentations to better understand the environment variables being used. Also, not all environment variables are a mandate, but rather a request to the runtimes involved, and it is a good practice to generate and analyze the debug outputs to ensure that the requested settings were obeyed.

## GPU Benchmarks in Intel® MPI Benchmarks

The Intel® MPI Benchmarks perform a set of measurements for point-to-point and collective communication operations for a range of message sizes. The generated benchmark data helps assess the performance characteristics of a cluster and efficiency of the MPI implementation. Traditionally, IMB benchmarks ran on CPU only, GPU support was recently introduced. A pre-built binary called `IMB-MPI1-GPU` is shipped along with the Intel® MPI Library package and can be found in `$I_MPI_ROOT/bin` folder on a Linux machine. This binary can run in the *GPU aware MPI* model described earlier.

The following is an example command line for running `IMB-MPI1-GPU` for the default 0 - 4 MB message size range with 200 iterations per message size on a single node with 2 GPU cards. In this case, just the `MPI_Allreduce` function was executed. Many other MPI functions are also supported.

```
$ I_MPI_DEBUG=3 I_MPI_OFFLOAD=1
mpieexec.hydra -n 4 -ppn 4
IMB-MPI1-GPU allreduce -mem_alloc_type device -npmin 4 -iter 200
-iter_policy off
```

The `-mem_alloc_type` flag controls where the MPI buffers reside. Since Intel® MPI Library 2021.4, the default value for `-mem_alloc_type` is `device`.

`-npmin 4` restricts the minimum number of MPI ranks for IMB to 4 (if not, the target number of MPI ranks as specified by `-n` is reached in an incremental manner).

`-iter 200` requests the number of iterations per message size to 200.

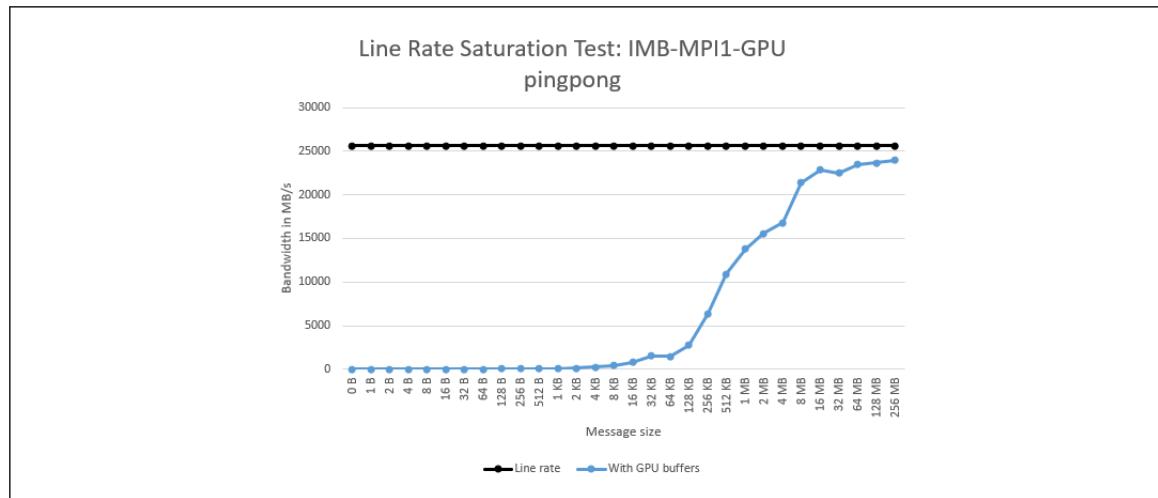
`-iter_policy off` switches off the auto reduction of iterations for large message sizes. In other words, `-iter_policy off` ensure that `-iter 200` request is obeyed for all message sizes.

### Line Saturation Test

In this sub-section we demonstrate Intel® MPI Library's ability to saturate the interconnect bandwidth when GPU buffers are employed. Tests were run on two nodes, each with a dual socket Intel® Xeon® Platinum 8480+ processor and 2 Intel® Data Center GPU Max 1550 (Code name: Ponte Vecchio). The nodes were connected using Mellanox Quantum HDR interconnect which is rated at 200 Gb/s or 25 GB/s. A total of 2 MPI ranks were launched, i.e. 1 rank per node and the PingPong test was selected from the IMB-MPI1-GPU binary as shown in the following command:

```
$ I_MPI_DEBUG=3 I_MPI_OFFLOAD=1 mpiexec.hydra -n 2 -ppn 1
  -hosts host1,host2 IMB-MPI1-GPU pingpong -iter 100 -iter_policy off
  -msglog 0:28
```

**IMB based line rate saturation test using GPU buffers. Many factors affect performance. Performance differs with different hardware and software configurations. Your measured performance can be different from our measurements.**



Intel® MPI Library 2021.8 was used for this line saturation test. Peak bandwidth of 23,948.98 MB/s was achieved which corresponds to a line rate efficiency of 93.6%.

### Profiling MPI Applications on GPUs

Tools like Intel® VTune™ Profiler, Intel® Advisor, Intel® Inspector, etc. have been enhanced to work on Intel GPUs. As was the case with CPU profiling, these tools continue to be available for MPI applications in the context of GPUs. Some of the capabilities shall be presented here in the context of MPI applications.

The IMB-MPI1-GPU benchmark of the previous section will be considered here. One way to confirm if the MPI buffers were really allocated on the GPU memory is to check for PCIe traffic, which is bound to increase (versus the `-mem_alloc_type CPU` case) since the GPUs are connected to the CPU via PCIe. Intel® VTune™ Profiler's Input and Output Analysis captures and displays several PCIe metrics.

Several modes of working with Intel® VTune™ Profiler on remote machines are available. Here the profiling data was collected from the command line interface of a remote machine. The result folder was then moved to a local machine for viewing on a local GUI. The following command lines were executed on the remote machine which hosted the GPU node.

## Buffer allocation on CPU

```
$ I_MPI_OFFLOAD=1 mpiexec.hydra  
-n 4 -ppn 4 -f hostfile -gtool "vtune -collect io -r ./vtune_data1:0"  
IMB-MPI1-GPU allreduce -npmin 4 -iter 200 -iter_policy off  
-mem_alloc_type CPU
```

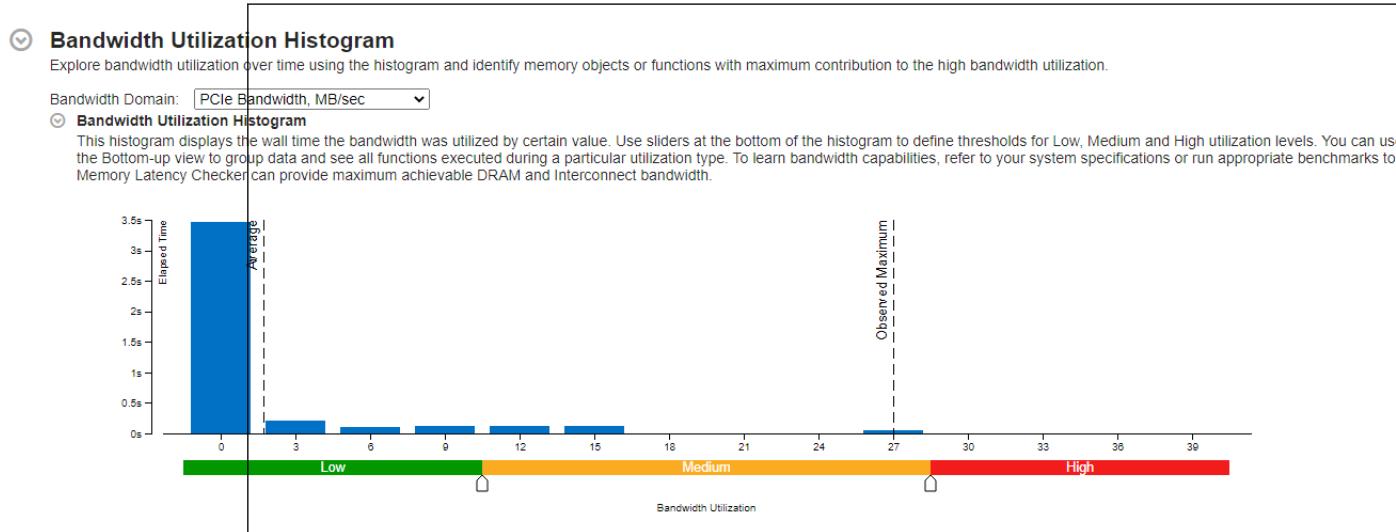
## Buffer allocation on GPU

```
$ I_MPI_OFFLOAD=1 mpiexec.hydra  
-n 4 -ppn 4 -f hostfile -gtool "vtune -collect io -r  
./vtune_data2:0" IMB-MPI1-GPU allreduce -npmin 4 -iter 200  
-iter_policy off
```

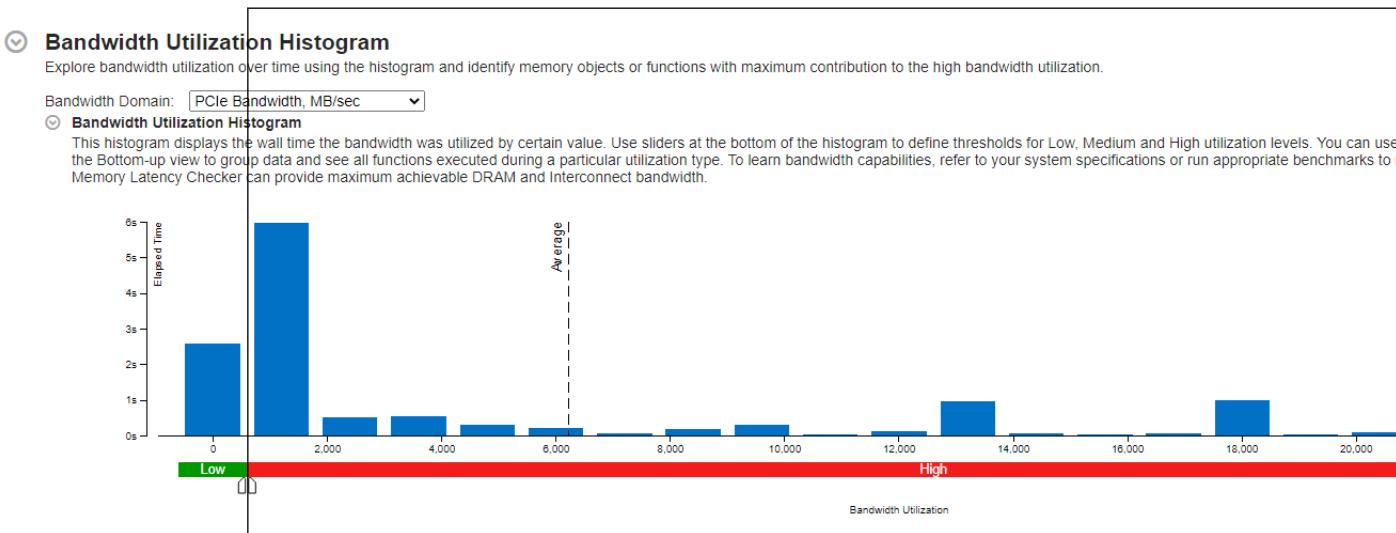
Here the `-gtool` flag was used to selectively launch VTune on rank 0 only (as specified by the last two arguments to `-gtool`, i.e. `:0`).

As can be observed from the Bandwidth Utilization Histogram in the Summary tab of the Input and Output Analysis (following figures), PCIe traffic is significantly higher in the case where the MPI buffers were allocated on the device. The observed maximum bandwidth in the `-mem_alloc_type` device case was 24,020 MB/s versus 27 MB/s for the CPU buffer case.

### Low PCIe bandwidth utilization with buffer allocation on CPU



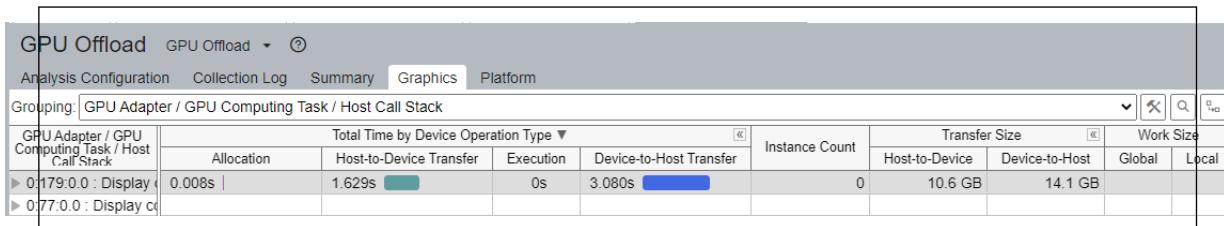
### High PCIe bandwidth utilization with buffer allocation on GPU



To see the amount of data transferred between host and device one may use Intel® VTune™ Profiler's GPU Offload analysis. Here Host-to-Device and Device-to-Host transfer times and transfer volumes are visible under the Graphics tab, as shown in the next figure. The following command line was used for running the GPU Offload analysis:

```
$ I_MPI_DEBUG=3 I_MPI_OFFLOAD=1
mpixexec.hydra -n 4 -ppn 4 -f
hostfile -gtool "vtune -collect gpu-offload -knob
collect-host-gpu-bandwidth=true -r ./vtune_data3:0" IMB-MPI1-GPU
allreduce -npmin 4 -iter 200 -iter_policy off
```

## GPU offload analysis



The purpose of this sub-section was to demonstrate the use of analysis tools in the context of MPI applications running in a heterogeneous environment. For more details on the various GPU specific capabilities of Intel® VTune™ Profiler, please refer to [Intel® VTune™ Profiler User Guide](#). One can use analysis/debug tools like Intel® Advisor, Intel® Inspector, GDB, etc. with MPI applications using the `-gtool` flag as demonstrated earlier. Please refer to their respective user guides [Intel® Advisor User Guide](#), [Intel® Inspector User Guide for Linux](#) for more details.

## Recommendations

1. It is a good practice to reuse the same GPU buffers in MPI communications as much as possible. Buffer creation in the Intel® MPI Library is an expensive operation and unnecessary allocation and deallocation of buffers must be avoided.
2. If your application uses several GPU buffers for non-blocking point-to-point operations, it is possible to increase the number of parallel buffers used by Intel® MPI Library for better performance. An environment variable named `I_MPI_OFFLOAD_BUFFER_NUM` is available for this purpose. Its default value is 4, i.e. Intel® MPI Library can handle 4 user's device buffers in parallel. However, increasing the value of this variable also increases memory consumption.
3. If your application uses large device buffers, then increasing the size of the scratch buffers used by Intel® MPI Library might improve performance in some cases. An environment variable named `I_MPI_OFFLOAD_BUFFER_SIZE` is available for this purpose. Its default value is 8 MB. Increasing the value of this variable will also result in increased memory consumption.
4. In the GPU aware model, Intel® MPI Library uses pipelining for better performance. This technique, however, is better suited for large message sizes and an environment variable called `I_MPI_OFFLOAD_PIPELINE_THRESHOLD` is available to control when pipelining algorithm is activated. The default value for this variable is 524288 bytes. Users may optimize this control for the value best suited in their environments.
5. Another variable called `I_MPI_OFFLOAD_CACHE_TOTAL_MAX_SIZE` is available to increase the size of all internal caches used by Intel® MPI Library. Its default value is 67108864 bytes. If an application uses many GPU buffers, it might not be possible to cache all of them and increasing the value of this variable can potentially improve performance.

## References

1. Intel® MPI Library Release Notes
2. Intel® oneAPI Level Zero
3. Intel® oneCCL
4. GPU Pinning
5. Intel® MPI Benchmarks
6. Intel® VTune™ Profiler User Guide
7. Intel® Advisor User Guide
8. Intel® Inspector User Guide for Linux

## Object Missing

This object is not available in the repository.

## Advanced Topics

### Compute Command Streamers (CCSs)

Each stack of the Intel® Data Center GPU Max contains 4 Compute Command Streamers (CCSs), which can be used to access a pool of Execution Units (EUs).

Hardware allows for the selection of a specific distribution of EUs among the CCSs. The EUs in a stack may be assigned to a single CCS, 2 CCSs, or 4 CCSs in the stack.

- 1-CCS mode (Default): In this mode, 1 CCS in each stack is exposed. The CCS has access to all the EUs in the stack. Other CCSs are disabled.
- 2-CCS mode: In this mode, 2 CCSs in each stack are exposed, each having half of the EUs in the stack assigned to it. If the EUs of one of the CCSs are idle, those EUs cannot be used by the other CCSs.
- 4-CCS mode: In this mode, all 4 CCSs of the stack are exposed, each having a quarter of the EUs in the stack assigned to it. As with the 2-CCS mode, EUs of idle CCSs cannot be used by other CCSs.

The default is 1-CCS mode.

Some applications may benefit from using 1 CCS per stack to access all the EUs in the stack, while other applications may benefit from using 2 or 4 CCSs per stack where a subset of the EUs are assigned to each CCS.

Using 2 or 4 CCSs per stack may be useful when running multiple small kernels concurrently on a stack, and the computations by each of these kernels does not require all the compute resources (EUs) of the stack. In this case, it may be advantageous to submit different kernels to different CCSs in the stack, thus allowing the kernels to run in parallel.

The environment variable ZEX\_NUMBER\_OF\_CCS can be used to specify how many CCSs are exposed in each of the stacks in a GPU card.

The format for ZEX\_NUMBER\_OF\_CCS is a comma-separated list of device-mode pairs, i.e., ZEX\_NUMBER\_OF\_CCS=<Root Device Index>:<CCS Mode>,<Root Device Index>:<CCS Mode>... For instance, in a GPU card with 2 stacks, one would specify the following to set stack 0 in 4-CCS mode, and stack 1 in 1-CCS mode.

```
ZEX_NUMBER_OF_CCS=0:4,1:1
```

Below we show examples of exposing CCSs in SYCL, OpenMP and MPI applications.

Using Multiple CCSs in SYCL

In SYCL, one can create a context associated with a CCS (subsubdevice), giving the program fine-grained control at the CCS level. The following example finds all stacks (subdevices) and CCSs (subsubdevices) on a GPU card (device):

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

int main() {
    // Find all GPU devices
    auto devices = sycl::platform(sycl::gpu_selector_v).get_devices();
    for (int n = 0; n < devices.size(); n++) {
        std::cout << "\nGPU" << n << ":" 
            << devices[n].get_info<sycl::info::device::name>() << " (" 
            << devices[n].get_info<sycl::info::device::max_compute_units>()
            << ")\n";
        std::vector<sycl::device> subdevices;
        std::vector<sycl::device> subsubdevices;
        auto part_prop =
            devices[n].get_info<sycl::info::device::partition_properties>();
        if (part_prop.empty()) {
            std::cout << "No partition_properties\n";
        } else {
            for (int i = 0; i < part_prop.size(); i++) {
                // Check if device can be partitioned into Tiles
                if (part_prop[i] ==
                    sycl::info::partition_property::partition_by_affinity_domain) {
                    auto sub_devices =
                        devices[n]
                            .create_sub_devices<sycl::info::partition_property::
                                partition_by_affinity_domain>(
                                sycl::info::partition_affinity_domain::numa);
                    for (int j = 0; j < sub_devices.size(); j++) {
                        subdevices.push_back(sub_devices[j]);
                        std::cout << "\ntile" << j << ":" 
                            << subdevices[j].get_info<sycl::info::device::name>()
                            << " (" 
                            << subdevices[j]
                                .get_info<sycl::info::device::max_compute_units>()
                            << ")\n";
                    }
                    auto part_prop1 =
                        subdevices[j]
                            .get_info<sycl::info::device::partition_properties>();
                    if (part_prop1.empty()) {
                        std::cout << "No partition_properties\n";
                    } else {
                        for (int i = 0; i < part_prop1.size(); i++) {
                            // Check if Tile can be partitioned into Slices (CCS)
                            if (part_prop1[i] == sycl::info::partition_property::
                                ext_intel_partition_by_cslice) {
                                auto sub_devices =
                                    subdevices[j]
                                        .create_sub_devices<
                                            sycl::info::partition_property::
                                                ext_intel_partition_by_cslice>();
                                for (int k = 0; k < sub_devices.size(); k++) {
                                    subsubdevices.push_back(sub_devices[k]);
                                    std::cout
                                        << "slice" << k << ":" 
                                        << subsubdevices[k].get_info<sycl::info::device::name>()
                                        << " ("
```

```

        << subsubdevices[k]
            .get_info<
                sycl::info::device::max_compute_units>()
        << ")\n";
    }
    break;
} else {
    std::cout << "No ext_intel_partition_by_cslice\n";
}
}
}
break;
// Check if device can be partitioned into Slices (CCS)
} else if (part_prop[i] == sycl::info::partition_property::
            ext_intel_partition_by_cslice) {
auto sub_devices =
    devices[n]
        .create_sub_devices<sycl::info::partition_property::
            ext_intel_partition_by_cslice>();
for (int k = 0; k < sub_devices.size(); k++) {
    subsubdevices.push_back(sub_devices[k]);
    std::cout << "slice" << k << ":" "
        << subsubdevices[k].get_info<sycl::info::device::name>()
        << " (""
        << subsubdevices[k]
            .get_info<sycl::info::device::max_compute_units>()
        << ")\n";
}
break;
} else {
    std::cout << "No ext_intel_partition_by_cslice or "
        "partition_by_affinity_domain\n";
}
}
}
return 0;
}
}

```

The SYCL code below demonstrates how multiple kernels can be submitted to multiple CCSs to execute concurrently.

The example code finds all CCSs, creates `sycl::queue` for each CCS found on GPU device and submits kernels to all CCSs using a for-loop.

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

static constexpr size_t N = 5280; // global size
static constexpr size_t B = 32; // WG size

void kernel_compute_mm(sycl::queue &q, float *a, float *b, float *c, size_t n,
                      size_t wg) {
    q.parallel_for(
        sycl::nd_range<2>(sycl::range<2>{n, n}, sycl::range<2>{wg, wg}),
        [=](sycl::nd_item<2> item) {
            const int i = item.get_global_id(0);
            const int j = item.get_global_id(1);
            float temp = 0.0f;
```

```
        for (int k = 0; k < N; k++) {
            temp += a[i * N + k] * b[k * N + j];
        }
        c[i * N + j] = temp;
    });
}

int main() {
    auto start =
        std::chrono::high_resolution_clock::now().time_since_epoch().count();

    // find all CCS / Tiles in GPU
    auto device = sycl::device(sycl::gpu_selector_v);
    std::cout << "\nGPU: " << device.get_info<sycl::info::device::name>() << " (" 
        << device.get_info<sycl::info::device::max_compute_units>()
        << ")\n";
    std::vector<sycl::device> subdevices;
    std::vector<sycl::device> subsubdevices;
    auto part_prop = device.get_info<sycl::info::device::partition_properties>();
    if (part_prop.empty()) {
        std::cout << "No partition_properties\n";
    } else {
        for (int i = 0; i < part_prop.size(); i++) {
            // Check if device can be partitioned into Tiles
            if (part_prop[i] ==
                sycl::info::partition_property::partition_by_affinity_domain) {
                auto sub_devices = device.create_sub_devices<
                    sycl::info::partition_property::partition_by_affinity_domain>(
                    sycl::info::partition_affinity_domain::numa);
                for (int j = 0; j < sub_devices.size(); j++) {
                    subdevices.push_back(sub_devices[j]);
                    std::cout
                        << "\nTile" << j << ":" 
                        << subdevices[j].get_info<sycl::info::device::name>() << " (" 
                        << subdevices[j].get_info<sycl::info::device::max_compute_units>()
                        << ")\n";
                    auto part_prop1 =
                        subdevices[j]
                            .get_info<sycl::info::device::partition_properties>();
                    if (part_prop1.empty()) {
                        std::cout << "No partition_properties\n";
                    } else {
                        for (int i = 0; i < part_prop1.size(); i++) {
                            // Check if Tile can be partitioned into Slices (CCS)
                            if (part_prop1[i] == sycl::info::partition_property::
                                ext_intel_partition_by_cslice) {
                                auto sub_devices = subdevices[j]
                                    .create_sub_devices<
                                        sycl::info::partition_property::ext_intel_partition_by_cslice>();
                                for (int k = 0; k < sub_devices.size(); k++) {
                                    subsubdevices.push_back(sub_devices[k]);
                                    std::cout
                                        << "Slice" << k << ":" 
                                        << subsubdevices[k].get_info<sycl::info::device::name>()
                                        << " (" 
                                        << subsubdevices[k]
                                            .get_info<sycl::info::device::max_compute_units>()
```

```
        << ") \n";
    }
    break;
} else {
    std::cout << "No ext_intel_partition_by_cslice\n";
}
}

}

break;
// Check if device can be partitioned into Slices (CCS)
} else if (part_prop[i] == sycl::info::partition_property::
           ext_intel_partition_by_cslice) {
    auto sub_devices = device.create_sub_devices<
        sycl::info::partition_property::ext_intel_partition_by_cslice>();
    for (int k = 0; k < sub_devices.size(); k++) {
        subsubdevices.push_back(sub_devices[k]);
        std::cout << "Slice" << k << ":" 
            << subsubdevices[k].get_info<sycl::info::device::name>()
            << " (" 
            << subsubdevices[k]
            .get_info<sycl::info::device::max_compute_units>()
            << ") \n";
    }
    break;
} else {
    std::cout << "No ext_intel_partition_by_cslice or "
        "partition_by_affinity_domain\n";
}
}

// Set devices to submit compute kernel
std::vector<sycl::device> devices(1, device);
if (subsubdevices.size())
    devices = subsubdevices;
else if (subdevices.size())
    devices = subdevices;
auto num_devices = devices.size();

// Define matrices
float *matrix_a[num_devices];
float *matrix_b[num_devices];
float *matrix_c[num_devices];

float v1 = 2.f;
float v2 = 3.f;
for (int n = 0; n < num_devices; n++) {
    matrix_a[n] = static_cast<float *>(malloc(N * N * sizeof(float)));
    matrix_b[n] = static_cast<float *>(malloc(N * N * sizeof(float)));
    matrix_c[n] = static_cast<float *>(malloc(N * N * sizeof(float)));

    // Initialize matrices with values
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            matrix_a[n][i * N + j] = v1++;
            matrix_b[n][i * N + j] = v2++;
            matrix_c[n][i * N + j] = 0.f;
```

```
    }

}

float *da[num_devices];
float *db[num_devices];
float *dc[num_devices];

std::vector<sycl::queue> q(num_devices);

// create queues for each device
std::cout << "\nSubmitting Compute Kernel to Devices:\n";
for (int i = 0; i < num_devices; i++) {
    q[i] = sycl::queue(devices[i]);
    std::cout
        << "Device" << i << ":" "
        << q[i].get_device().get_info<sycl::info::device::name>() << " (" "
        << q[i].get_device().get_info<sycl::info::device::max_compute_units>()
        << ")\n";
}

// device mem alloc for matrix a,b,c for each device
for (int i = 0; i < num_devices; i++) {
    da[i] = sycl::malloc_device<float>(N * N, q[i]);
    db[i] = sycl::malloc_device<float>(N * N, q[i]);
    dc[i] = sycl::malloc_device<float>(N * N, q[i]);
}

// warm up: kernel submit with zero size
for (int i = 0; i < num_devices; i++)
    kernel_compute_mm(q[i], da[i], db[i], dc[i], 0, 0);

// kernel sync
for (int i = 0; i < num_devices; i++)
    q[i].wait();

// memcpy for matrix and b to device alloc
for (int i = 0; i < num_devices; i++) {
    q[i].memcpy(&da[i][0], &matrix_a[i][0], N * N * sizeof(float));
    q[i].memcpy(&db[i][0], &matrix_b[i][0], N * N * sizeof(float));
}

// wait for copy to complete
for (int i = 0; i < num_devices; i++)
    q[i].wait();

// submit matrix multiply kernels to all devices
for (int i = 0; i < num_devices; i++)
    kernel_compute_mm(q[i], da[i], db[i], dc[i], N, B);

// wait for compute complete
for (int i = 0; i < num_devices; i++)
    q[i].wait();

// copy back result to host
for (int i = 0; i < num_devices; i++)
    q[i].memcpy(&matrix_c[i][0], &dc[i][0], N * N * sizeof(float));

// wait for copy to complete
```

```

for (int i = 0; i < num_devices; i++)
    q[i].wait();

// print first element of result matrix
std::cout << "\nMatrix Multiplication Complete\n";
for (int i = 0; i < num_devices; i++)
    std::cout << "device" << i << ": matrix_c[0][0] = " << matrix_c[i][0] << "\n";

for (int i = 0; i < num_devices; i++) {
    free(matrix_a[i]);
    free(matrix_b[i]);
    free(matrix_c[i]);
    sycl::free(da[i], q[i]);
    sycl::free(db[i], q[i]);
    sycl::free(dc[i], q[i]);
}

auto duration =
    std::chrono::high_resolution_clock::now().time_since_epoch().count() -
    start;
std::cout << "Compute Duration: " << duration / 1e9 << " seconds\n";
return 0;
}

```

To build the examples, run:

```

$ icpx -fsycl -o ccs ccs.cpp
$ icpx -fsycl -o ccs_matrixmul ccs_matrixmul.cpp

```

The number of CCSs found in `ccs` and the number of kernels executing in parallel in `ccs_matrixmul` depend on the setting of the environment variable `ZEX_NUMBER_OF_CCS`.

#### Using Multiple CCSs in OpenMP

In OpenMP, the CCSs in each stack can be exposed as devices to offer fine-grained partitioning and control at the CCS level.

In order to expose CCSs as devices, one of the following two environment variables should be set before running the program:

```
$ export ONEAPI_DEVICE_SELECTOR="*:.*.*"
```

or

```
$ LIBOMPTARGET_DEVICES=SUBSUBDEVICE
```

The following OpenMP program illustrates the use of CCSs in FLAT mode.

First, the program determines the number of devices that are available on the platform by calling `omp_get_num_devices()`. Then the program offloads kernels to each of the devices, where each kernel initializes a different chunk of array `A`.

`omp_get_num_devices()` returns the total number of devices that are available.

The `device` clause on the target directive is used to specify to which device a kernel should be offloaded.

At runtime the environment variable `ONEAPI_DEVICE_SELECTOR=".*.*"` (or `LIBOMPTARGET_DEVICES=SUBSUBDEVICE`) is set, along with `ZEX_NUMBER_OF_CCS`, to expose CCSs as devices.

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
```

```
#define SIZE 320

int num_devices = omp_get_num_devices();
int chunksize = SIZE/num_devices;

int main(void)
{
    int *A;
    A = new int[sizeof(int) * SIZE];

    printf ("num_devices = %d\n", num_devices);

    for (int i = 0; i < SIZE; i++)
        A[i] = -9;

    #pragma omp parallel for
    for (int id = 0; id < num_devices; id++) {
        #pragma omp target teams distribute parallel for device(id) \
            map(tofrom: A[id * chunksize : chunksize])
        for (int i = id * chunksize; i < (id + 1) * chunksize; i++) {
            A[i] = i;
        }
    }

    for (int i = 0; i < SIZE; i++)
        if (A[i] != i)
            printf ("Error in: %d\n", A[i]);
        else
            printf ("%d\n", A[i]);
}
}
```

**Compilation command:**

```
$ icpx -fiopenmp -fopenmp-targets=spir64 flat_openmp_02.cpp
```

**Run command:**

```
$ OMP_TARGET_OFFLOAD=MANDATORY ONEAPI_DEVICE_SELECTOR="*.*.*.*" \
ZEX_NUMBER_OF_CCS="0:4,1:4 ./a.out
```

**Notes:**

- The program is identical to the one in the [FLAT Mode Example - OpenMP](#). The only difference is that additional environment variables (ONEAPI\_DEVICE\_SELECTOR and ZEX\_NUMBER\_OF\_CCS) are set before running the program to expose CCSs (instead of stacks) as devices.
- Setting ONEAPI\_DEVICE\_SELECTOR="\*.\*.\*" causes CCSs to be exposed to the application as root devices. Alternatively, LIBOMPTARGET\_DEVICES=SUSBSUBDEVICE may be set.
- ZEX\_NUMBER\_OF\_CCS="0:4,1:4 specifies that the 4 CCSs in stack 0, as well as the 4 CCSs in stack 1, are exposed.
- OMP\_TARGET\_OFFLOAD=MANDATORY is used to make sure that the target region will run on the GPU. The program will fail if a GPU is not found.
- There is no need to specify ZE\_FLAT\_DEVICE\_HIERARCHY=FLAT with the run command, since FLAT mode is the default.

**Running on a system with a single GPU card (2 stacks in total):**

We add `LIBOMPTARGET_DEBUG=1` to the run command to get `libomptarget.so` debug information.

```
$ OMP_TARGET_OFFLOAD=MANDATORY ONEAPI_DEVICE_SELECTOR="*:.*.*" \
ZEX_NUMBER_OF_CCS="0:4,1:4 LIBOMPTARGET_DEBUG=1 ./a.out >& libomptarget_debug.log
```

We see the following in `libomptarget_debug.log`, showing that 8 devices corresponding to the 8 CCSs (4 CCSs in each of the 2 stacks) have been found.

```
Target LEVEL_ZERO RTL --> Found a GPU device, Name = Intel(R) Data Center GPU Max 1550
Target LEVEL_ZERO RTL --> Found 8 root devices, 8 total devices.
Target LEVEL_ZERO RTL --> List of devices (DeviceID[.SubID[.CCSID]])
Target LEVEL_ZERO RTL --> -- 0.0.0
Target LEVEL_ZERO RTL --> -- 0.0.1
Target LEVEL_ZERO RTL --> -- 0.0.2
Target LEVEL_ZERO RTL --> -- 0.0.3
Target LEVEL_ZERO RTL --> -- 1.0.0
Target LEVEL_ZERO RTL --> -- 1.0.1
Target LEVEL_ZERO RTL --> -- 1.0.2
Target LEVEL_ZERO RTL --> -- 1.0.3
```

#### **Running on a system with 4 GPU cards (8 stacks in total):**

We add `LIBOMPTARGET_DEBUG=1` to the run command to get `libomptarget.so` debug information.

```
$ OMP_TARGET_OFFLOAD=MANDATORY LIBOMPTARGET_DEBUG=1 ./a.out >& libomptarget_debug.log
```

We see the following in `libomptarget_debug.log`, showing that 32 devices corresponding to the 32 CCSs (4 CCSs in each of the 8 stacks) have been found:

```
Target LEVEL_ZERO RTL --> Found a GPU device, Name = Intel(R) Data Center GPU Max 1550
Target LEVEL_ZERO RTL --> Found 32 root devices, 32 total devices.
Target LEVEL_ZERO RTL --> List of devices (DeviceID[.SubID[.CCSID]])
Target LEVEL_ZERO RTL --> -- 0.0.0
Target LEVEL_ZERO RTL --> -- 0.0.1
Target LEVEL_ZERO RTL --> -- 0.0.2
Target LEVEL_ZERO RTL --> -- 0.0.3
Target LEVEL_ZERO RTL --> -- 1.0.0
Target LEVEL_ZERO RTL --> -- 1.0.1
Target LEVEL_ZERO RTL --> -- 1.0.2
Target LEVEL_ZERO RTL --> -- 1.0.3
Target LEVEL_ZERO RTL --> -- 2.0.0
Target LEVEL_ZERO RTL --> -- 2.0.1
Target LEVEL_ZERO RTL --> -- 2.0.2
Target LEVEL_ZERO RTL --> -- 2.0.3
Target LEVEL_ZERO RTL --> -- 3.0.0
Target LEVEL_ZERO RTL --> -- 3.0.1
Target LEVEL_ZERO RTL --> -- 3.0.2
Target LEVEL_ZERO RTL --> -- 3.0.3
Target LEVEL_ZERO RTL --> -- 4.0.0
Target LEVEL_ZERO RTL --> -- 4.0.1
Target LEVEL_ZERO RTL --> -- 4.0.2
Target LEVEL_ZERO RTL --> -- 4.0.3
Target LEVEL_ZERO RTL --> -- 5.0.0
Target LEVEL_ZERO RTL --> -- 5.0.1
Target LEVEL_ZERO RTL --> -- 5.0.2
Target LEVEL_ZERO RTL --> -- 5.0.3
Target LEVEL_ZERO RTL --> -- 6.0.0
Target LEVEL_ZERO RTL --> -- 6.0.1
Target LEVEL_ZERO RTL --> -- 6.0.2
Target LEVEL_ZERO RTL --> -- 6.0.3
Target LEVEL_ZERO RTL --> -- 7.0.0
```

```
Target LEVEL_ZERO RTL --> -- 7.0.1
Target LEVEL_ZERO RTL --> -- 7.0.2
Target LEVEL_ZERO RTL --> -- 7.0.3
```

## Using Multiple CCSs in MPI

A typical use case for running more than 1 CCS per GPU stack is in MPI applications where there are large portions of the application time consumed by non-offloaded code run on the CPU. Running with 4-CCS mode will allow the user to run with MPI ranks numbering four times the number of GPU stacks, allowing the host process to consume more CPU cores.

An example of DGEMMs executed through MPI is shown in the following source:

```
#include "mkl.h"
#include "mkl_omp_offload.h"
#include <algorithm>
#include <chrono>
#include <limits>
#include <mpi.h>
#include <omp.h>
#define FLOAT double
#define MPI_FLOAT_T MPI_DOUBLE
#define MKL_INT_T MKL_INT
#define index(i, j, ld) (((j) * (ld)) + (i))
#define RAND() ((FLOAT)rand() / (FLOAT)RAND_MAX * 2.0 - 1.0)
#define LD_ALIGN 256
#define LD_BIAS 8
#define HPL_PTR(ptr_, al_) (((size_t)(ptr_) + (al_-1) / (al_)) * (al_))
static inline MKL_INT_T getld(MKL_INT_T x) {
    MKL_INT_T ld;
    ld = HPL_PTR(x, LD_ALIGN);
    if (ld - LD_BIAS >= x)
        ld -= LD_BIAS;
    else
        ld += LD_BIAS;
    return ld;
}
int main(int argc, char **argv) {
    if ((argc < 4) || (argc > 4 && argc < 8)) {
        printf("Performs a DGEMM test C = alpha*A*B + beta*C\n");
        printf("A matrix is MxK and B matrix is KxN\n");
        printf("All matrices are stored in column-major format\n");
        printf("Run as ./dgemm <M> <K> <N> [<alpha> <beta> <iterations>]\n");
        printf("Required inputs are:\n");
        printf("      M: number of rows of matrix A\n");
        printf("      K: number of cols of matrix A\n");
        printf("      N: number of cols of matrix B\n");
        printf("Optional inputs are (all must be provided if providing any):\n");
        printf("      alpha: scalar multiplier (default: 1.0)\n");
        printf("      beta: scalar multiplier (default: 0.0)\n");
        printf("      iterations: number of blocking DGEMM calls to perform "
               "(default: 10)\n");
        return EXIT_FAILURE;
    }
    MKL_INT_T HA = (MKL_INT_T)(atoi(argv[1]));
    MKL_INT_T WA = (MKL_INT_T)(atoi(argv[2]));
    MKL_INT_T WB = (MKL_INT_T)(atoi(argv[3]));
    FLOAT alpha, beta;
    int niter;
```

```
if (argc > 4) {
    sscanf(argv[4], "%lf", &alpha);
    sscanf(argv[5], "%lf", &beta);
    niter = atoi(argv[6]);
} else {
    alpha = 1.0;
    beta = 0.0;
    niter = 10;
}
MKL_INT_T HB = WA;
MKL_INT_T WC = WB;
MKL_INT_T HC = HA;
MKL_INT_T ldA = getld(HA);
MKL_INT_T ldB = getld(HB);
MKL_INT_T ldC = getld(HC);
double tot_t = 0.0, best_t = std::numeric_limits<double>::max();
FLOAT *A = new FLOAT[ldA * WA];
FLOAT *B, *C, *local_B, *local_C;
MPI_Init(&argc, &argv);
int mpi_rank, mpi_size;
MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
if (mpi_rank == 0) {
    B = new FLOAT[ldB * WB];
    C = new FLOAT[ldC * WC];
    srand(2864);
    for (int j = 0; j < WA; j++)
        for (int i = 0; i < HA; i++)
            A[index(i, j, ldA)] = RAND();
    for (int j = 0; j < WB; j++)
        for (int i = 0; i < HB; i++)
            B[index(i, j, ldB)] = RAND();
    if (beta != 0.0) {
        for (int j = 0; j < WC; j++)
            for (int i = 0; i < HC; i++)
                C[index(i, j, ldC)] = RAND();
    } else {
        for (int j = 0; j < WC; j++)
            for (int i = 0; i < HC; i++)
                C[index(i, j, ldC)] = 0.0;
    }
}
size_t sizea = (size_t)ldA * WA;
size_t local_sizeb, local_sizec;
int *displacements_b = new int[mpi_size];
int *send_counts_b = new int[mpi_size];
int *displacements_c = new int[mpi_size];
int *send_counts_c = new int[mpi_size];
int local_WB = WB / mpi_size;
send_counts_b[0] = ldB * (local_WB + WB % mpi_size);
send_counts_c[0] = ldC * (local_WB + WB % mpi_size);
displacements_b[0] = 0;
displacements_c[0] = 0;
for (int i = 1; i < mpi_size; i++) {
    send_counts_b[i] = ldB * local_WB;
    send_counts_c[i] = ldC * local_WB;
    displacements_b[i] = displacements_b[i - 1] + send_counts_b[i - 1];
    displacements_c[i] = displacements_c[i - 1] + send_counts_c[i - 1];
}
```

```

}
if (mpi_rank == 0) {
    local_WB += WB % mpi_size;
}
local_sizeb = ldB * local_WB;
local_sizec = ldC * local_WB;
local_B = new FLOAT[local_sizeb];
local_C = new FLOAT[local_sizec];
MPI_Bcast(A, sizea, MPI_FLOAT_T, 0, MPI_COMM_WORLD);
MPI_Scatterv(B, send_counts_b, displacements_b, MPI_FLOAT_T, local_B,
    local_sizeb, MPI_FLOAT_T, 0, MPI_COMM_WORLD);
MPI_Scatterv(C, send_counts_c, displacements_c, MPI_FLOAT_T, local_C,
    local_sizec, MPI_FLOAT_T, 0, MPI_COMM_WORLD);
#endif defined(OMP_AFFINITIZATION)
#ifndef OMP_AFFINITIZATION == 1
    int ndev = omp_get_num_devices();
    int dnum = mpi_rank % ndev;
    omp_set_default_device(dnum);
#endif
#endif
#pragma omp target data map(to
    : A [0:sizea], local_B [0:local_sizeb]) \
    map(tofrom
    : local_C [0:local_sizec]) \
{
#pragma omp dispatch
    dgemm("N", "N", &HA, &local_WB, &WA, &alpha, A, &ldA, local_B, &ldB, &beta,
        local_C, &ldC);
    for (int i = 0; i < niter; i++) {
        auto start_t = std::chrono::high_resolution_clock::now();
#pragma omp dispatch
        dgemm("N", "N", &HA, &local_WB, &WA, &alpha, A, &ldA, local_B, &ldB,
            &beta, local_C, &ldC);
        MPI_Barrier(MPI_COMM_WORLD);
        auto end_t = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> diff = end_t - start_t;
        tot_t += diff.count();
        best_t = std::min(best_t, diff.count());
    }
}
MPI_Gatherv(local_C, local_sizec, MPI_FLOAT_T, C, send_counts_c,
    displacements_c, MPI_FLOAT_T, 0, MPI_COMM_WORLD);
delete[] local_B;
delete[] local_C;
delete[] displacements_b;
delete[] displacements_c;
delete[] send_counts_b;
delete[] send_counts_c;
MPI_Allreduce(MPI_IN_PLACE, &tot_t, 1, MPI_FLOAT_T, MPI_MAX, MPI_COMM_WORLD);
MPI_Allreduce(MPI_IN_PLACE, &best_t, 1, MPI_FLOAT_T, MPI_MAX, MPI_COMM_WORLD);
if (mpi_rank == 0) {
    double tflop_count = (double)2.0 * HA * WB * WA;
    if (beta != 0.0)
        tflop_count += (double)HA * WB;
    tflop_count *= 1.E-12;
    printf("Total runtime for %d iterations: %f seconds.\n", niter, tot_t);
    printf("Mean TFLOP/s: %f\n", (double)niter * tflop_count / tot_t);
    printf("Best TFLOP/s: %f\n", (double)tflop_count / best_t);
}

```

```

    delete[] B;
    delete[] C;
}
delete[] A;
MPI_Finalize();
return EXIT_SUCCESS;
}

```

In this example, the DGEMMs are MKL calls executed through OpenMP offload. The matrices are statically partitioned among the MPI ranks.

In order to build the binary, execute:

```
$ cd examples/MPI/02_omp_mpi_onemkl_dgemm
$ make
```

With Intel® MPI, each process can bind to one or multiple GPU stacks. If more than one process is allocated to a GPU stack, the GPU driver will enqueue a kernel to one of the CCS associated with the stack. We can rely on the environment variable `I_MPI_OFFLOAD_CELL_LIST` to specify the device stacks used.

For example to run the application in 4-CCS mode, with the four MPI ranks being allocated to the first device's first stack:

```
$ export ZEX_NUMBER_OF_CCS=0:4
$ export I_MPI_OFFLOAD_CELL_LIST=0,0,0,0
$ mpirun -n 4 ./dgemm 8192 8192 8192
```

If we want to run the application with the first four MPI ranks being allocated to the first device's first stack, and second four MPI ranks being allocated to the first device's second stack. The expectation is that this will have double the FLOP/s as the previous run:

```
$ export ZEX_NUMBER_OF_CCS=0:4,1:4
$ export I_MPI_OFFLOAD_CELL_LIST=0,0,0,0,1,1,1,1
$ mpirun -n 8 ./dgemm 8192 8192 8192
```

Note that the following Intel® MPI environment variables are default, but may be useful to specify or modify in some cases:

```
$ export I_MPI_OFFLOAD_CELL=tile #Associated MPI ranks with GPU tiles
$ export I_MPI_OFFLOAD=1 #Enable MPI work with device pointers
$ export I_MPI_OFFLOAD_TOPOLIB=level_zero #Use Level Zero for topology detection (GPU pinning)
```

With MPICH, MPI ranks associate with a GPU stack explicitly through the environment variable `ZE_AFFINITY_MASK`. The driver will subsequently associate the rank to a CCS on the stack.

The same example application can be built with MPICH, if an appropriate MPICH installation is loaded. An example script to bind MPI ranks in a similar matter is provided in:

```
#!/bin/bash

if [ -z ${NCCS} ]; then
    NCCS=1
fi

if [ -z ${NGPUS} ]; then
    NGPU=1
fi

if [ -z ${NSTACKS} ]; then
    NSTACK=1
fi
```

```
subdevices=$((NGPU*NSTACK))  
export ZE_AFFINITY_MASK=$((((MPI_LOCALRANKID/NCCS)%subdevices))  
echo MPI_LOCALRANKID = $MPI_LOCALRANKID ZE_AFFINITY_MASK = $ZE_AFFINITY_MASK  
exec $@
```

Assuming that a node has 6 GPUs, each GPU has 2 stacks, and you want to run with 4 CCS per stack, usage of this script is as follows:

```
$ export NGPUS=6  
$ export NSTACKS=2  
$ export NCCS=4  
$ export ZEX_NUMBER_OF_CCS=0:${NCCS},1:${NCCS}  
$ mpixexec -n 48 ./gpu_rank_bind.sh ./dgemm 8192 8192 8192
```

Since the MKL DGEMM is handled through OpenMP offload, we can also associate the MPI ranks explicitly with specific CCSs through OpenMP. Some mixed MPI/OpenMP offload applications use this strategy.

```
#if OMP_AFFINITIZATION == 1  
int ndev = omp_get_num_devices();  
dnum = mpi_rank % ndev;  
omp_set_default_device(dnum);  
#endif  
...  
#pragma omp target data map(to:A[0:sizea],local_B[0:local_sizeb])  
map(tofrom:local_C[0:local_sizec])  
{  
...  
#pragma omp dispatch  
    dgemm("N","N",&HA,&WA,&alpha,A,&ldA,local_B,&ldB,&beta,local_C,&ldC);  
...  
}
```

This mode of the binary can be built by using the following options:

```
$ cd examples/MPI/02_omp_mpi_onemkl_dgemm  
$ make OMP_AFFINITIZATION=1
```

We can then run the application without explicitly specifying device affinity.

```
$ export ZEX_NUMBER_OF_CCS=0:4,1:4  
$ export LIBOMPTARGET_DEVICES=SUSUBDEVICE  
$ mpirun -n 8 ./dgemm.out
```

## Terminology

### Terminologies : header-rows: 1

Term	Abbreviation	Definition
Blitter	BLT	Block Image Transferrer
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 a VE and forwarded to the Thread

Command		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 <i>Parent Thread</i> .
Command Streamer	CS or CSI	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 a VE.
Core		Functional unit of the Graphics Processing Engine that fetches commands, parses them, and routes them to the appropriate pipeline.
Dual Sub-slice	DSS	The collection of Vector Engines (VE) or 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.
Exception		Type of (normally rare) interruption to VE/EU execution of a thread's instructions. An exception occurrence causes the VE/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 (SR0) 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/VEs 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.
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 <i>Graphics Processing Engine</i> .
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.
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/VE. 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, $r2$ , is a 256-bits wide, 256-bit aligned register. A sub-register, $r2.3:d$ , is the fourth dword of GRF register $r2$ .
Subsystem		The name given to the resources shared by the FF units, including shared functions and EUs/VEs.
Surface		A rendering operand or destination, including textures, buffers, and render targets.
Surface State		State associated with a render surface.

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/VE. The life cycle for a thread starts from the executing the first instruction after being dispatched from Thread Dispatcher to an EU/VE to the execution of the last instruction - a <i>send</i> 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/VEs.
Thread Identifier	TID	The field within a thread state register (SR0) that identifies which thread slots on an EU/VE a thread occupies. A thread can be uniquely identified by the EUID and TID.
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.

## Level Zero

Level Zero is the low-level interface that enables oneAPI libraries to exploit the hardware capabilities of target devices. This section provides an insight into the architectural design followed in the Intel® Graphics Compute Runtime for oneAPI Level Zero. Implementation details and optimization guidelines are explained, as well as a description of the different features available for the different supported platforms.

- [Immediate Command Lists](#)

### Immediate Command Lists

#### Introduction

Immediate command lists is a feature provided by Level-Zero specification to allow for very low latency submission usage models. In this scheme, commands appended on the command list such as launching a kernel or performing a memory copy are immediately submitted to the device for execution. This is different from a regular command list where multiple commands can be stitched and submitted together for execution .

Distinctions between an immediate command list compared to a regular command list include (but not limited to) the following:

- An immediate command list is an implicit command queue and is therefore created using a command queue descriptor.
- Commands appended to an immediate command list are submitted for execution immediately on the device.
- Immediate command lists are not required to be closed or reset.
- Synchronization of immediate command lists cannot be performed via `zeCommandQueueSynchronize` or `zeFenceHostSynchronize` as there is no command queue handle associated with the immediate command list. Recommendation is to use events to confirm commands submitted to the immediate command list have completed.

Since the intention of immediate command lists is to primarily provide a razor thin submission interface to the device, they are well suited to be used in workloads which have tendency to launch small or short running kernels and also need to run multiple iterations of such kernels. Examples of workloads with such characteristics can be found in HPC environments and also ML/DL frameworks.

### Programming Model

Following code shows how to create an immediate command list and submitting a kernel with it. Synchronization is achieved by querying the event status.

```
ze_command_queue_desc_t cmdQueueDesc = {ZE_STRUCTURE_TYPE_COMMAND_QUEUE_DESC};
cmdQueueDesc.pNext = nullptr;
cmdQueueDesc.flags = 0;
cmdQueueDesc.priority = ZE_COMMAND_QUEUE_PRIORITY_NORMAL;
cmdQueueDesc.ordinal = queueGroupOrdinal;
cmdQueueDesc.index = 0;
```

```

cmdQueueDesc.mode = ZE_COMMAND_QUEUE_MODE_ASYNCHRONOUS;
zeCommandListCreateImmediate(context, device, &cmdQueueDesc, &cmdList);

zeCommandListAppendLaunchKernel(cmdList, kernel, &dispatchTraits,
                               events[0], 0, nullptr);
// If Async mode, use event for sync
zeEventHostSynchronize(events[0], std::numeric_limits<uint64_t>::max() - 1);

```

Immediate command lists may also be used to implement in-order queues. In this case, commands submitted to the list are chained together using events, as seen below.

```

zeCommandListAppendMemoryCopy(cmdList, deviceBuffer, hostBuffer, allocSize,
                             events[0],
                             0, nullptr);

zeCommandListAppendMemoryCopy(cmdList, stackBuffer, deviceBuffer, allocSize,
                             events[1],
                             1,
                             &events[0]);

zeEventHostSynchronize(events[1], std::numeric_limits<uint64_t>::max() - 1));

```

As with regular lists, immediate command lists may also be synchronous. In this case, synchronization is performed implicitly and each command submitted to the list is immediately submitted, and is guaranteed to have completed upon return from the call.

```

ze_command_queue_desc_t cmdQueueDesc = {ZE_STRUCTURE_TYPE_COMMAND_QUEUE_DESC};
cmdQueueDesc.pNext = nullptr;
cmdQueueDesc.flags = 0;
cmdQueueDesc.priority = ZE_COMMAND_QUEUE_PRIORITY_NORMAL;
cmdQueueDesc.ordinal = queueGroupOrdinal;
cmdQueueDesc.index = 0;
cmdQueueDesc.mode = ZE_COMMAND_QUEUE_MODE_SYNCHRONOUS;
zeCommandListCreateImmediate(context, device, &cmdQueueDesc, &cmdList);

zeCommandListAppendLaunchKernel(cmdList, kernel, &dispatchTraits,
                               nullptr, 0, nullptr);

// At this point, kernel has been executed

```

For more code samples, please refer compute-benchmarks repository <https://github.com/intel/compute-benchmarks>. Scenarios such as `create_command_list_immediate_10.cpp` and `execute_command_list_immediate_10.cpp` serve as good starting points.

## Performance Profiling and Analysis

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 like interval timers, kernel timers, and print statements are lighter weight. But all of them serve an important purpose in the optimization process.

This section covers topics related to these tools' use for software optimization.

- [Using the Timers](#)
- [Intel® VTune™ Profiler](#)
- [Intel® Advisor](#)
- [Intel® Intercept Layer for OpenCL™ Applications](#)
- [Performance Tools in Intel® Profiling Tools Interfaces for GPU](#)

## Using the Timers

The standard C++ chrono library can be used for tracking times with varying degrees of precision in SYCL. The following example shows how to use the chrono timer class to time kernel execution from the host side.

```
#include <CL/sycl.hpp>
#include <iostream>
using 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);

    auto t1 = std::chrono::steady_clock::now(); // Start timing

    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]; });
    }).wait();

    auto t2 = std::chrono::steady_clock::now(); // Stop timing

    return (std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count());
}

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);

    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 in " << t << " microseconds\n";
    return 0;
}
```

Note 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. SYCL provides a profiling capability that let you 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);
        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 in " << t << " nanoseconds\n";
    return 0;
}
```

When these examples are run, it is quite possible that the time reported by chrono is much larger than the time reported by the SYCL profiling class. This is because the SYCL profiling does not include any data transfer times between the host and the offload device.

## 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 SYCL 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 SYCL 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 in 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 lets you:

- 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 SYCL or OpenCL kernels for inefficient kernel code algorithms or incorrect work item configuration.
- Run GPU Offload Analysis on a SYCL Application.

## Using VTune Profiler to Analyze GPU Applications

1. Launch VTune Profiler and from the Welcome page, click **New Project**. 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 the GPU Offload analysis type from the Accelerators group.
6. Click **Start** to launch the analysis.

This is the least intrusive analysis for applications running on platforms with Intel Graphics as well as third-party GPUs supported by VTune Profiler.

## Run Analysis from the 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
```

1. Run the analysis command:

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

On Windows\* OS:

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

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

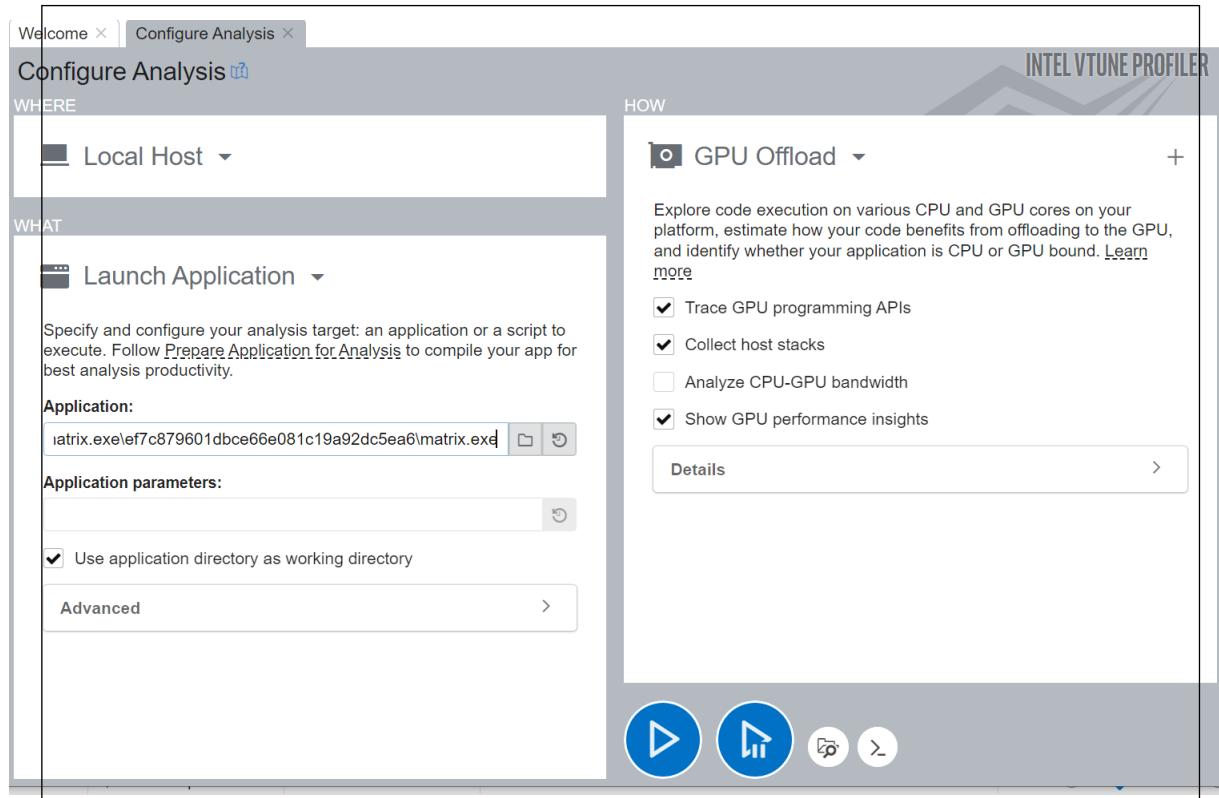
1. Run the analysis command:

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

## Analyzing Collected Data

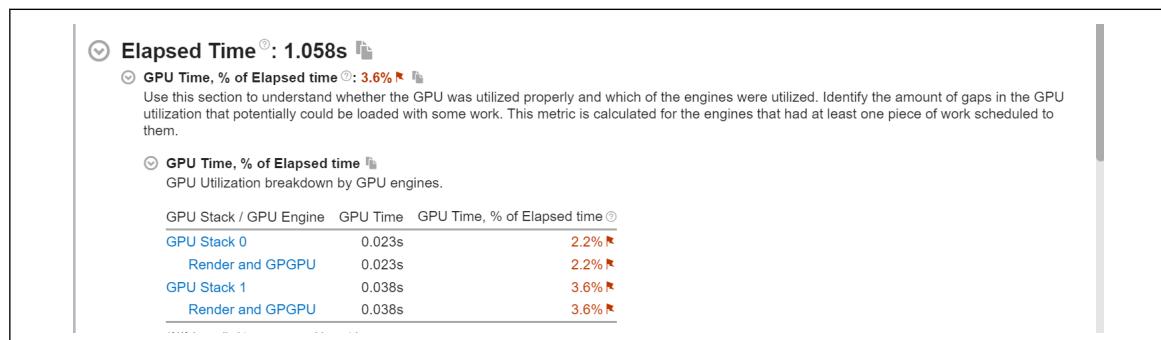
Start your analysis with the GPU Offload viewpoint. In the Summary window, you can see see statistics on CPU and GPU resource usage to determine if your application is GPU-bound, CPU-bound, or not effectively utilizing the compute capabilities of the system. In this example, the application should use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

### 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.

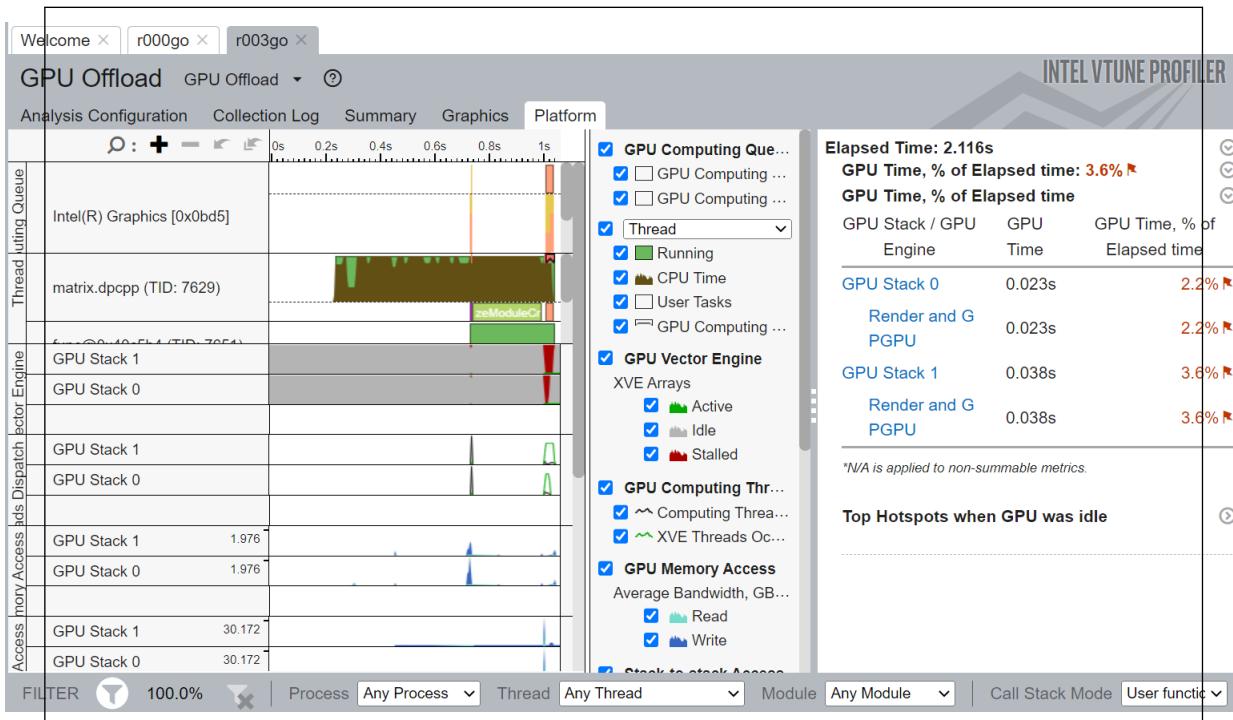
### GPU Utilization



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 ineffective scheduling on the GPU, which can lead you to mistakenly interpret the application as GPU bound.

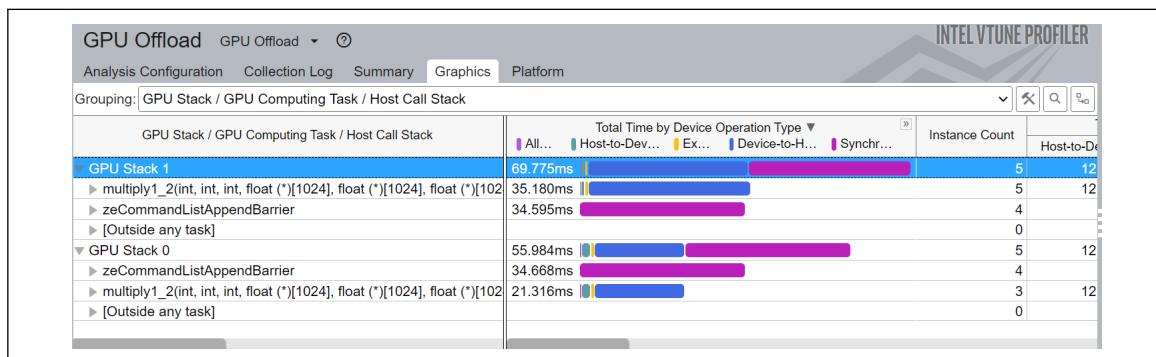
Identify the GPU execution phase based on the computing task reference and GPU Utilization metrics. Then, you can define the overhead for creating the task and placing it in a queue.

## 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.

## Computing Task grouping

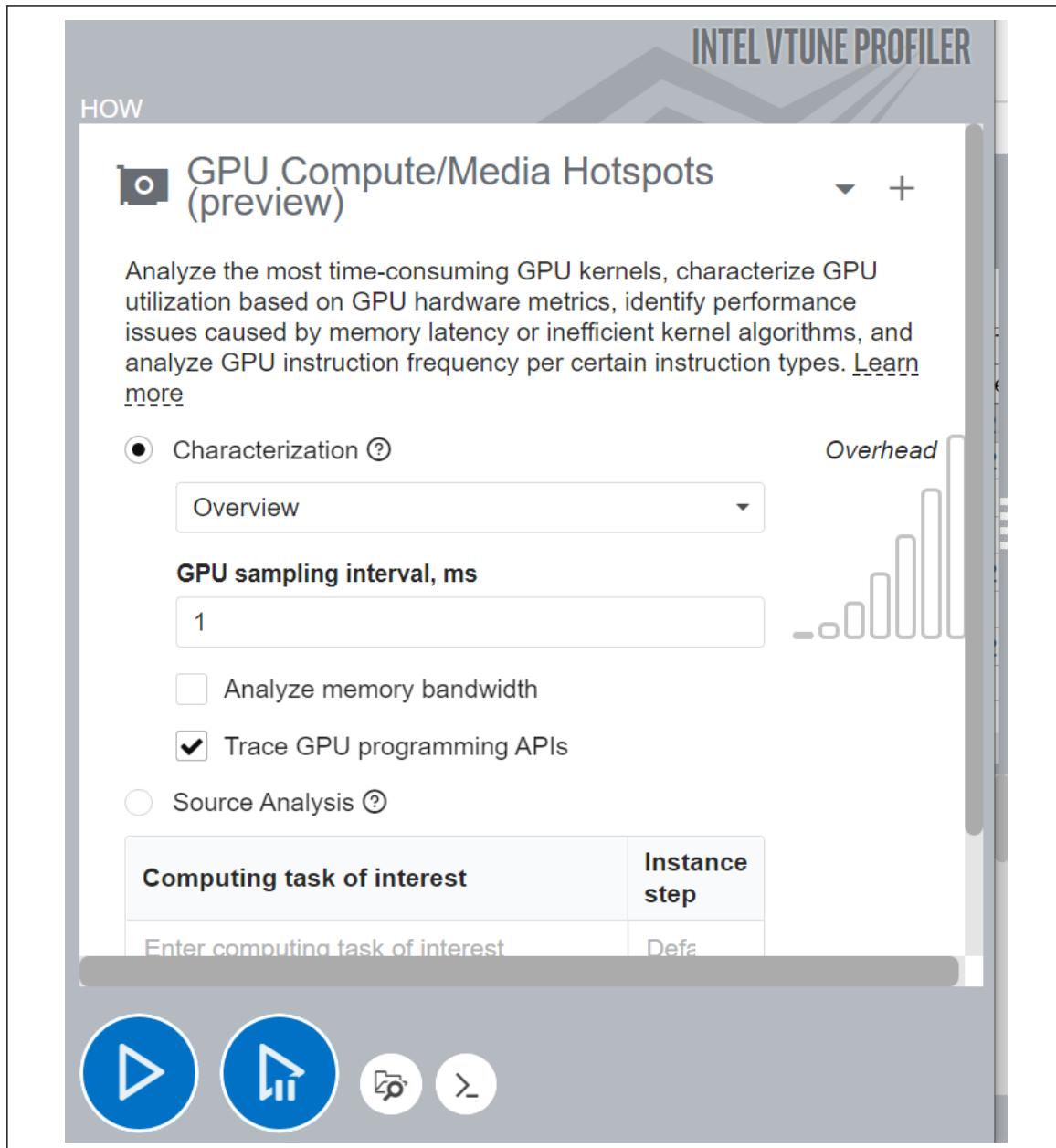


## Run GPU Compute/Media Hotspots Analysis

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 **Start** to run the analysis.

### 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
```

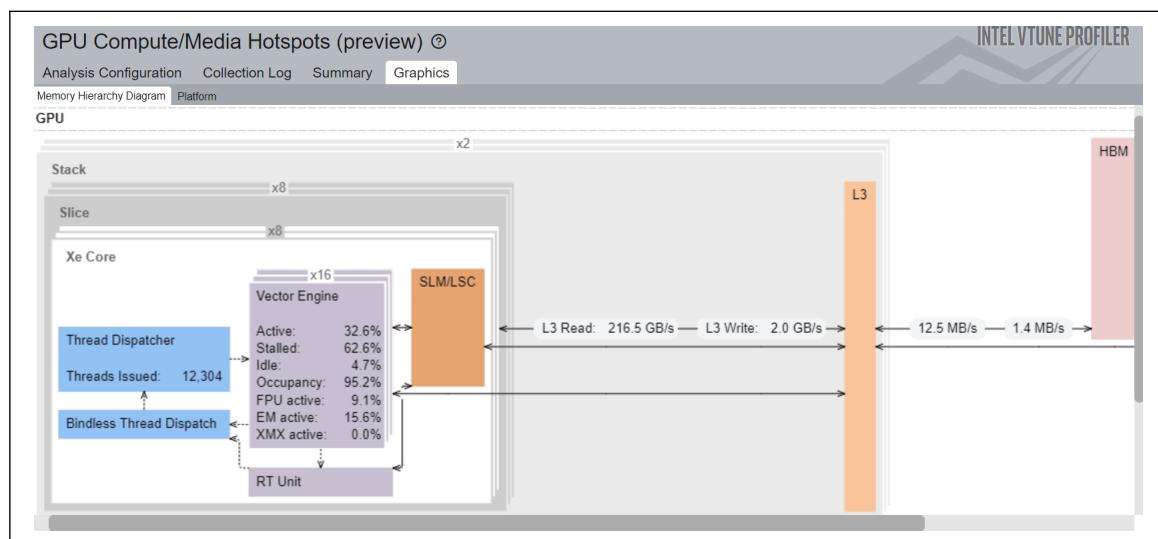
## Analyze Your Compute Tasks

### Characterization profile

GPU Stack / Computing Task	Work Size		Computing Task			
	Global ▼	Local	Total Time	Average Time	Instance Count	SIMD Width
GPU Stack 0			54.286ms	6.032ms	9	
▶ multiply1_2(int, int, int, float (*)[1024], float (*)[1024], float (*)[1024], float (*)[1024] x16 x 16)			0.532ms	0.532ms	1	32
▶ zeCommandListAppendMemoryCopyRegion			0.380ms	0.127ms	3	
▶ zeCommandListAppendMemoryCopyRegion			18.659ms	18.659ms	1	
▶ zeCommandListAppendBarrier			34.716ms	8.679ms	4	
▶ [Outside any task]			0ms	0ms	0	
GPU Stack 1			69.811ms	7.757ms	9	
▶ multiply1_2(int, int, int, float (*)[1024], float (*)[1024], float (*)[1024], float (*)[1024] x16 x 16)			0.537ms	0.537ms	1	32
▶ zeCommandListAppendMemoryCopyRegion			0.534ms	0.178ms	3	
▶ zeCommandListAppendMemoryCopyRegion			34.089ms	34.089ms	1	
▶ zeCommandListAppendBarrier			34.652ms	8.663ms	4	
▶ [Outside any task]			0ms	0ms	0	

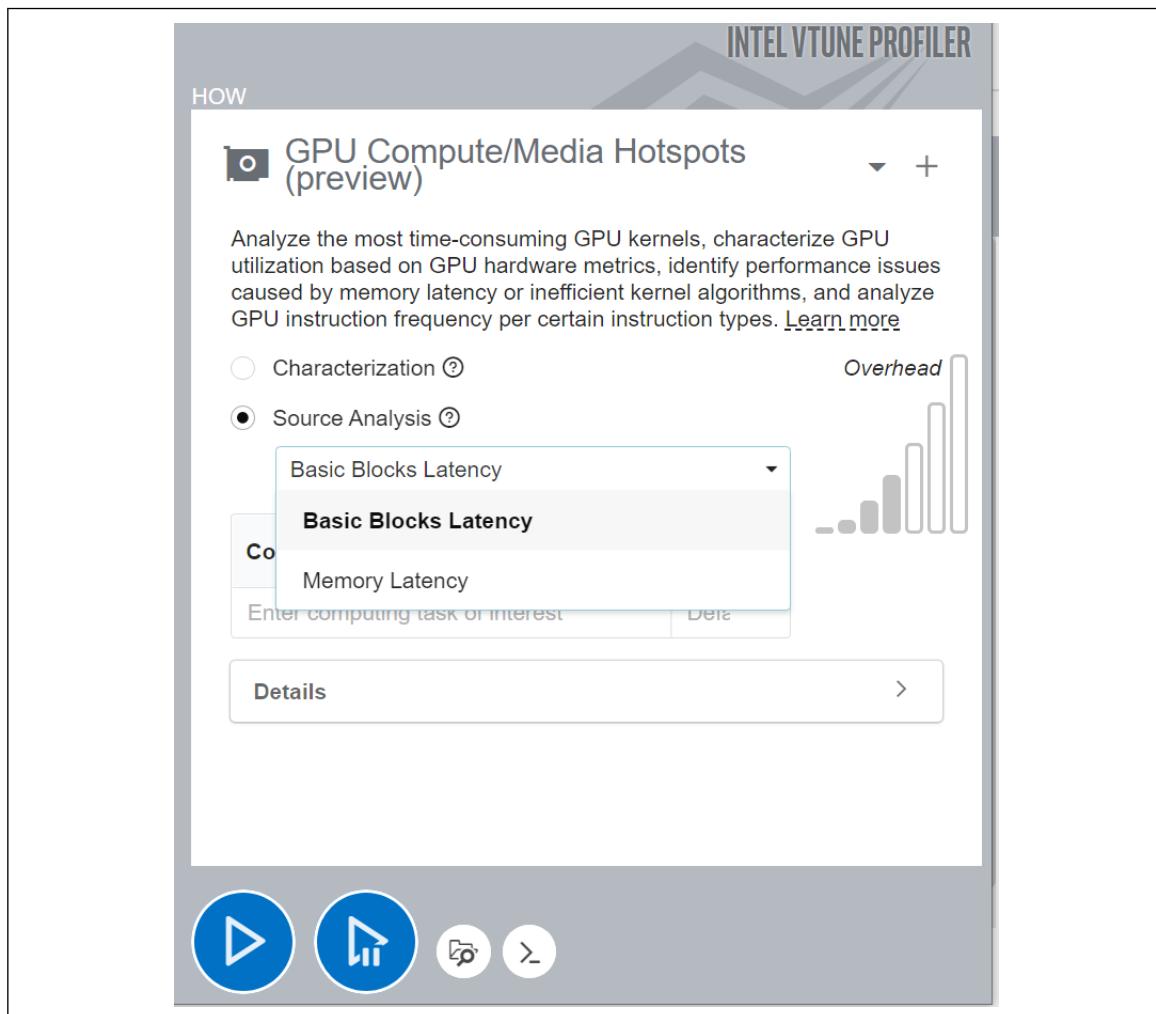
The default analysis configuration invokes the Characterization profile with the Overview metric set. In addition to individual compute task characterization available through the GPU Offload analysis, VTune Profiler provides memory bandwidth metrics that are categorized by different levels of GPU memory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

### 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.

### GPU Compute/Media Hotspots analysis, Source Analysis



Use our matrix multiply example in SYCL:

```
// Basic matrix multiply
void multiply1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
              TYPE c[][NUM], TYPE t[][NUM]) {
    int i, j, k;

    // Declare a deviceQueue
    sycl::queue q(sycl::default_selector_v, exception_handler);
    cout << "Running on " << q.get_device().get_info<sycl::info::device::name>()
        << "\n";
    // Declare a 2 dimensional range
    sycl::range<2> matrix_range{NUM, NUM};

    // Declare 3 buffers and Initialize them
    sycl::buffer<TYPE, 2> bufferA((TYPE *)a, matrix_range);
    sycl::buffer<TYPE, 2> bufferB((TYPE *)b, matrix_range);
    sycl::buffer<TYPE, 2> bufferC((TYPE *)c, matrix_range);
    // Submit our job to the queue
    q.submit([&](auto &h) {
```

```

// Declare 3 accessors to our buffers. The first 2 read and the last
// read_write
sycl::accessor accessorA(bufferA, h, sycl::read_only);
sycl::accessor accessorB(bufferB, h, sycl::read_only);
sycl::accessor accessorC(bufferC, h);

// Execute matrix multiply in parallel over our matrix_range
// ind is an index into this range
h.parallel_for(matrix_range, [=](sycl::id<2> ind) {
    int k;
    for (k = 0; k < NUM; k++) {
        // Perform computation ind[0] is row, ind[1] is col
        accessorC[ind[0]][ind[1]] +=
            accessorA[ind[0]][k] * accessorB[k][ind[1]];
    }
});
}).wait_and_throw();
} // multiply1

```

Analyzing the GPU-offload report from the command-line gives detailed recommendations on how to optimize the application.

GPU Time, % of Elapsed time: 6.8%

| GPU utilization is low. Consider offloading more work to the GPU to  
| increase overall application performance.  
|

Top Hotspots when GPU was idle

Function	Module	CPU Time
[Outside any known module]	[Unknown]	0.402s
[Skipped stack frame(s)]	[Unknown]	0.228s
operator new	libc++abi.so	0.136s
memmove	libc-dynamic.so	0.078s
memcmp	libc-dynamic.so	0.058s
[Others]	N/A	0.918s

Hottest Host Tasks

Host Task	Task Time	% of Elapsed Time (%)	Task Count
zeModuleCreate	0.233s	22.4%	1
zeEventHostSynchronize	0.068s	6.5%	6
zeCommandListCreateImmediate	0.003s	0.3%	1
zeCommandListAppendMemoryCopyRegion	0.002s	0.2%	4
zeMemAllocDevice	0.002s	0.2%	3
[Others]	0.003s	0.3%	14

Hottest GPU Computing Tasks

GPU Adapter	Task	Computing				
Total Time	Execution Time	% of Total Time (%)	SIMD Width	XVE Threads	Occupancy (%)	SIMD
Utilization (%)						

```

0:41:0.0 : Display controller: Intel Corporation Device 0x0bdb multiply1_2(int, int, int, float
(*)[1024], float (*)[1024], float (*)[1024], float (*)[1024])::
{lambd()#1}::operator()<sycl::_V1::handler>(), signed char) const::
{lambd(sycl::_V1::nd_item<(int)2>)#1}          0.071s      0.003s
4.5%           32           97.0%       100.0%
0:41:0.0 : Display controller: Intel Corporation Device 0x0bdb
zeCommandListAppendBarrier

          0.065s      0s      0.0%
Collection and Platform Info
  Application Command Line: ./matrix.dpcpp
  Operating System: 5.15.47+prerelease6379.25 DISTRO_ID=Ubuntu DISTRO_RELEASE=20.04
DISTRO_CODENAME=focal DISTRO_DESCRIPTION="Ubuntu 20.04.2 LTS"
  Computer Name: ortce-pvc1
  Result Size: 49.2 MB
  Collection start time: 04:55:43 16/03/2023 UTC
  Collection stop time: 04:55:44 16/03/2023 UTC
  Collector Type: Event-based sampling driver,Driverless Perf system-wide sampling,User-mode
sampling and tracing
CPU
  Name: Intel(R) Xeon(R) Processor code named Sapphirerapids
  Frequency: 2.000 GHz
  Logical CPU Count: 224
  LLC size: 110.1 MB
GPU
  0:41:0.0: Display controller: Intel Corporation Device 0x0bdb:
    XVE Count: 448
    Max XVE Thread Count: 8
    Max Core Frequency: 1.550 GHz
  0:58:0.0: Display controller: Intel Corporation Device 0x0bdb:
    XVE Count: 448
    Max XVE Thread Count: 8
    Max Core Frequency: 1.550 GHz

Recommendations:
  GPU Time, % of Elapsed time: 6.8%
  | GPU utilization is low. Switch to the for in-depth analysis of host
  | activity. Poor GPU utilization can prevent the application from
  | offloading effectively.
  XVE Array Stalled/Idle: 97.0% of Elapsed time with GPU busy
  | GPU metrics detect some kernel issues. Use GPU Compute/Media Hotspots
  | (preview) to understand how well your application runs on the specified
  | hardware.
  Execution % of Total Time: 2.4%
  | 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 efficiently 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.

```

Elapsed Time: 1.682s
GPU Time: 0.071s
Display controller: Intel Corporation Device 0x0bdb Device Group
XVE Array Stalled/Idle: 98.5% of Elapsed time with GPU busy
| The percentage of time when the XVEs were stalled or idle is high, which
| has a negative impact on compute-bound applications.

```

XVE Metrics per Stack			XVE Array
GPU Stack	GPU Adapter		XVE Array
Active(%)	XVE Array Stalled(%)	XVE Array Idle(%)	
-----	-----	-----	-----
0	0:58:0.0 : Display controller: Intel Corporation Device 0x0bdb		
0.0%	0.0%	100.0%	
0	0:41:0.0 : Display controller: Intel Corporation Device 0x0bdb		
0.1%	3.8%	96.0%	
GPU L3 Bandwidth Bound: 0.2% of peak value			

## Running VTune on a Multi-Stack GPU

The Intel® Data Center GPU Max Series has a multi-stack 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 stacks set the following environment variable.

```
export AMPLXE_EXPERIMENTAL=gpu-multi-tile-metrics
```

On a multi-stack 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 stacks.

To use implicit scaling, set the following environment variable.

```
export EnableImplicitScaling=1
```

Explicit scaling requires code modifications, so you need to specify to which stack 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-stack attribution of GPU HW metrics and compute kernels

### Application Scaling

To test whether application is scaling on two stacks. First try running on a single stack 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 stack vs two stacks with implicit scaling.

Because of NUMA (and other issues), 2 stacks may not scale without tuning. There are some architecture details that you need to consider:

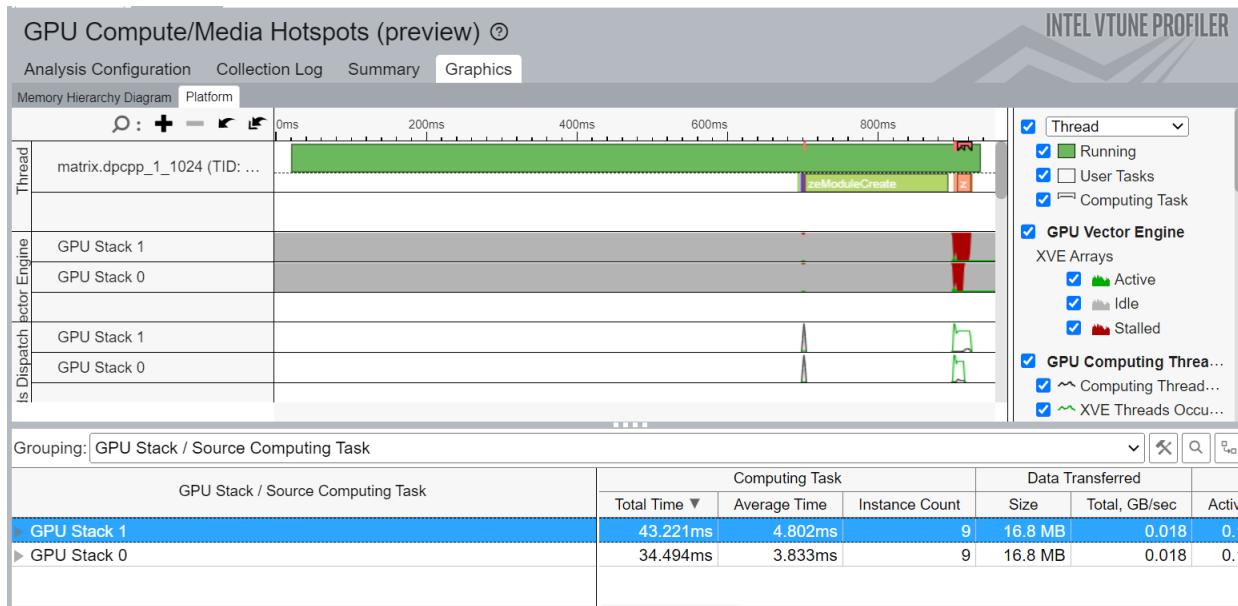
- Each stack on an Intel® Data Center GPU Max has its own L3 cache.
- These caches are *not coherent*.
- USM allocations are managed by the GPU driver.
- A device cannot access memory allocated on another device.

To understand your kernels performance it is important to understand:

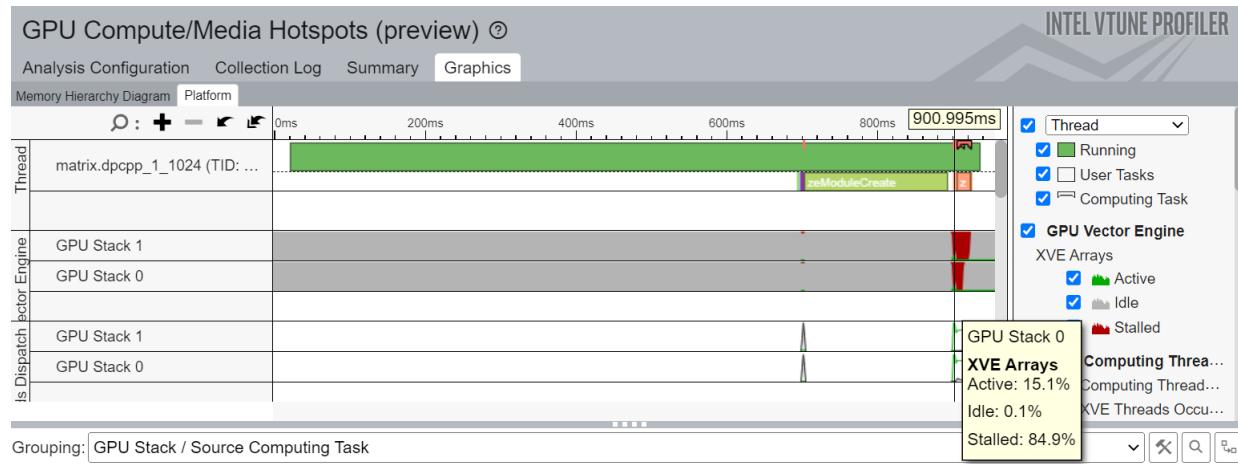
- How kernels utilize the execution resources of different stacks.
- How allocations are spread across the memory of different stacks.
- Where allocations “live” and how they move around the system.

### Multi-Stack Analysis

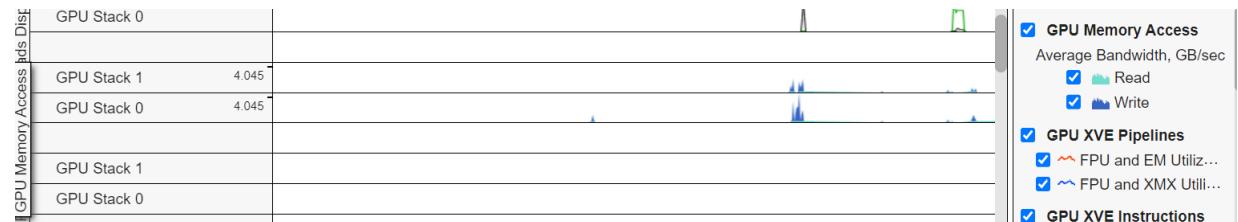
In the VTune GUI you can see all of your GPU execution units on a per stack basis, as shown below.



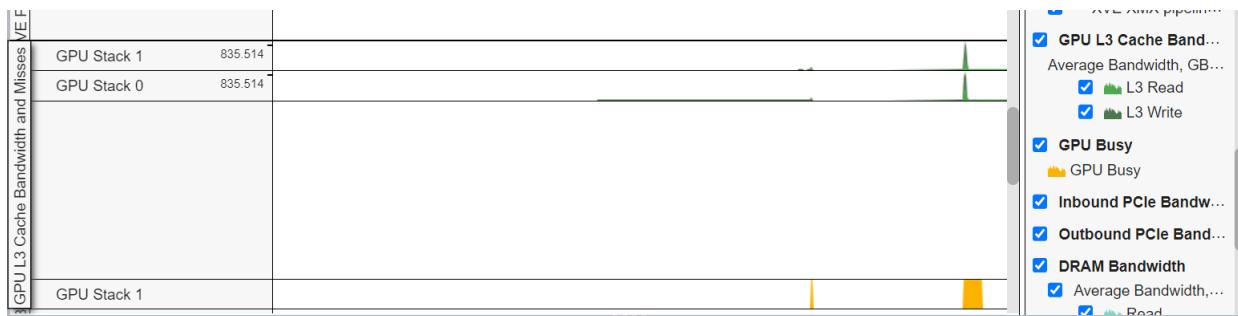
You can also correlate your activity on the stacks and see when one is idle or stalled.



VTune also shows you your read/write memory bandwidth on each stack 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 stacks are operating at different frequencies. This will require additional investigation



## 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)*

## Multi-GPU

In addition to being able to analyze multiple stacks, VTune can also analyze multiple GPUs. Multiple GPUs are used 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 and run `lspci` on Linux to find out the bus/device/function (BDF) of its adapter.

You need to use this format [B:D:F] when specifying your GPU on collection and 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>
```

In the GUI, use the **Target GPU** pulldown menu to specify the device you want to profile. The pulldown menu displays only when VTune Profiler detects multiple GPUs running on the system. The menu then displays the name of each GPU with the BDF of its adapter. If you do not select a GPU, VTune Profiler selects the most recent device family in the list by default.

## Hardware-assist Stall Sampling

Hardware-assisted stall sampling is a new performance monitoring capability implemented in Intel® Data Center GPU Max Series. The following section covers some of the details.

- **Hardware-assisted Stall Sampling**

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](#).

## Hardware-assist Stall Sampling

Hardware-assisted stall sampling is a new performance monitoring capability implemented in Intel® Data Center GPU Max Series. The following section covers some of the details.

- [Hardware-assisted Stall Sampling](#)

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](#).

## Hardware-assisted Stall Sampling

In Intel VTune Profiler, GPU Offload analysis can be used to find out offload efficiency and GPU Compute/Media Hotspots analysis covers the execution efficiency on GPU. The characterization of GPU execution efficiency is partly addressed by VTune Profiler GPU Compute/Media Hotspots Characterization mode in the previous section. Except for the Dynamic Instruction Count option, the Characterization mode is hardware counter based and its granularity is a compute kernel. While it allows estimation of execution efficiency at kernel level (e.g., Active/Stalled/Idle metric), the Characterization mode provides limited performance insights at source/assembly level. As a result, performance optimization for medium or large kernels could become a time-consuming task, based on assumptions and guesswork, with multiple iterations of source-level changes that help address inefficiencies observed at the kernel or application level. Source Analysis mode in GPU Compute/Media Hotspot analysis can be a good next step after Characterization mode to look closer at time consuming kernels and pinpoint source lines or instructions which take a significant amount of time, including Basic Block and Memory latency analysis and HW-assisted stall sampling.

HW-assisted stall sampling is a new performance monitoring capability implemented in Intel® Data Center GPU Max Series, which statistically correlates Xe-Vector Engine (XVE) stall events to the executed instructions and breaks down the stall events by different stall reasons. In this sampling mechanism, XVEs are sampled based on a fixed, configurable number of cycles that have been checked for stalls. An XVE is considered to be stalled, if and only if, there is at least one thread loaded, but no thread can execute in the sampled cycle. If there is an XVE stall, a representative thread out of the hardware threads is selected based on a proprietary heuristic, and the Instruction Pointer of the selected thread is recorded along with the cause for the stall. An XVE can stall due to several reasons such as on an instruction fetch, send operation on a barrier, etc. In Intel® Data Center GPU Max Series, XVE stall sampling provides eight counters which count stalls due to eight different reasons as shown in Table 3. With the most fine-grain interval, HW-assisted stall sampling is expected to have an overhead of ~10% of kernel/application execution.

### Intel® GPU Compute Throughput Rates (Ops/clock/EU)

HW Stall Reason	Description
ACTIVE	Actively executing in at least one pipeline
INST_FETCH	Stalled due to an instruction fetch operation
SYNC	Stalled due to sync operation
SCOREBOARD ID	Stalled due to memory dependency or internal XVE pipeline dependency
DIST or ACC	Stalled due to internal pipeline dependency
SEND	Stalled due to memory dependency or internal pipeline dependency for send
PIPESTALL	Stalled due to XVE pipeline
CONTROL	Stalled due to branch

HW Stall Reason	Description
OTHER	Stalled due to any other reason

## Configuring and Running in GUI

- Set the environment variable (AMPLXE\_EXPERIMENTAL=gpu-stall-sampling)\*
- Launch VTune Profiler and click New Project from the Welcome page.
- Create a Project dialog box opens.
- Specify a project name and a location for your project and click Create Project.
- The Configure Analysis window opens.
- Make sure the Local Host is selected in the WHERE pane.
- In the WHAT pane, make sure the Launch Application target is selected and specify application name and parameters in the correspondent fields.
- In the HOW pane, select GPU Compute/Media Hotspots analysis type from the Accelerators group.
- Make sure that Source Analysis mode is selected on the GPU Compute/Media Hotspots analysis configuration pane. Choose “Stall Sampling” option from drop box correspondent to Source Analysis mode.

## Configuring and Running in CLI

- Set environment by sourcing the script:

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

- Set the environmental variable since stall sampling is still an experimental feature:

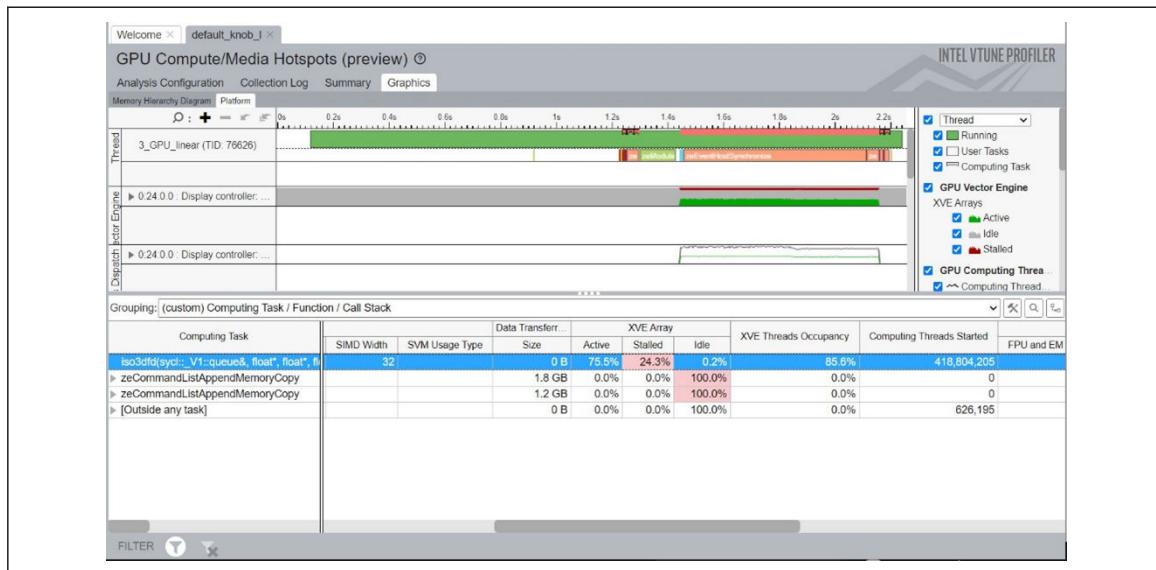
```
export AMPLXE_EXPERIMENTAL=gpu-stall-sampling
```

- Run the analysis command:

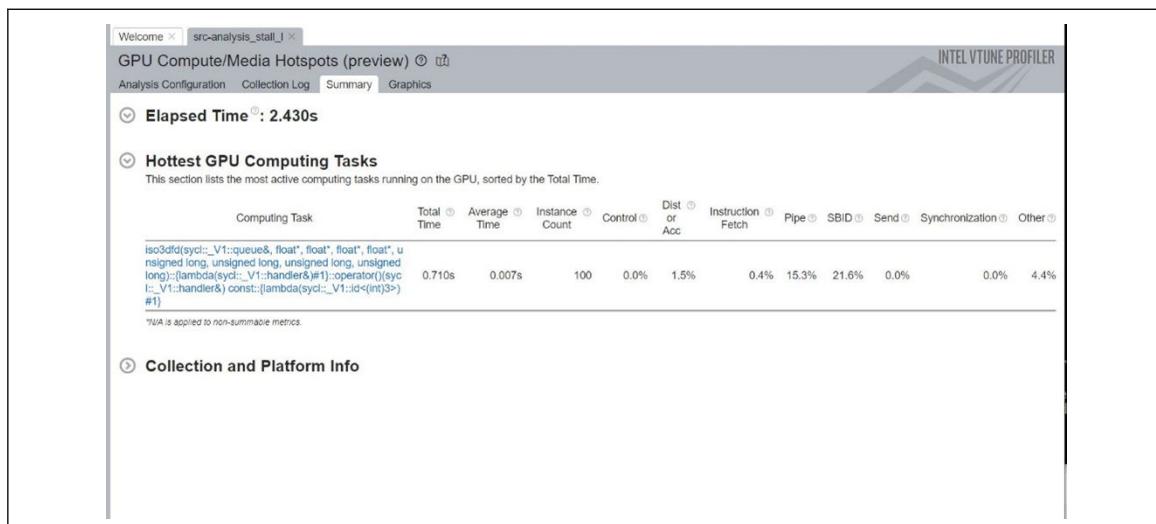
```
vtune -collect gpu-hotspots -knob profiling-mode=source-analysis -knob source-analysis=stall-sampling -- <app> [parameters]
```

## Visualizing Data

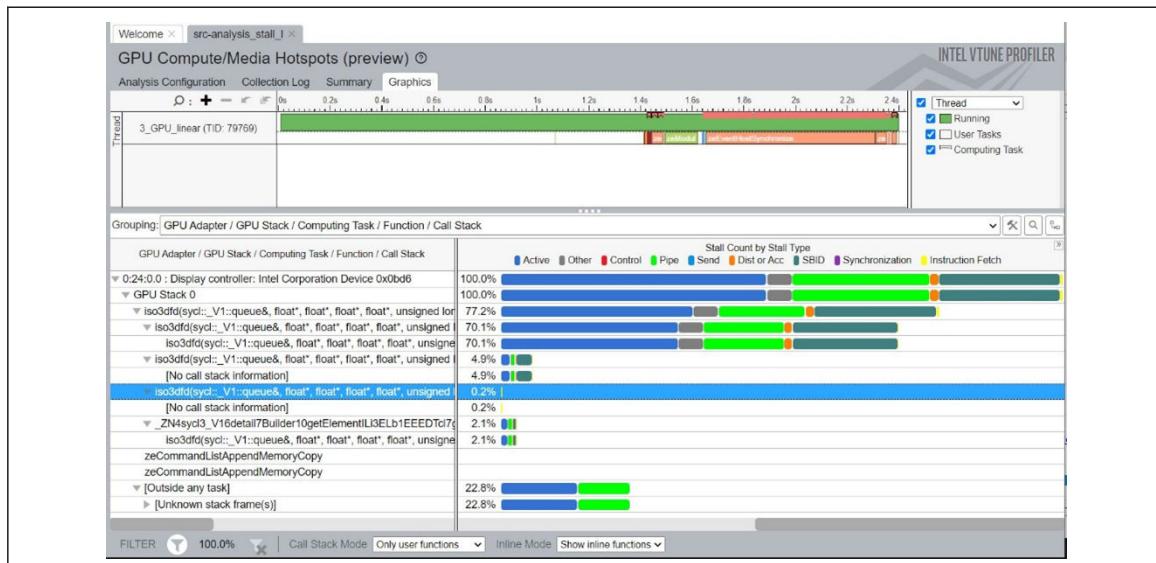
The use case of iso3dfd (linear indexing version) from oneAPI sample ([https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/C%2B%2BSYCL/StructuredGrids/guided\\_iso3dfd\\_GPUOptimization](https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/C%2B%2BSYCL/StructuredGrids/guided_iso3dfd_GPUOptimization)) is used to illustrate how to visualize the data from HW-assisted Stall Sampling analysis. In this example, the Characterization mode of GPU Compute/Media Hotspots shows that about a quarter of the GPU time is spent in stalls during the application execution. Stall Sampling is followed to identify where and why the application is stalled and gain insights to the behavior of the code at an instruction level.



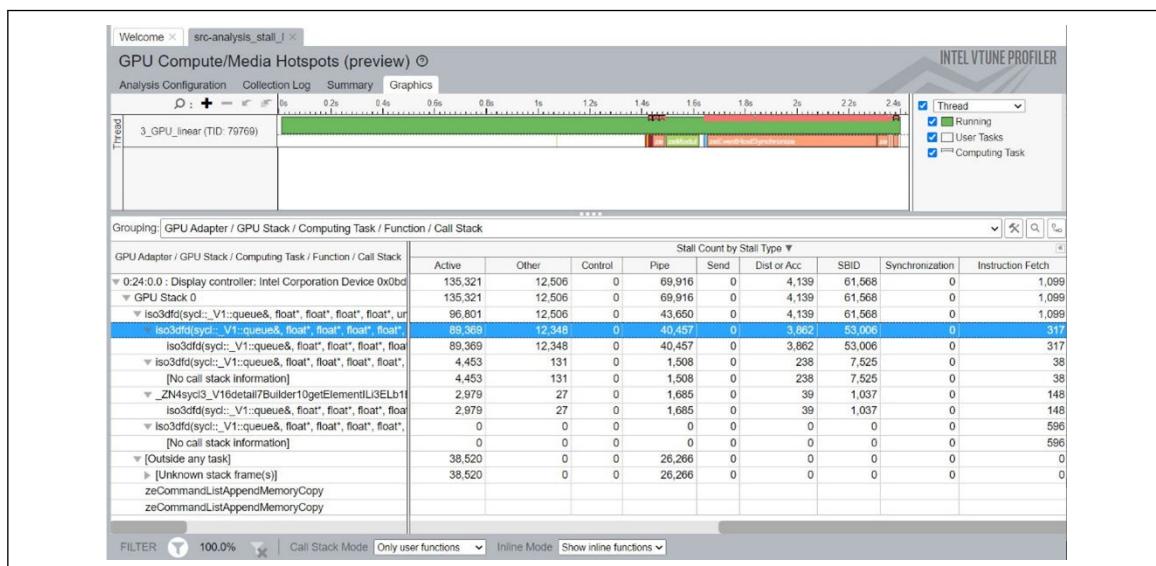
The main views which are exploited by HW-Stall Sampling are the Summary and Graphics tabs. In the Summary tab, the most time-consuming GPU tasks are displayed along with their associated percentage breakdown of each stall reason. In our example, the two main reasons for stall in the hottest computing task are Pipestall and Scoreboard ID.



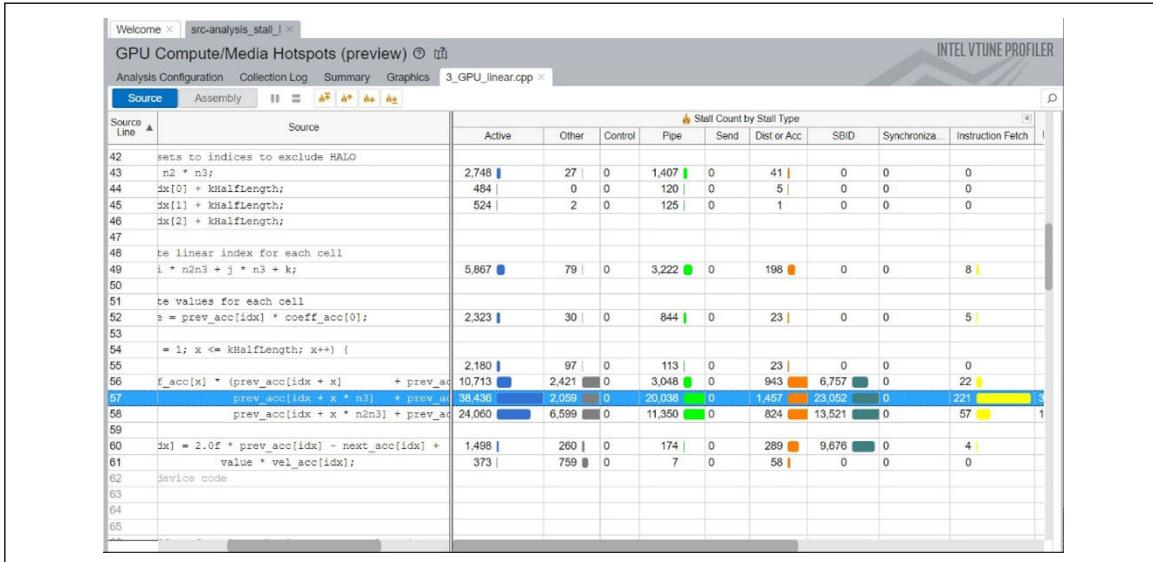
The Graphics tab shows a grid with compute tasks (kernels) and are active and stalled by reason sample count or percent from samples for each compute kernel aggregated by all instances. The grid grouping GPU Adapter/GPU Stack/Compute Task/Function/Call Stack expands a kernel row to see statistics by functions and GPU call stacks at GPU Stack and Card levels.



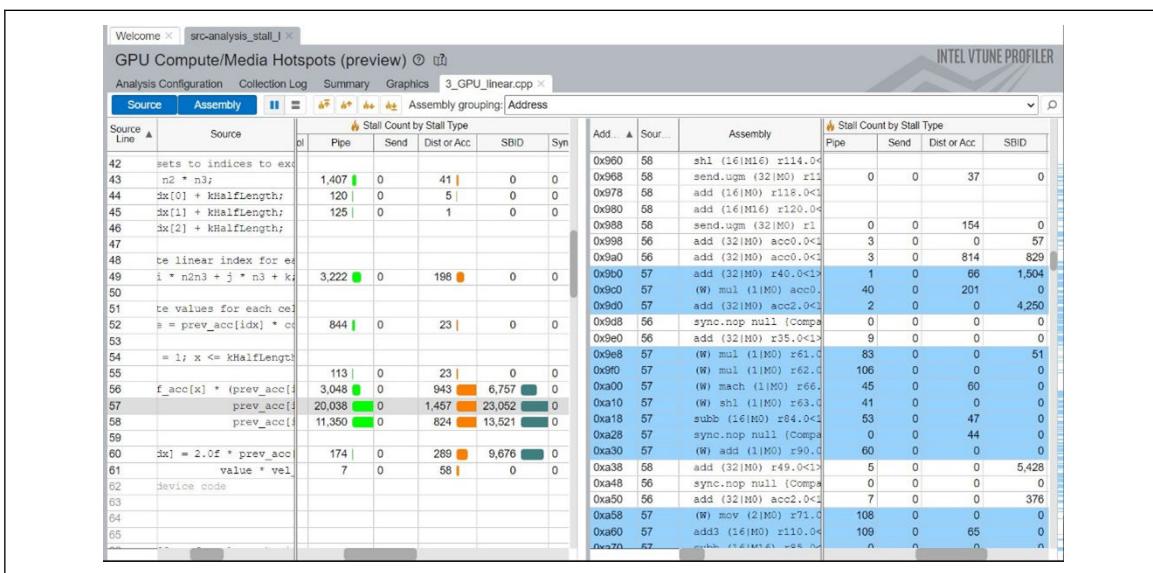
It is also possible to expand Stall Count by Stall Type column to have a breakdown by stall type in the columns to see precise numbers and do sorting for a specific reason.



To see active or stalled samples distributed by source/assembly, go to Source View by double-clicking on a compute task or function of interest.



To see Assembly View or Source/Assembly side-by-side, use the "Assembly" toggle button. It is worth to note that Source and Assembly pane are synchronized on a row selection.



## Hardware-assisted Stall Sampling

In Intel VTune Profiler, GPU Offload analysis can be used to find out offload efficiency and GPU Compute/Media Hotspots analysis covers the execution efficiency on GPU. The characterization of GPU execution efficiency is partly addressed by VTune Profiler GPU Compute/Media Hotspots Characterization mode in the previous section. Except for the Dynamic Instruction Count option, the Characterization mode is hardware counter based and its granularity is a compute kernel. While it allows estimation of execution efficiency at kernel level (e.g., Active/Stalled/Idle metric), the Characterization mode provides limited performance insights at source/assembly level. As a result, performance optimization for medium or large kernels could become a time-consuming task, based on assumptions and guesswork, with multiple iterations of source-level changes that help address inefficiencies observed at the kernel or application level. Source Analysis mode in GPU Compute/Media Hotspot analysis can be a good next step after Characterization mode to look closer at time consuming kernels and pinpoint source lines or instructions which take a significant amount of time, including Basic Block and Memory latency analysis and HW-assisted stall sampling.

HW-assisted stall sampling is a new performance monitoring capability implemented in Intel® Data Center GPU Max Series, which statistically correlates Xe-Vector Engine (XVE) stall events to the executed instructions and breaks down the stall events by different stall reasons. In this sampling mechanism, XVEs

are sampled based on a fixed, configurable number of cycles that have been checked for stalls. An XVE is considered to be stalled, if and only if, there is at least one thread loaded, but no thread can execute in the sampled cycle. If there is an XVE stall, a representative thread out of the hardware threads is selected based on a proprietary heuristic, and the Instruction Pointer of the selected thread is recorded along with the cause for the stall. An XVE can stall due to several reasons such as on an instruction fetch, send operation on a barrier, etc. In Intel® Data Center GPU Max Series, XVE stall sampling provides eight counters which count stalls due to eight different reasons as shown in Table 3. With the most fine-grain interval, HW-assisted stall sampling is expected to have an overhead of ~10% of kernel/application execution.

#### Intel® GPU Compute Throughput Rates (Ops/clock/EU)

HW Stall Reason	Description
ACTIVE	Actively executing in at least one pipeline
INST_FETCH	Stalled due to an instruction fetch operation
SYNC	Stalled due to sync operation
SCOREBOARD ID	Stalled due to memory dependency or internal XVE pipeline dependency
DIST or ACC	Stalled due to internal pipeline dependency
SEND	Stalled due to memory dependency or internal pipeline dependency for send
PIPESTALL	Stalled due to XVE pipeline
CONTROL	Stalled due to branch
OTHER	Stalled due to any other reason

#### Configuring and Running in GUI

- Set the environment variable (AMPLXE\_EXPERIMENTAL=gpu-stall-sampling)\*
- Launch VTune Profiler and click New Project from the Welcome page.
- Create a Project dialog box opens.
- Specify a project name and a location for your project and click Create Project.
- The Configure Analysis window opens.
- Make sure the Local Host is selected in the WHERE pane.
- In the WHAT pane, make sure the Launch Application target is selected and specify application name and parameters in the correspondent fields.
- In the HOW pane, select GPU Compute/Media Hotspots analysis type from the Accelerators group.
- Make sure that Source Analysis mode is selected on the GPU Compute/Media Hotspots analysis configuration pane. Choose "Stall Sampling" option from drop box correspondent to Source Analysis mode.

#### Configuring and Running in CLI

- Set environment by sourcing the script:

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

- Set the environmental variable since stall sampling is still an experimental feature:

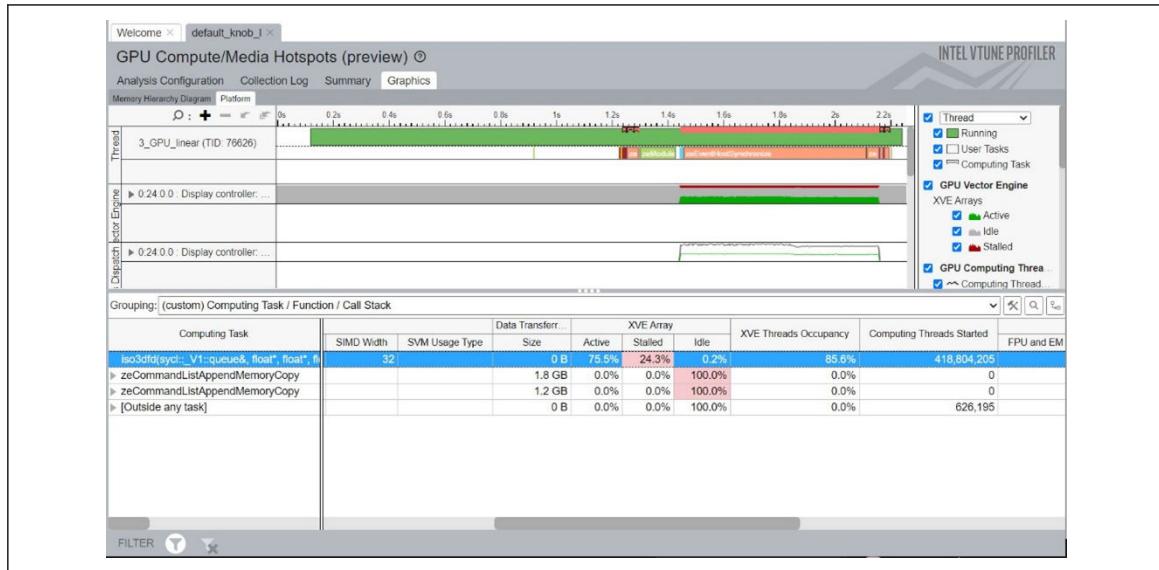
```
export AMPLXE_EXPERIMENTAL=gpu-stall-sampling
```

- Run the analysis command:

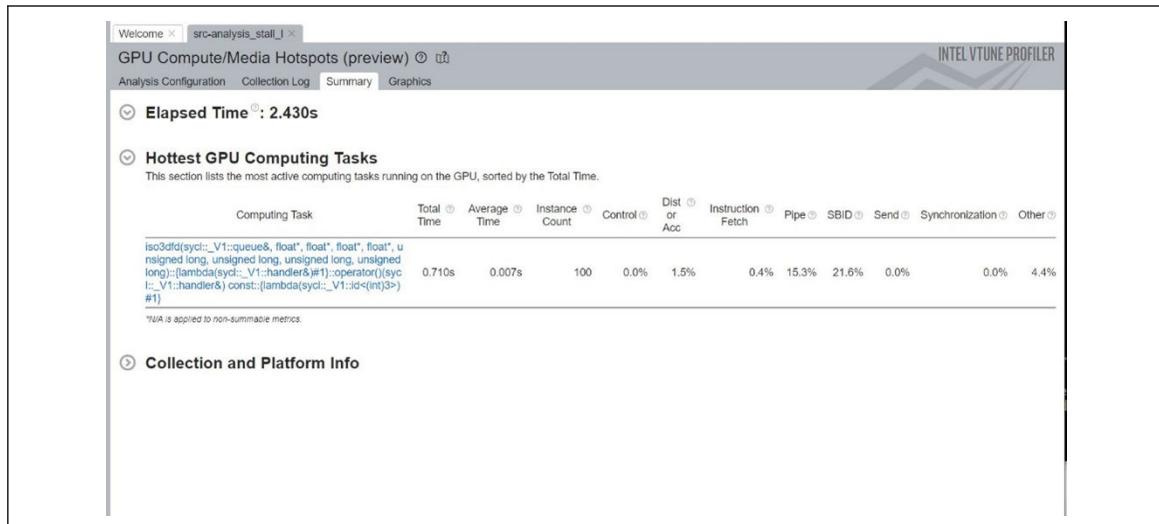
```
vtune -collect gpu-hotspots -knob profiling-mode=source-analysis -knob source-analysis=stall-sampling -- <app> [parameters]
```

## Visualizing Data

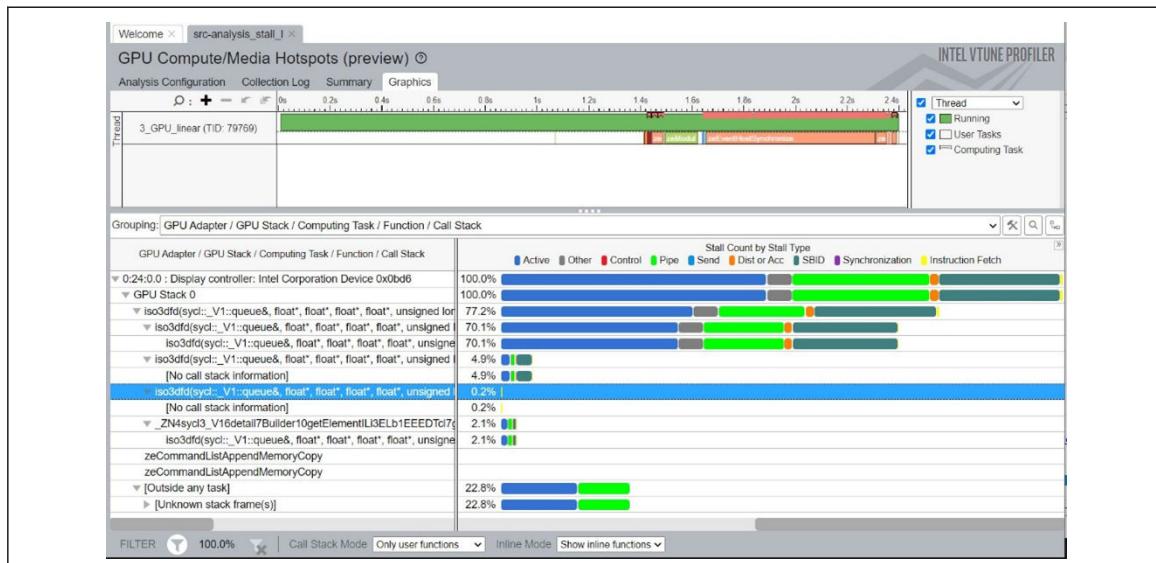
The use case of iso3dfd (linear indexing version) from oneAPI sample ([https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/C%2B%2BSYCL/StructuredGrids/guided\\_iso3dfd\\_GPUOptimization](https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/C%2B%2BSYCL/StructuredGrids/guided_iso3dfd_GPUOptimization)) is used to illustrate how to visualize the data from HW-assisted Stall Sampling analysis. In this example, the Characterization mode of GPU Compute/Media Hotspots shows that about a quarter of the GPU time is spent in stalls during the application execution. Stall Sampling is followed to identify where and why the application is stalled and gain insights to the behavior of the code at an instruction level.



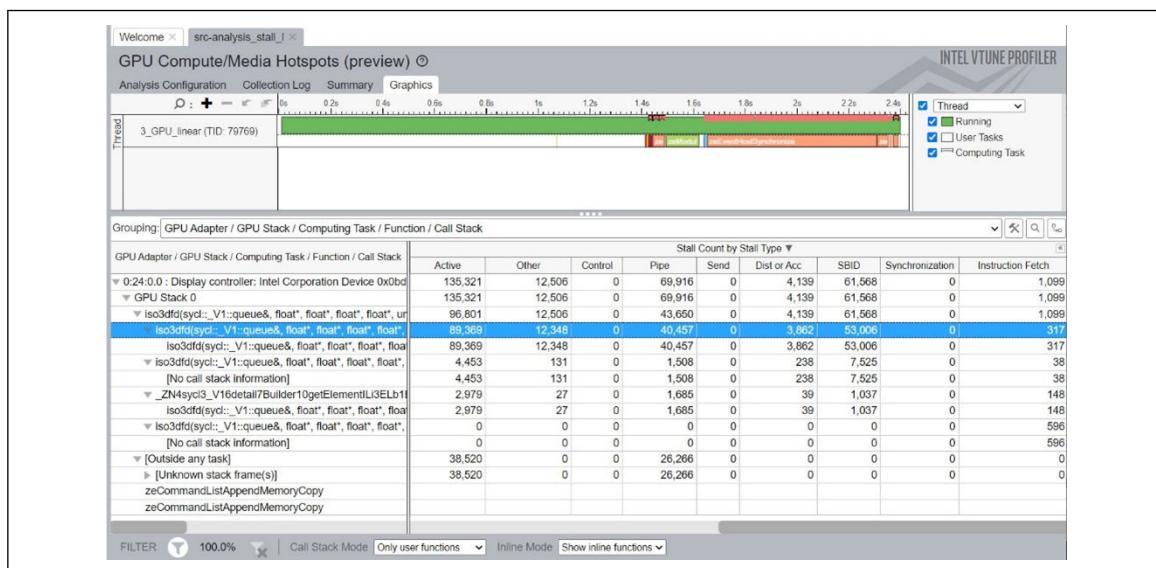
The main views which are exploited by HW-Stall Sampling are the Summary and Graphics tabs. In the Summary tab, the most time-consuming GPU tasks are displayed along with their associated percentage breakdown of each stall reason. In our example, the two main reasons for stall in the hottest computing task are Pipe stall and Scoreboard ID.



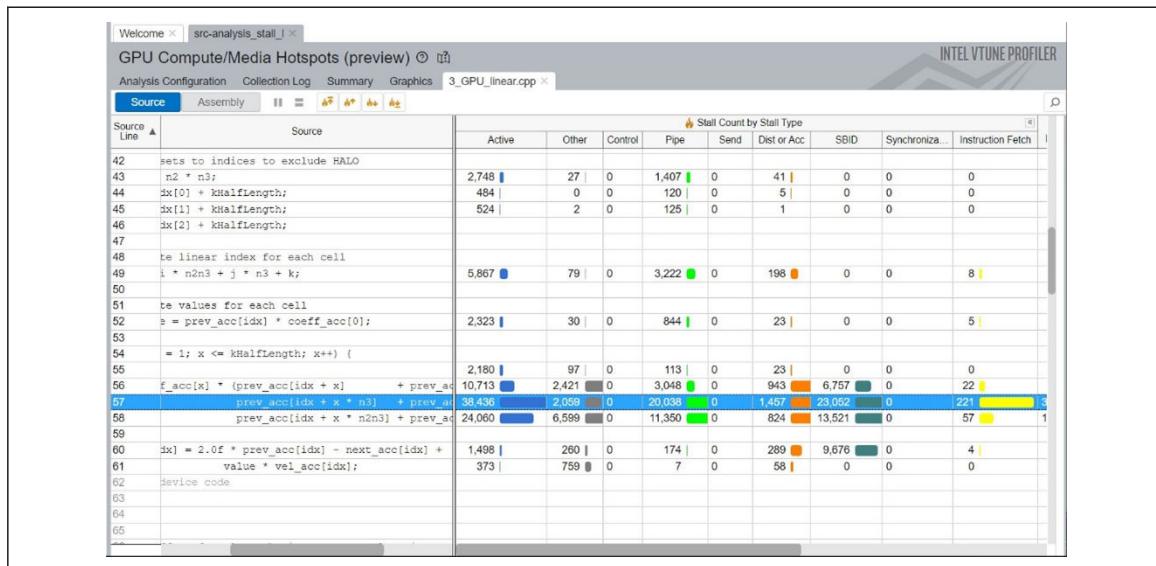
The Graphics tab shows a grid with compute tasks (kernels) and are active and stalled by reason sample count or percent from samples for each compute kernel aggregated by all instances. The grid grouping GPU Adapter/GPU Stack/Compute Task/Function/Call Stack expands a kernel row to see statistics by functions and GPU call stacks at GPU Stack and Card levels.



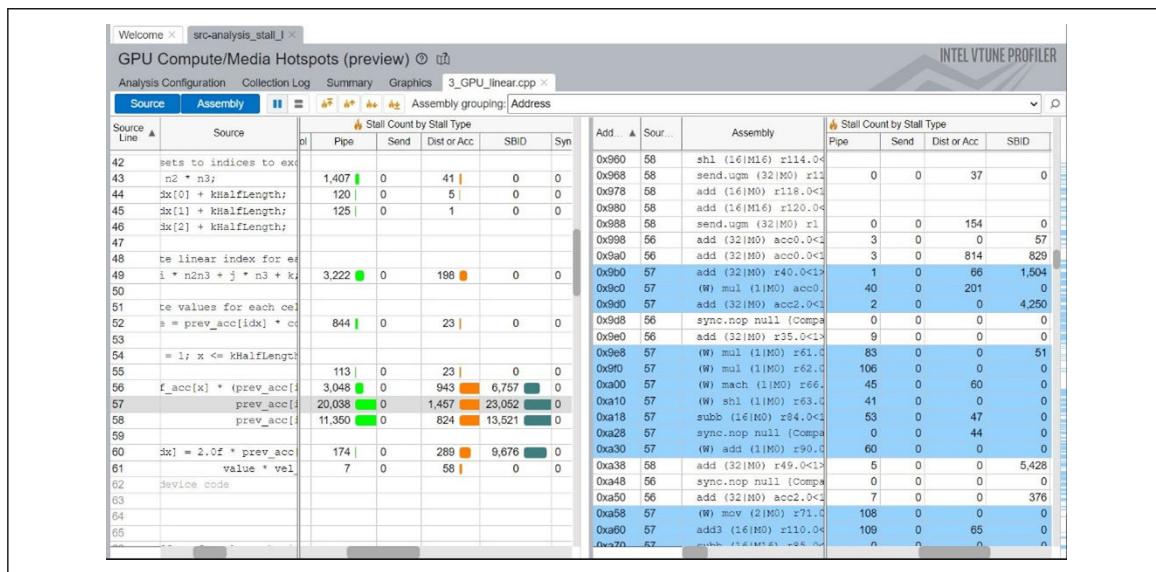
It is also possible to expand Stall Count by Stall Type column to have a breakdown by stall type in the columns to see precise numbers and do sorting for a specific reason.



To see active or stalled samples distributed by source/assembly, go to Source View by double-clicking on a compute task or function of interest.



To see Assembly View or Source/Assembly side-by-side, use the “Assembly” toggle button. It is worth to note that Source and Assembly pane are synchronized on a row selection.



## Intel® Advisor

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

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

**Prerequisites:** To use Intel® Advisor, first set the tool environment:

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

The rest of this chapter covers the two features introduced above, and a detailed recipe on using GPU Roofline Insights to analyze and optimize memory-bound applications.

- [Identify Regions to Offload to GPU with Offload Modeling](#)
- [Run a GPU Roofline Analysis](#)
- [Optimize Memory-bound Applications with GPU Roofline](#)

## [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 can profitably 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 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 by using the `advisor` command line interface, or by 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](#).

You are recommended to run the GPU-to-GPU performance modeling to analyze SYCL, OpenMP target, and OpenCL application because it provides more accurate estimations. The GPU-to-GPU modeling analyzes only GPU compute kernels and ignores the application parts executed on a CPU. Below is an example of model performance of the C++ Matrix Multiply target application running on an Intel® HD Graphics 630 GPU for a different GPU, Intel® Data Center GPU Max Series.

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

1. Collect application performance metrics with `collect.py`:

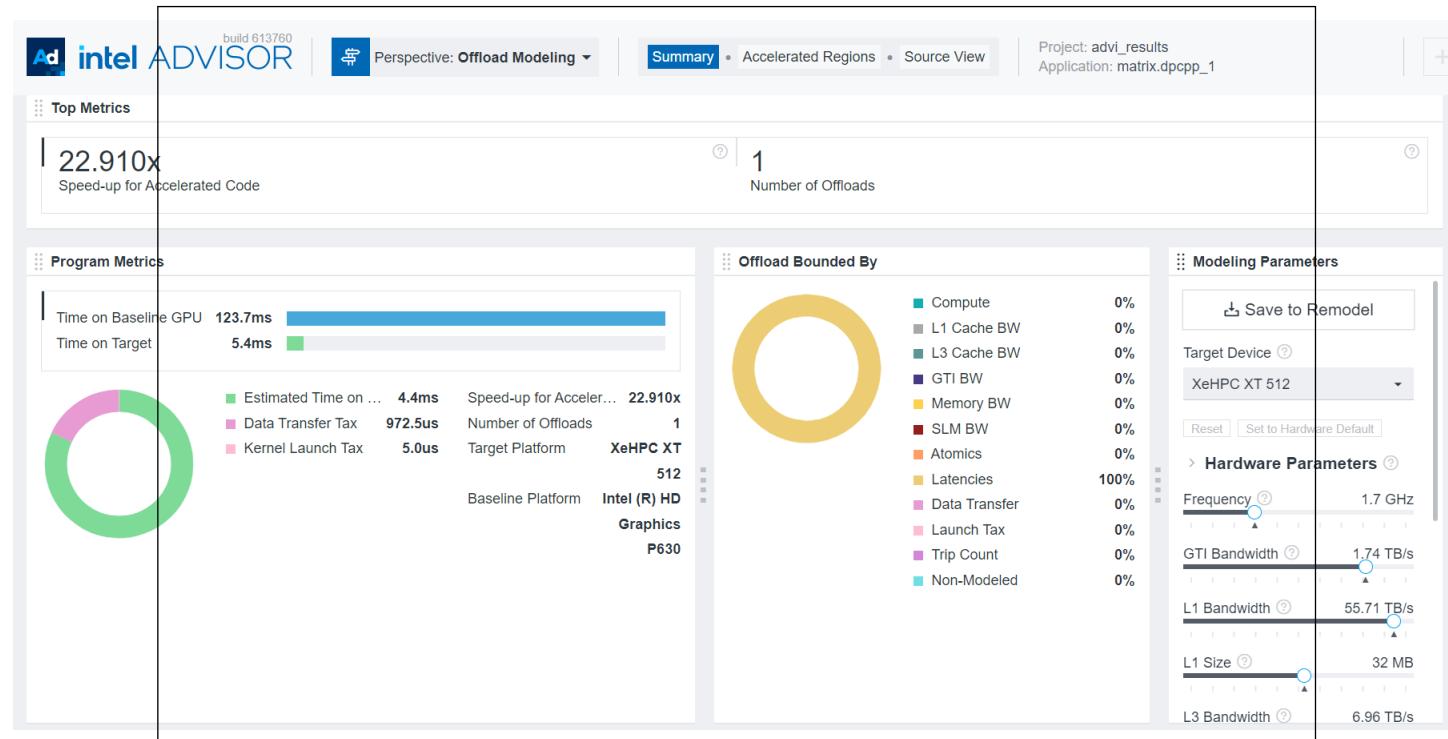
```
advisor-python $APM/collect.py ./advi_results --gpu --config=pvc_xt_512xve -- ./matrix.dpcpp
```

2. Model your application performance on a GPU with `analyze.py`:

```
advisor-python $APM/analyze.py ./advi_results --gpu --config=pvc_xt_512xve
```

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 `advi_results/e000/pp000/data.csv`.

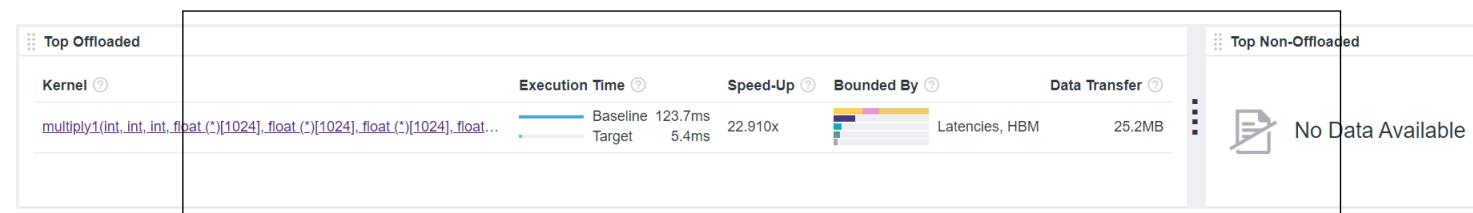
## Intel® Advisor GUI, Offload Advisor



For example, in the Summary section of the report, review the following:

- The original execution time on the baseline platform, the predicted execution time on the target GPU accelerator, the number of offloaded regions, and the estimated speedup in the Program metrics pane. For Matrix Multiply, Intel® Advisor reports a 22.9x potential speedup.
- What the offloads are bounded by. This pane reports the main limiting factors that prevent your application from achieving better performance on a target device. 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.

## Top Regions



Go to the Offloaded Regions tab to view the detailed measured and estimated metrics for the code regions recommended for offloading. It also reports the estimated amount of data transferred for the code regions and the corresponding offload taxes.

```
// Basic matrix multiply
void multiply1(int msizes, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
              TYPE c[][NUM], TYPE t[][NUM]) {
    int i, j, k;
```

```

// Declare a deviceQueue
sycl::queue q(sycl::default_selector_v, exception_handler);
cout << "Running on " << q.get_device().get_info<sycl::info::device::name>()
    << "\n";
// Declare a 2 dimensional range
sycl::range<2> matrix_range{NUM, NUM};

// Declare 3 buffers and Initialize them
sycl::buffer<TYPE, 2> bufferA((TYPE *)a, matrix_range);
sycl::buffer<TYPE, 2> bufferB((TYPE *)b, matrix_range);
sycl::buffer<TYPE, 2> bufferC((TYPE *)c, matrix_range);
// Submit our job to the queue
q.submit([&](auto &h) {
    // Declare 3 accessors to our buffers. The first 2 read and the last
    // read_write
    sycl::accessor accessorA(bufferA, h, sycl::read_only);
    sycl::accessor accessorB(bufferB, h, sycl::read_only);
    sycl::accessor accessorC(bufferC, h);

    // Execute matrix multiply in parallel over our matrix_range
    // ind is an index into this range
    h.parallel_for(matrix_range, [=](sycl::id<2> ind) {
        int k;
        for (k = 0; k < NUM; k++) {
            // Perform computation ind[0] is row, ind[1] is col
            accessorC[ind[0]][ind[1]] +=
                accessorA[ind[0]][k] * accessorB[k][ind[1]];
        }
    });
    }).wait_and_throw();
} // multiply
}

```

## Run a GPU Roofline Analysis

To estimate actual performance of a GPU application against hardware-imposed ceilings, 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.

## Linux OS Users

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
```

1. Disable time limits to run the OpenCL kernel for a longer period:

```
sudo sh -c "echo N> /sys/module/i915/parameters/enable_hangcheck"
```

## All Users

1. Make sure that your SYCL code runs correctly on the GPU. To check which hardware you are running on, add the following to your SYCL code and run it:

```
sycl::default_selector selector;
sycl::queue queue(delector);
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.

## Top Hotspots pane

Kernel	Elapsed Time ...	GFLOPS	GINTOPS	Global/Local	Active/Stalled/I...
multiply1_1(int,...)	<0.01s	396.972	332.716	1024 x 1024/10...	8.1/86.0/5.9

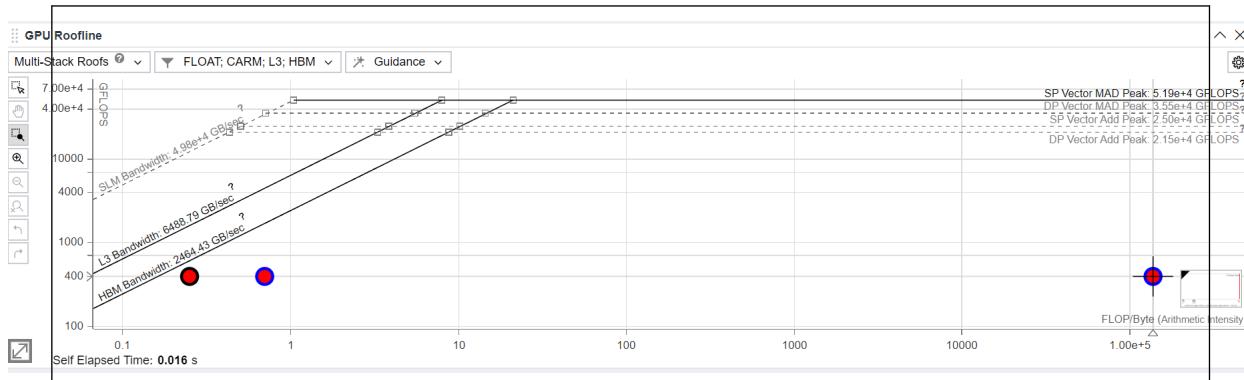
See how efficiently your application uses execution units for each GPU Kernel in the GPU Kernels pane. Clicking on a top hotspot in the Top Hotspot will show the XVE usage in GPU Kernels pane.

## GPU Kernels pane

Kernel	XVE Array			2 Stacks Active	XVE Threading Occupancy	Computing Threads Started	XVE Instructions	
	Active	Stalled	Idle				FPU and EM Active	FPU and Matrix E
[Outside any task]	0.0%	0.0%	100.0%		0.0%	393,504	0.0%	0.0%
zeCommandListAppendBarrier	0.0%	75.2%	24.8%	99.7%	57.1%	130,784	0.0%	0.0%
► zeCommandListAppendMemoryCopyRegion	0.0%	96.9%	3.0%	54.4%	73.7%	130,688	0.0%	0.0%
<b>multiply1_1(int, int, int, float (*)[1024], float (*)[1024], float (*)[1024])</b>	<b>8.1%</b>	<b>86.0%</b>	<b>5.9%</b>	<b>97.5%</b>	<b>90.0%</b>	<b>32,768</b>	<b>0.1%</b>	<b>0.0%</b>

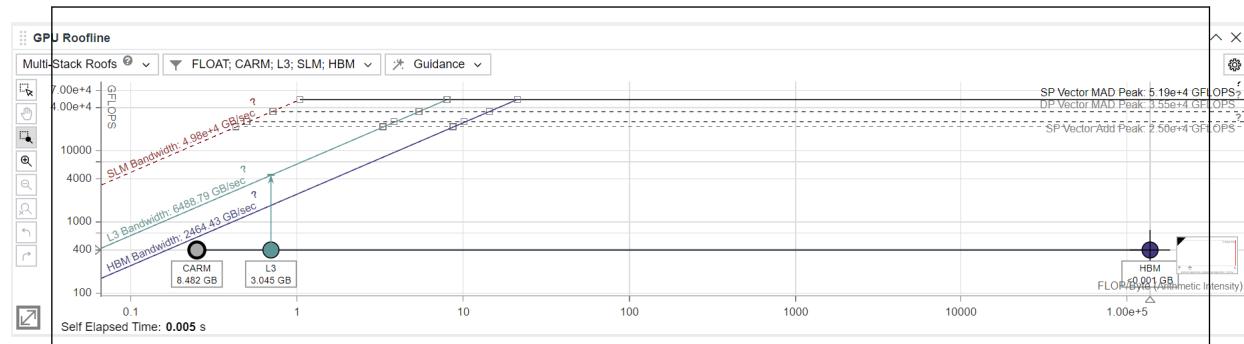
In the GPU Roofline tab, see the GPU Roofline chart and performance metrics.

## GPU Roofline chart and performance metrics



- The Matrix Multiply application achieves 396.97 GFLOPS of performance. It uses global memory and is not optimized for local (SLM) memory because the application uses a global accessor.
- This Roofline view shows 3 dots for a single loop, each dot having the same performance (vertical position) and different arithmetic intensities (horizontal position).
- The application is bounded by L3 bandwidth as represented by the green circle on the Roofline.
- The gray circle on the left is CARM dot and it considers memory transfers between registers and L1.
- The purple circle on the right is HBM dot and it considers memory transfers between L3 and HBM.

## GPU Roofline chart and performance metrics



As the GPU Roofline chart suggests, several possible optimizations might result in more efficient memory usage:

- Use local memory (SLM).
- Use the cache blocking technique to better use SLM.

The following code is the optimized version of the Matrix Multiply application. In this version, we declare two stacks and define them as `sycl::access::target::local`. We also modify the kernel to process these stacks in some inner loops.

```
// Replaces accessorC reference with a local variable
void multiply1_1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
                 TYPE c[] [NUM], TYPE t[] [NUM]) {

    int i, j, k;

    // Declare a deviceQueue
    sycl::queue q(sycl::default_selector_v, exception_handler);
    cout << "Running on " << q.get_device().get_info<sycl::info::device::name>()
        << "\n";

    // Declare a 2 dimensional range
    sycl::range<2> matrix_range{NUM, NUM};

    // Declare 3 buffers and Initialize them
    sycl::buffer<TYPE, 2> bufferA((TYPE *)a, matrix_range);
    sycl::buffer<TYPE, 2> bufferB((TYPE *)b, matrix_range);
    sycl::buffer<TYPE, 2> bufferC((TYPE *)c, matrix_range);

    // Submit our job to the queue
    q.submit([&](auto &h) {
        // Declare 3 accessors to our buffers. The first 2 read and the last
        // read_write
        sycl::accessor accessorA(bufferA, h, sycl::read_only);
        sycl::accessor accessorB(bufferB, h, sycl::read_only);
        sycl::accessor accessorC(bufferC, h);

        // Execute matrix multiply in parallel over our matrix_range
        // ind is an index into this range
        h.parallel_for(matrix_range, [=](sycl::id<2> ind) {
            int k;
            TYPE acc = 0.0;
            for (k = 0; k < NUM; k++) {
                // Perform computation ind[0] is row, ind[1] is col
                acc += accessorA[ind[0]][k] * accessorB[k][ind[1]];
            }
            accessorC[ind[0]][ind[1]] = acc;
        });
        }).wait_and_throw();
    }

    // Replaces accessorC reference with a local variable and adds matrix tiling
    void multiply1_2(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
                     TYPE c[] [NUM], TYPE t[] [NUM]) {
        int i, j, k;

        // Declare a deviceQueue
        sycl::queue q(sycl::default_selector_v, exception_handler);
        cout << "Running on " << q.get_device().get_info<sycl::info::device::name>()
            << "\n";

        // Declare a 2 dimensional range
```

```

sycl::range<2> matrix_range{NUM, NUM};
sycl::range<2> tile_range{MATRIXTILESIZE, MATRIXTILESIZE};

// Declare 3 buffers and Initialize them
sycl::buffer<TYPE, 2> bufferA((TYPE *)a, matrix_range);
sycl::buffer<TYPE, 2> bufferB((TYPE *)b, matrix_range);
sycl::buffer<TYPE, 2> bufferC((TYPE *)c, matrix_range);

// Submit our job to the queue
q.submit([&](auto &h) {
    // Declare 3 accessors to our buffers. The first 2 read and the last
    // read_write
    sycl::accessor accessorA(bufferA, h, sycl::read_only);
    sycl::accessor accessorB(bufferB, h, sycl::read_only);
    sycl::accessor accessorC(bufferC, h);

    // Create matrix tiles
    sycl::local_accessor<TYPE, 2> aTile(
        sycl::range<2>(MATRIXTILESIZE, MATRIXTILESIZE), h);
    sycl::local_accessor<TYPE, 2> bTile(
        sycl::range<2>(MATRIXTILESIZE, MATRIXTILESIZE), h);
    // Execute matrix multiply in parallel over our matrix_range
    // ind is an index into this range
    h.parallel_for(sycl::nd_range<2>(matrix_range, tile_range),
        [=](cl::sycl::nd_item<2> it) {
            int k;
            const int numTiles = NUM / MATRIXTILESIZE;
            const int row = it.get_local_id(0);
            const int col = it.get_local_id(1);
            const int globalRow =
                MATRIXTILESIZE * it.get_group(0) + row;
            const int globalCol =
                MATRIXTILESIZE * it.get_group(1) + col;
            TYPE acc = 0.0;
            for (int t = 0; t < numTiles; t++) {
                const int tiledRow = MATRIXTILESIZE * t + row;
                const int tiledCol = MATRIXTILESIZE * t + col;
                aTile[row][col] = accessorA[globalRow][tiledCol];
                bTile[row][col] = accessorB[tiledRow][globalCol];
                it.barrier(sycl::access::fence_space::local_space);
                for (k = 0; k < MATRIXTILESIZE; k++) {
                    // Perform computation ind[0] is row, ind[1] is col
                    acc += aTile[row][k] * bTile[k][col];
                }
                it.barrier(sycl::access::fence_space::local_space);
            }
            accessorC[globalRow][globalCol] = acc;
        });
    }).wait_and_throw();
} // multiply1_2

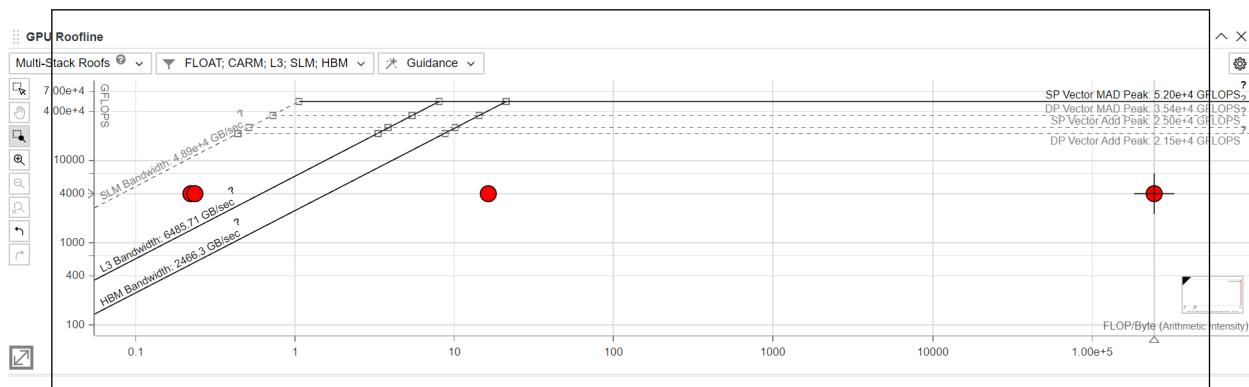
```

Save the optimized version as `multiply_1_2` and rerun the GPU Roofline. As the new chart shows:

- The optimized application achieves performance of 3994.98 GFLOPS.

- The application uses global and SLM memory, which represents the 16x16 stack. This increases memory bandwidth.

## GPU Roofline new chart



## Optimize Memory-bound Applications with GPU Roofline

This section explains how to identify performance problems of GPU applications and understand their reasons using the GPU Roofline Insights perspective of the Intel® Advisor.

When developing a GPU application with SYCL or OpenMP programming model, it is important to keep in mind kernel parallel execution. Usually, massive and uneven memory access is the main problem that limits the GPU performance. If the path between a memory level where global data is located and GPU execution units is complex, the GPU might be stalled and wait for data because of bandwidth limitations at different stages on the path. Understanding these stages and measuring data flow on the path is an essential part of performance optimization methodology. GPU Roofline Insights perspective can help you to analyze bottlenecks for each kernel and quickly find the data path stage that causes the problem. A typical method to optimize GPU execution algorithms is reconsidering data access parents.

## Memory Path in Intel® GPU Microarchitecture

Depending on a GPU generation, the compute architecture of Intel® Processor Graphics uses a system memory as a compute device memory, which is unified by sharing the same DRAM with the CPU, or a dedicated VRAM resided on a discrete GPU card.

On an integrated GPU, where DRAM is shared between CPU and GPU, global data can travel from system DRAM through last-level cache (LLC) to a graphics technology interface (GTI) on a GPU. If data is efficiently reused, it can stay in the L2 cache of a GPU where execution units can access it and fetch to an Xe Vector Engine (XVE) register file.

Assuming the fastest way to access data on a system with high bandwidth and low latency is accessing data from registers, the cache-aware Roofline model (CARM) of the Intel Advisor treats it as the most effective access with true, or pure payload, amount of data consumed by an algorithm. Let us call it *algorithmic data*. For example, for a naive implementation of the matrix multiplication algorithm, theoretically, the amount of data read for each matrix and used for calculations is:

$$D * M^3$$

where

- D is a size of a matrix element, in Bytes
- M is a size of a square matrix

If the General Register File (GRF) could theoretically fit all matrix data, the data is transferred from DRAM to GPU only once. Otherwise, the data is transferred from DRAM to the L2 cache and further in parts ordered by memory address access defined by the algorithm. If the calculations reuse data, some portion of it can stay

in cache for longer making access faster. Ideally, data used by the algorithm should fit to the L2 cache. In real life, the best situation is when data in the L2 cache is reused as much as possible, and then it is evicted to allow the next portion of data to be reused.

In some cases, data is evicted from L2 cache, but next calculations need it. This creates redundant cache traffic and adds more load to the data path bandwidth. A good indicator for such situation is the L2 cache miss metric.

As data is fetched from DRAM to the L2 cache by cache lines of 64 bytes, accessing data objects that are smaller than the cache line size or cross cache line boundaries creates excessive cache traffic because unnecessary data is yet to be fetched from DRAM. The worst-case scenario is accessing byte-size objects that are randomly resided in global memory spaces. In this case, each access brings extra 63 Bytes of unnecessary data to the L2 cache and the data path is loaded with transferring data overhead (as opposed to algorithmic data).

In addition to the L2 cache, there is shared local memory (SLM), which is a dedicated structure outside of the L2 cache (for the Gen9, it is a part of L2 physically but separated functionally) that supports the work-group local memory address space. SLM can significantly increase GPU performance, but as SLM size is limited, you should carefully select work-group size to leverage performance improvement.

Each stage has a bandwidth limitation. Usually, the further from VE, the lower the bandwidth (similar to CPU memory architecture). Depending on a particular data access pattern implemented by an algorithm, some stages can be a bottleneck for the data flow. A more complex algorithm can have a combination of bottlenecks as data access can be a combination of different patterns.

Intel Advisor GPU Roofline Insights perspective can identify bottlenecks at different stages of data transfer and map the bottlenecks to program source code helping you to focus on performance problems and optimization. In addition to source data provided by the GPU Roofline Insights, you can use other tools to identify data that creates a bottleneck.

## GPU Roofline Methodology

Intel Advisor implements the Roofline analysis in two perspectives: *CPU/Memory Roofline Insights*, which can analyze performance of CPU applications, and *GPU Roofline Insights*, which can analyze performance of GPU applications. General methodology of a Roofline model focused on the CPU/Memory Roofline is explained in the resources listed in the [Roofline Resources for Intel® Advisor Users](#). You are strongly recommended to learn the Roofline basics before continuing. This recipe explores features of the GPU Roofline Insights perspective for analyzing performance on Intel GPUs.

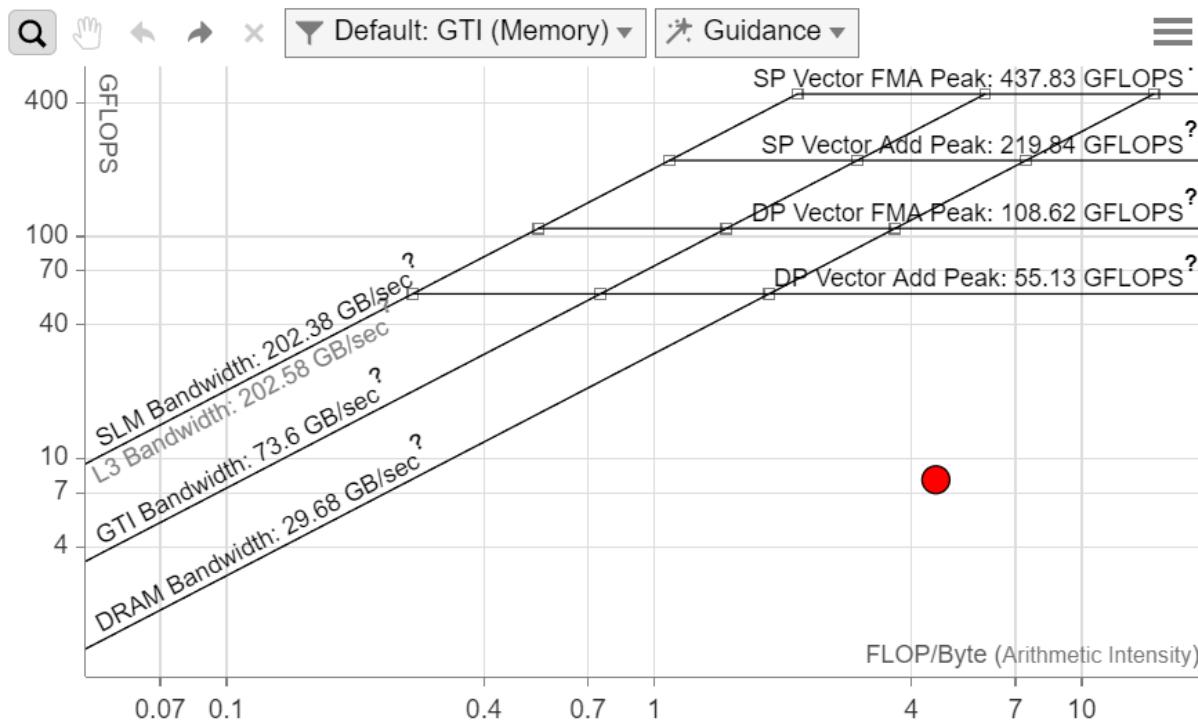
### Roofline Result Overview

Measuring GPU performance with GPU Roofline Insights is quite straightforward ([Using GPU Roofline](#)):

1. Run the GPU Roofline Insights with your preferred method: from [Intel Advisor GUI](#) or [Intel Advisor command line tool](#).

1. Open the analysis results and examine a GPU Roofline chart reported. It plots an application's achieved performance and arithmetic intensity against the machine's maximum achievable performance.

For example, for the matrix multiply application, the GPU Roofline chart filtered by GTI (memory) level has one red dot representing a GPU kernel.



In the chart, each dot represents a kernel plotted by its measured data and performance characteristics. They are a central point of analysis in two-dimensional coordinates: arithmetic intensity (X axis) and performance (Y axis). Dot location against these coordinates shows the relation of kernel's performance and its algorithm data consumption to GPU hardware limitations including its maximum computing performance and data flow bandwidth. On the chart, the hardware limitations are shown as diagonal lines, or roofs. The kernel location can help you to figure out two main things:

- If there is room for improvement to speed up kernel performance on the current GPU
- What the kernel is bound by: compute, cache, or memory bandwidth, and what you can change in the algorithm implementation to go beyond those boundaries to increase performance

This recipe describes only cases for memory- or cache-bound applications.

#### Kernel Location Calculation

It is important to know why exactly a kernel dot is located at a certain place of the chart for the following reasons.

A kernel is an implementation of an algorithm and it performs a fixed number of compute operations (such as add, mul, mad) with fixed amount of data. For example, for the matrix multiply naive implementation, assuming data is directly brought from memory, the algorithm arithmetic intensity AI is calculated as:

$$AI = M^3 / 3 \cdot M^2$$

where:

- M is a size of a square matrix
- $M^3$  is the number of operations
- $3 \cdot M^2$  is the amount of read/write data

The algorithm performance P is calculated as:

$$P = M^3 / T$$

where:

- T is time it takes for the operations to complete

- $M^3$  is the number of operations

These values AI and P define the location of the kernel dot on the graph.

Let us switch from theoretical calculations to a real-world case. Intel Advisor measures data at run time and is not aware of theoretical number of operations and amount of data the algorithm needs. Each kernel is isolated by an internal instrumenting tool and measured by API tracing. Measured performance  $P'$  is calculated as:

$$P' = I' / T'$$

where:

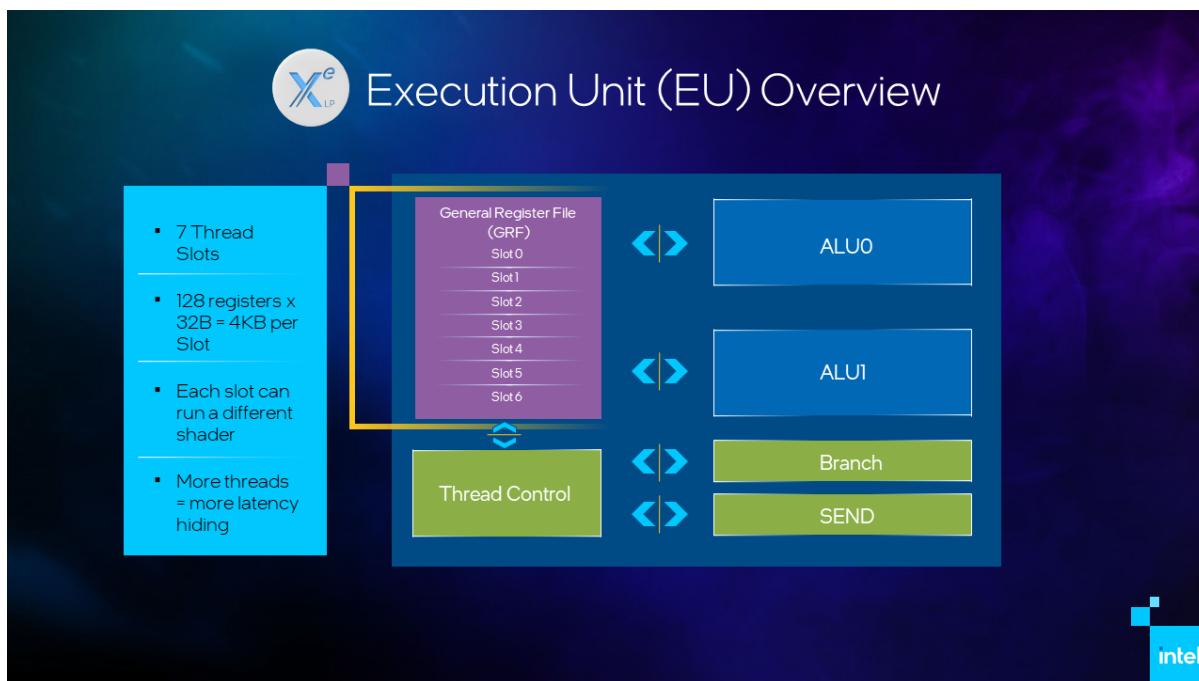
- $I'$  is measured number of executed computing instructions
- $T'$  is measured time

Measuring data used in the algorithm is easy only at the stage when VEs fetch data from the GRF because computing instructions have specific data reference syntax, which helps to calculate the amount of bytes used by the kernel. However, this data may come from different sources in the memory hierarchy in the GPU microarchitecture, and the amount of data that goes through different stages can be different.

On the Roofline chart, the kernel dot can be split into multiple dots for different memory levels. The following sections describe each memory level in detail and how Intel Advisor plots them on the Roofline chart.

How do we understand which memory level limits the algorithm execution? The algorithm performance is measured as the number of instructions  $I'$  executed during time  $T'$ , and it requires data traffic  $D_{XX}$  at each memory stage. Assuming the algorithm is memory bound, at some levels, the data flow should be close to hardware bandwidth, while at other levels, it can be less limited. To identify the most probable bottleneck of the algorithm implementation, you need to find out which dot is the closest to its corresponding memory level roof line. Note that data flows may have more than one bottleneck, and the distance between dots and their corresponding roof lines should be similar.

In the Intel Advisor, double-click a dot on the Roofline chart to quickly find the limiting roof with the shortest distance to the dot and identify the bottleneck. The tool also provides additional hints about memory limitations, but we will review them later.



## Performance Optimization Scenarios using GPU Roofline

The Roofline chart does not directly indicate what you need to change in your code to make it run faster on a GPU (although it provides some code hints, or guidance), but it shows a memory locality pattern(s) that dominate in your algorithm implementation. By examining where the kernel dots are located on the chart in relation to memory levels, you can identify the memory stage that is too narrow for the data flow and is the bottleneck. With this information, you can modify the data pattern used in your algorithm and apply, for example, using data blocking to reuse cache, avoiding multiple unnecessary data reads.

The more experience you have, the better you can understand data patterns, but there are basic cases that we can examine. Although, real-life applications do not usually show extreme behavior, like purely bound to a certain roof, as they are affected by:

- Auxiliary data transferred between memory and VEs, such as indexes of work-group item calculations, memory addresses arithmetic, loop indexes
- Data caching being more complicated as it is affected by the auxiliary data
- VE thread scheduling affecting data management

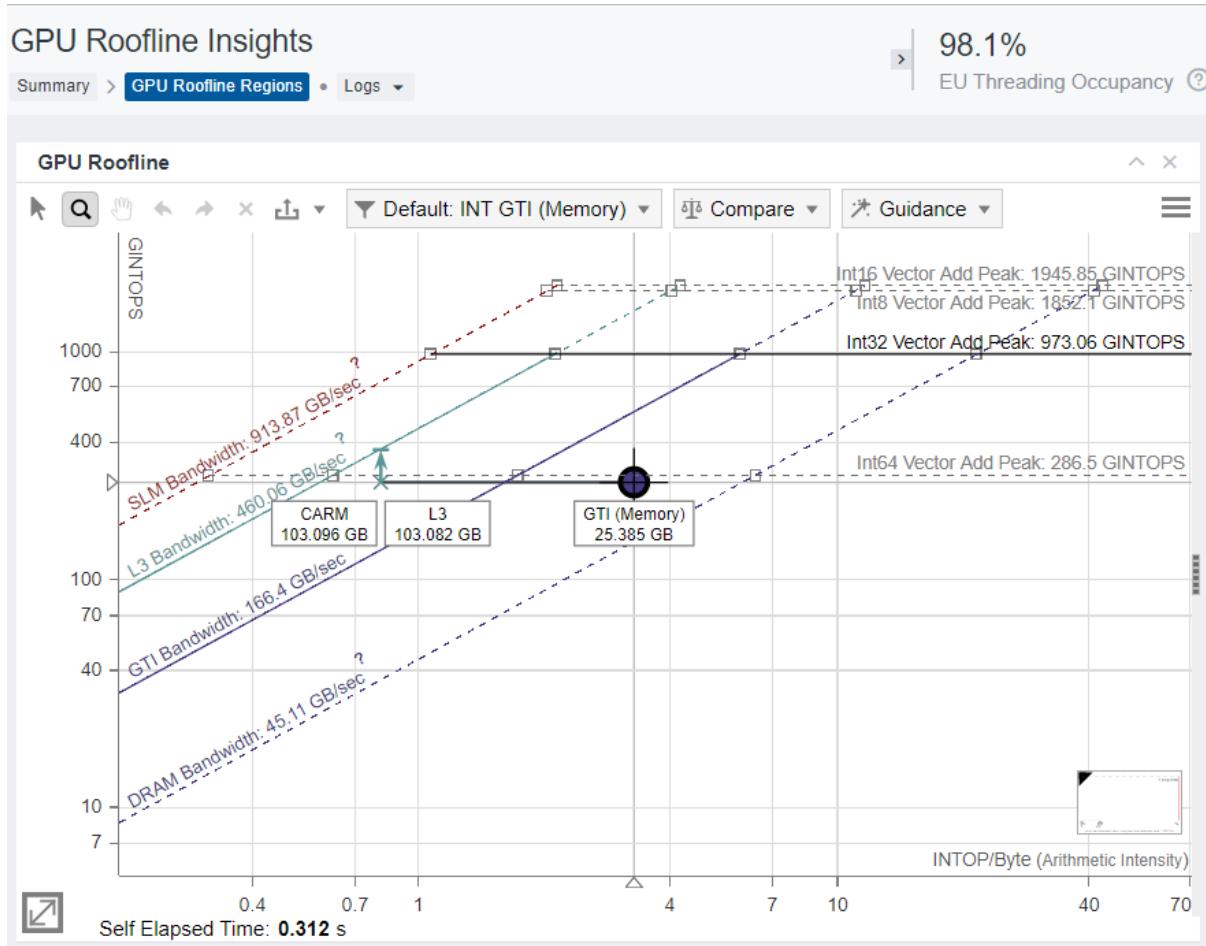
Let us consider several real-life examples of different applications and their implementations similar the theoretical cases discussed earlier.

### Data Block Optimized for the Matrix Multiply Algorithm with no SLM Usage

This implementation is a naïve matrix multiply without data blocking and is similar to the optimized kernel and data flow optimization case.

Even though data is not organized in blocks in the source code, the compiler recognizes the pattern and optimizes access to matrix arrays. As a result, we have a high level of data reuse in cache, and kernel performance is limited by the L2 cache. The Roofline chart shows one dot corresponding to a single kernel in the application. Based on its location on the chart:

- The kernel is memory bound, and the corresponding dot is close to L2 Bandwidth roof.
- GTI data traffic is four times smaller than the L2 cache data traffic, which indicates high data reuse.
- CARM and L2 traffics are almost the same, which indirectly indicates 100% of cache line usage because cache lines are fully filled with algorithmic data.



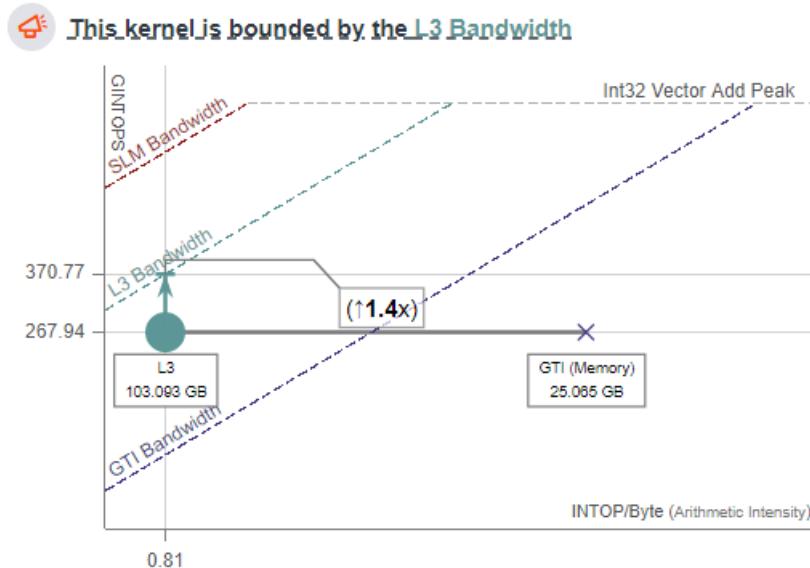
To confirm the 100% of cache line usage, review the L2 Cache Line Utilization metric in the **GPU** pane grid, which is 100%. The grid also reports VE Threading Occupancy of 98.1%, which indicates good scheduling and data distribution among threads.

GPU						
Compute Task	CARM (EU <> Data Port)			EU Array		EU Threading Occupancy
	Total, GB	L3 Cache Line Utilization	Active	Stalled	Idle	
[Outside any task]	0.000	0.0%	0.2%	0.3%	99.6%	0.4%
zeCommandListAppendMemoryCopyReg	0.000	0.0%	21.5%	34.3%	44.2%	39.6%
zeCommandListAppendBarrier	0.000	0.0%	0.0%	0.0%	100.0%	0.0%
matrixMultiply2<int, (unsigned long)2048>	103.096	100.0%	47.8%	52.2%	0.0%	98.3%

To understand the limitations for future kernel code optimization, review the following data reported by the Intel Advisor:

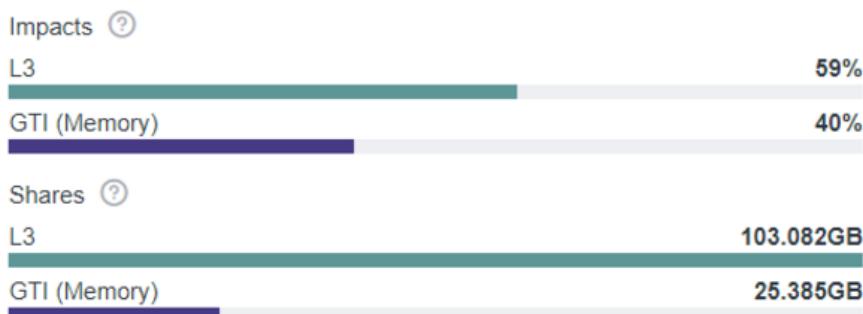
- The Roofline Guidance pane shows kernel limitation summary and provides estimation for possible performance speedup after optimization. For the matrix multiply kernel, the main limitation is the L2 cache bandwidth. The kernel can run 1.4x faster if it uses the L2 cache more effectively and reaches its maximum bandwidth with the same arithmetic intensity, but a better data access pattern.

## ROOFLINE GUIDANCE



- The Memory Metrics pane can help you understand memory level impact, which is time spent in requests from different memory levels, in per cent to the total time. For this kernel, GTI has less impact than L2 cache, but it is still taking a big part of total kernel execution time and may become a bottleneck after the L2 bandwidth limits are eliminated, for example, using SLM. Shares metric is a visual way of estimating data portions processed from different memory levels. In this case, L2 cache has 4x more data than GTI.

## MEMORY METRICS



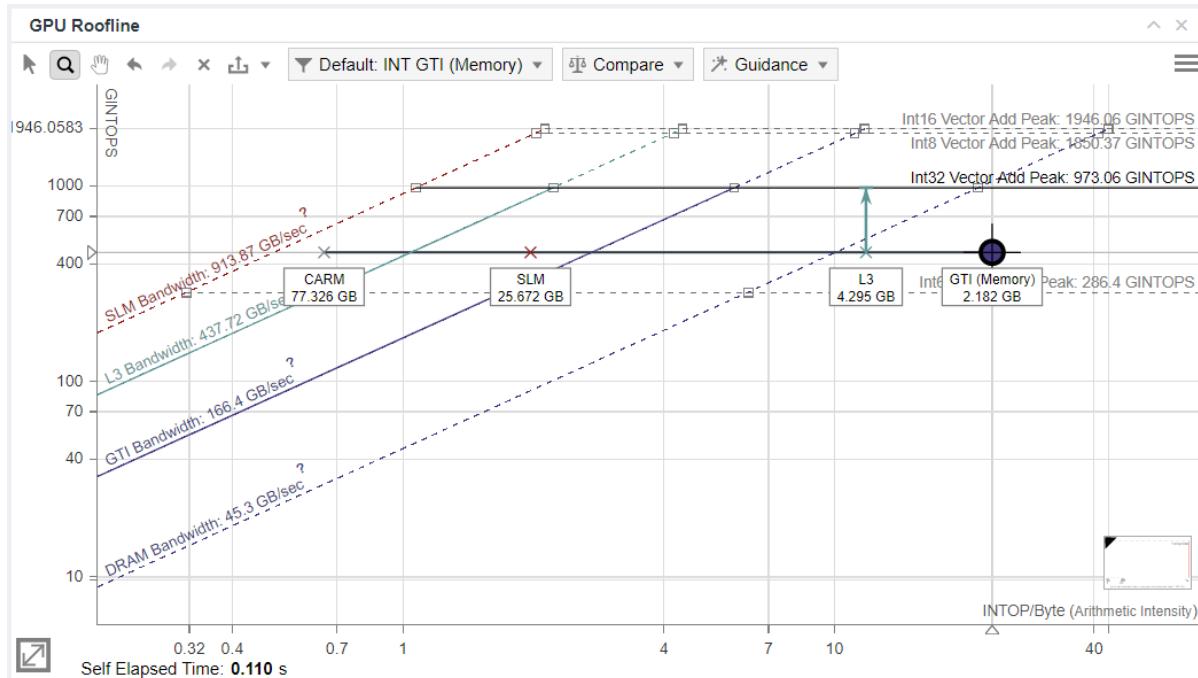
- The OP/S and Bandwidth pane shows the number of measured operations per second and data traffic in relation to the bandwidth limitations. For this kernel, the summary reports the following data:
  - The theoretical SLM bandwidth is almost 3x times higher than the L2 cache bandwidth, but the SLM is not used in this implementation. Blocking matrix arrays to use them as local shared data can eliminate the L2 cache bandwidth limits.
  - The kernel performance is only 27% of theoretically possible peak performance for Int32 data. With better memory access implementation, we could potentially reach 3x performance increase for this kernel.

OP/S AND BANDWIDTH			
► GINTOPS:	(?)	266.94	27% of 973.06 Int32 GINTOPS
► GFLOPS:	(?)	0	0% of 973.05 SP GFLOPS
► CARM Bandwidth:	(?)	330.89	GB/sec
► SLM Bandwidth:	(?)	0	0% of 913.87 GB/sec
► L3 Bandwidth:	(?)	330.85	71% of 460.06 GB/sec
► GTI (Memory) Bandwidth:	(?)	81.47	48% of 166.40 GB/sec

### Data Block Optimized Matrix Multiply Algorithm with SLM

Following the recommendations from the previous Intel Advisor result, we split the matrix arrays into small blocks to implement matrix multiplication data blocking and put the data blocks to the SLM for faster data reuse on a X<sup>e</sup>-core level.

For this optimized implementation with data blocking, the Roofline chart looks as follows:



The data distribution has changed from the previous result. Firstly, the execution is not limited to memory, but is compute bound, which is good for overall performance and further optimizations.

There are a couple things to note in the memory-level dots:

- SLM traffic is much bigger than L2 traffic. L2 traffic is not zero, which is expected as data blocks are read to L2 cache and then copied to SLM for reuse.
- CARM data traffic is three times bigger than the SLM traffic. The reason is not clear from the result, but it is a known effect that happens due to VE data port buffering data brought from SLM and accessed sequentially. This effect is positive and implies data reuse on the memory level closest to VEs.

Let us review data in the GPU Detail pane to understand changes in performance for this algorithm implementation:

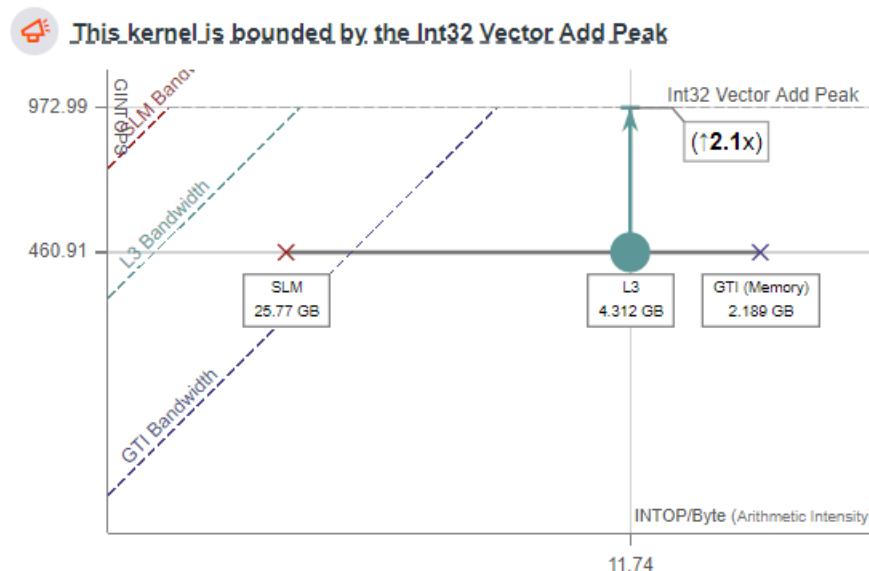
- As the OP/S and Bandwidth pane shows, the L2 and SLM bandwidth are far from their limits. The kernel performance has increased to 47% of its theoretical limit of integer operations per second (INTOPS).

### OP/S AND BANDWIDTH

▶ GINTOPS: ⓘ	460.91	47% of 972.99 Int32 GINTOPS
▶ GFLOPS: ⓘ	0	0% of 973.13 SP GFLOPS
▶ CARM Bandwidth: ⓘ	704.20	GB/sec
▶ SLM Bandwidth: ⓘ	234.68	25% of 913.87 GB/sec
▶ L3 Bandwidth: ⓘ	39.27	8% of 459.33 GB/sec
▶ GTI (Memory) Bandwidth: ⓘ	19.93	11% of 166.40 GB/sec

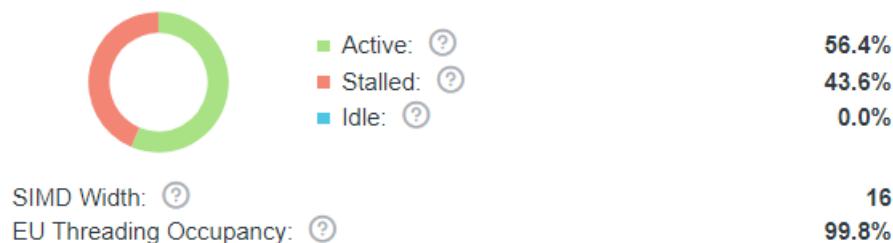
- As the Roofline Guidance chart shows, the kernel performance is limited by the Int32 Vector operations, which are the operations that the compiler used to implement the code. The chart indicates that the kernel can be optimized to run 2.1x faster.

### ROOFLINE GUIDANCE



- As the Performance Characteristics pane shows, the VEs are stalled for 43.6% of execution cycles. As the algorithm is fully vectorized, there should be other reasons for the VE stalls. By optimizing the VE performance, you might get the 2.1x performance improvement indicated in the Roofline Guidance pane.

### PERFORMANCE CHARACTERISTICS

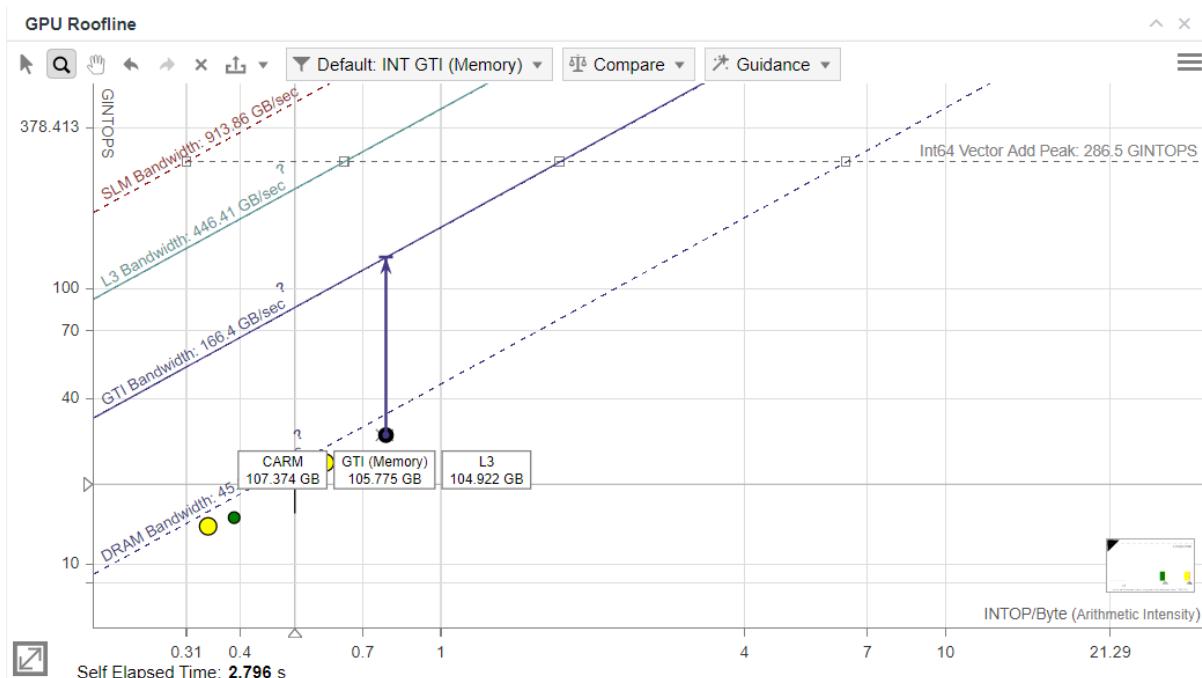


You can run the [GPU Compute/Media Hotspots analysis](#) of the Intel® VTune™ GPU Hotspot analysis to investigate reasons for the VE stalls further.

### Big Data Array with Data Reuse for a STREAM Benchmark

The STREAM benchmark is a small application that brings a big chunk of data from memory and executes basic compute kernels: Copy, Scalar, Add, and Triad. The number of compute operations per kernel is small or equals to 0, so the kernels are expected to be memory bound. For this reason, we use it to define data bandwidth limits in a system.

After analyzing the benchmark with the GPU Roofline Insights on the Intel Processor Graphics code-named Tiger Lake, the Roofline chart shows four dots that correspond to the benchmark kernels. The dots are located on the memory-bound side of the chart below the DRAM bandwidth roof.



The Roofline Guidance chart shows that the kernels are GTI Bandwidth bound, not DRAM bound as the main Roofline chart suggests. The reason for it is that Intel Advisor cannot measure the bandwidth for data transferred between DRAM and XVE on integrated GPUs due to hardware limitations.

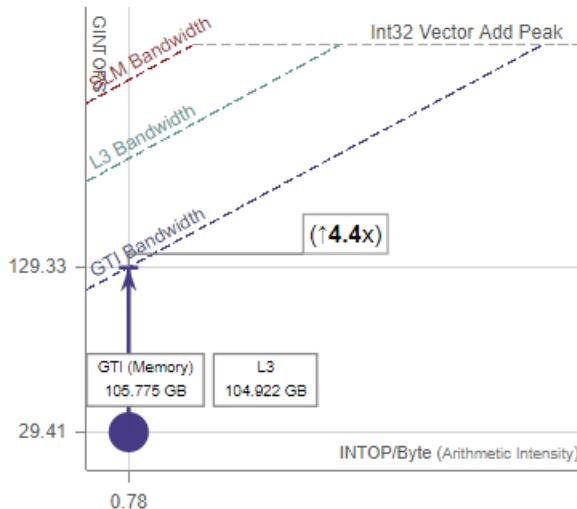
The Roofline Guidance suggests you improving cache locality to optimize performance and get better data reuse. This advice is also applicable to other cases when we test data bandwidth and compute performance is not a purpose for optimization.

## ROOFLINE GUIDANCE



### This kernel is bounded by the GTI Bandwidth

Improve cache locality. For example, optimize cache accesses by implementing cache blocking technique.



In the OP/S and Bandwidth pane, review the specific numbers for the achieved memory bandwidth. Notice that CARM, L2, and GTI stages has similar achieved bandwidth, so the bottleneck for this benchmark is the most distant memory interface.

## OP/S AND BANDWIDTH

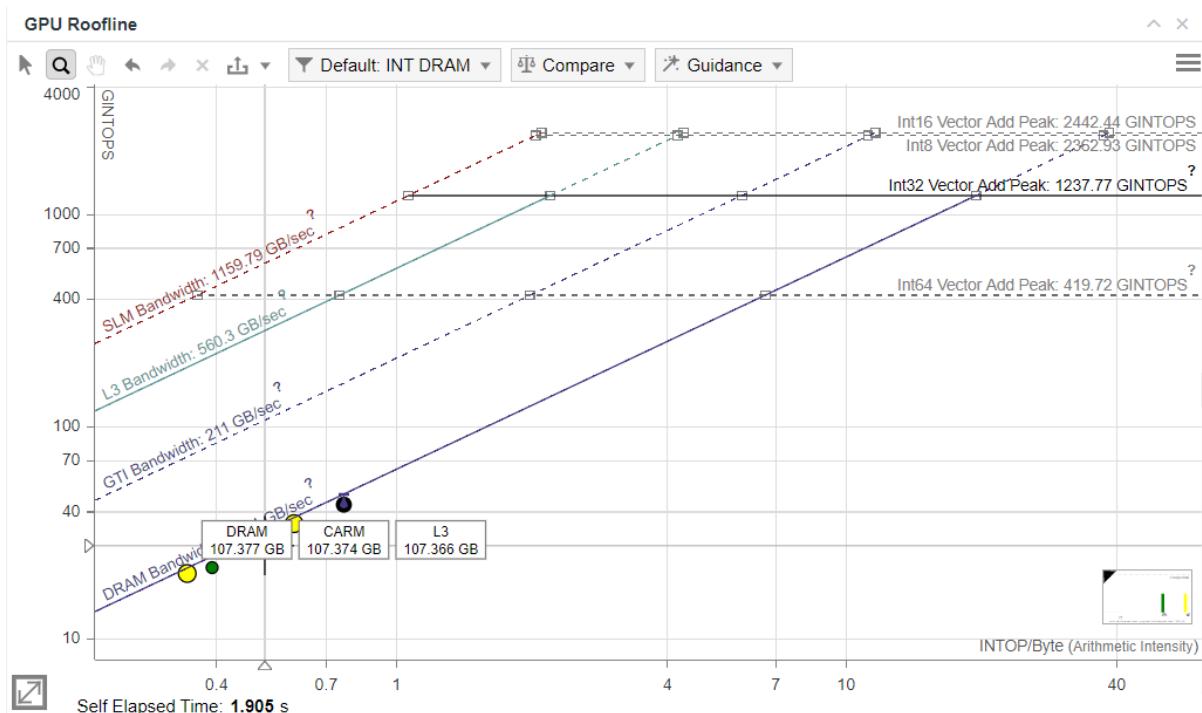
► GINTOPS: <a href="#">?</a>	23,43	2% of 973,06 Int32 GINTOPS
► GFLOPS: <a href="#">?</a>	0	0% of 973,08 SP GFLOPS
► CARM Bandwidth: <a href="#">?</a>	39,45	GB/sec
► SLM Bandwidth: <a href="#">?</a>	0	0% of 913,86 GB/sec
► L3 Bandwidth: <a href="#">?</a>	39,38	8% of 446,41 GB/sec
► GTI (Memory) Bandwidth: <a href="#">?</a>	39,63	23% of 166,40 GB/sec

Note here that all stages CARM, L2, and GTI have the similar effective BW and all 3 memory components are roughly identical to each other for a given kernel. Having identical roofline components means that there is no reuse in the cache or register file and every attempt to fetch the data requires accessing all the way down to external memory, because no data is cached any time. This (equal CARM, L2 and External Memory roofline components) is a common indication of “streaming” pattern.

In given case, this also indicates that the most distant memory interface is the bottleneck for this benchmark. Slight difference in kernels BW which is still can be observed is due to Copy/Scale kernels have equal Reads/Writes, while Add/Triad kernels have twice more Reads than Writes, and Read BW is higher on the system.

GPU		GPU Memory					
Compute Task		Total, GB	Read, GB	Write, GB	Bandwidth, GB/sec	Read, GB/sec	Write, GB/sec
[Outside any task]		4.984	2.892	2.093	2.300	1.334	0.966
zeCommandListAppendMemoryC		6.463	3.247	3.217	27.502	13.814	13.687
_ZTSZ17tuned_STREAM_Copyv		107.904	54.376	53.528	37.885	19.091	18.793
_ZTSZ18tuned_STREAM_Scaleit		105.775	53.313	52.462	37.837	19.071	18.766
_ZTSZ16tuned_STREAM_AddvE		160.174	107.105	53.069	39.589	26.472	13.117
_ZTSZ18tuned_STREAM_TriadiE		161.783	108.173	53.610	39.630	26.498	13.132

To eliminate the hardware limitations of the Intel Processor Graphics code-named Tiger Lake that do not allow Intel Advisor to measure bandwidth between DRAM and VE, let us analyze the benchmark running on a discrete Intel® Iris® Xe MAX graphics. The resulting Roofline chart shows four kernel dots below the DRAM bandwidth roof.



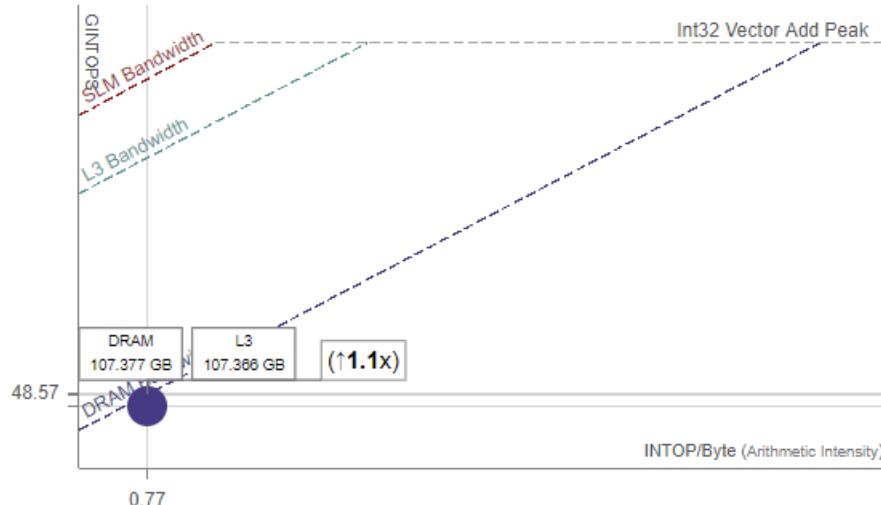
In the OP/S and Bandwidth pane, Intel Advisor now correctly identifies DRAM as the highest level of bottleneck.

## ROOFLINE GUIDANCE



This kernel is bounded by the DRAM Bandwidth

Improve cache locality. For example, optimize cache accesses by implementing cache blocking technique.



As the OP/S and Bandwidth and Memory Metrics panes show, the DRAM data traffic is very close to its theoretical limit, and the stream benchmark really measures the practical limits of the data flow.

## OP/S AND BANDWIDTH

▶ Compute (GINTOPS):	②	43,16	3% of 1237,77 GINTOPS	<b>Int32</b>
▶ Compute (GFLOPS):	②	0	0% of 1246,58 GFLOPS	<b>SP Vec</b>
▶ CARM Bandwidth:	②	56,38	GB/sec	
▶ SLM Bandwidth:	②	0	0% of 1159,79 GB/sec	
▶ L3 Bandwidth:	②	56,37	10% of 560,30 GB/sec	
▶ DRAM Bandwidth:	②	56,38	88% of 63,44 GB/sec	

## MEMORY METRICS

Impacts ②

L3

10%

DRAM

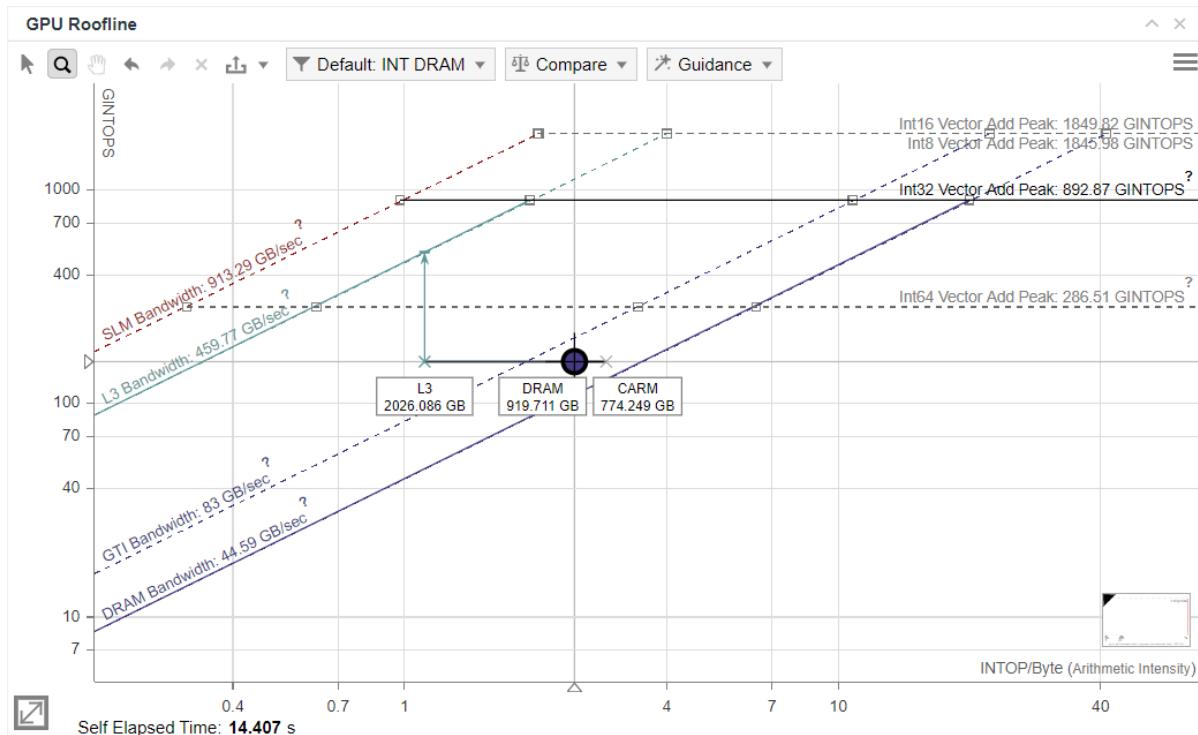
89%

### Partially Effective Data Access in a Stencil Data Pattern Benchmark

One of the most interesting cases is when data access is compact but in a very limited local range, while globally, the access is sparse. Such case is frequent in real-life applications, for example, in a stencil-based kernel computation where data in two axes, for example, X and Y, is accessed sequentially in the memory space, but data in Z axis is accessed in a big unit stride.

Let us analyze a 504.polbm applications from the SPEC ACCEL benchmark set running on the Intel Processor Graphics with Gen12 architecture. This benchmark application is written on C with OpenMP\* offload to GPU. It works with double-precision numbers, but the Intel Processor Graphics with Gen12 architecture can only simulate the calculations with integer data. That is why we examine the Roofline chart for integer operations.

The GPU Roofline chart shows one dot that correspond to the benchmark kernel. The dot is located between memory and compute roofs, which means that if GPU parameters are changed (for example, if you run the analysis for a hardware with a higher memory bandwidth), the kernel might slightly move from memory bound to compute bound.



As the Roofline Guidance pane shows, the kernel is limited by L2 cache bandwidth. Intel Advisor also detects low cache line utilization for the kernel, which is expected from a stencil-based kernel.

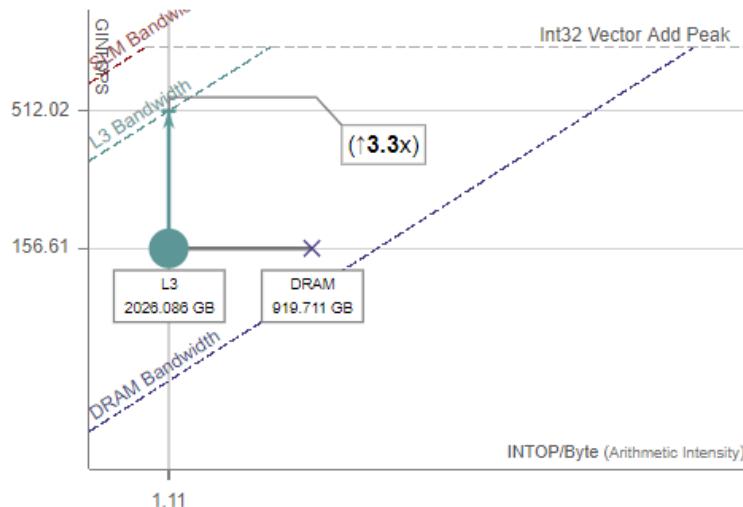
## ROOFLINE GUIDANCE



### This kernel is bounded by the L3 Bandwidth

Improve cache line utilization by optimizing memory access pattern.

Current cache line utilization is **38%**. It means that the number of bytes used by an execution unit is less than the number of bytes transferred to the L3 cache.



In general, to optimize data access in the stencil-based kernels, you need to apply different techniques that change data layout to use SLM for data locality and SIMD parallelism per data axis. However, you cannot change data layout for benchmarks, and all optimizations are done by the Graphics Compiler.

## Conclusion

GPU Roofline Insights perspective of the Intel® Advisor is a powerful tool that helps you investigate performance of kernel offloaded to Intel GPUs. It is easier to understand Roofline results for theoretical extreme cases. Such cases have only hardware limitations, so performance optimization strategy is clearer. However, real-life applications might have several limiting factors combined. The Roofline can help you address performance issues by identifying the most contributing factors. Once you eliminate one limitation, the analysis identifies the next factor you can address, until the performance is close to theoretical hardware limitations and optimization stops bringing improvements.

## Intel® Intercept Layer for OpenCL™ Applications

The [Intercept Layer for OpenCL Applications](#) is a command-line tool for tracing and analyzing OpenCL applications. It has a set of features for logging host and device activities, dumping and disassembling kernel binaries, etc. If your oneAPI applications use OpenCL backend, you may find this tool is helpful for performance tuning.

More information including the source code is available at the github repository [Intercept Layer for OpenCL Applications](#).

## Performance Tools in Intel® Profiling Tools Interfaces for GPU

[Intel® Profiling Tools Interfaces for GPU](#) has a set of tracing and profiling tools for Intel® oneAPI applications in the [tools](#) folder. These lightweight command-line tools support both Level Zero and OpenCL backends.

In addition to device activities, the tools are capable of tracing host activities from Level Zero or OpenCL runtime layer all the way up to MPI layer. You can use the tools to quickly identify bottlenecks on device and host. You can also use the tools to profile GPU hardware performance counters , for example, thread occupancies, memory traffic, cache utilization and function unit utilization etc.. Furthermore, You can run the tool [unitrace](#) to profile GPU kernels at instruction level, pinpoint instructions that stalls the execution unit or the vector engine and report reasons of the stalls.

The performance data generated in JSON format can be viewed in a browser using a trace viewer such as <https://ui.perfetto.dev>.

Please check the [tools](#) folder for detailed information and the source code of each tool.

## Configuring GPU Device

### Introduction

For extracting best end-to-end performance from applications, users may need to sometimes modify device settings at a system level. These knobs may need to be applied by a system administrator, depending on whether sudo access is required.

This chapter provides guidelines and instructions for changing certain device settings for performance on Linux\*.

### Install GPU Drivers or Plug-ins (Optional)

You can develop oneAPI applications using C++ and SYCL\* that will run on Intel, AMD\*, or NVIDIA\* GPUs.

To develop and run applications for specific GPUs you must first install the corresponding drivers or plug-ins:

- To use an Intel GPU, [install the latest Intel GPU drivers](#).
- To use an AMD\* GPU with the Intel® oneAPI DPC++ Compiler, [install the oneAPI for AMD GPUs plugin](#) from Codeplay.
- To use an NVIDIA\* GPU with the Intel® oneAPI DPC++ Compiler, [install the oneAPI for NVIDIA GPUs plugin](#) from Codeplay.

### Performance Impact of Pinning GPU Frequency

When applications use GPU for large portions of their computations, it's recommended to pin the GPU at an optimal frequency. Some examples of these applications are High Performance Computing workloads which have computationally intensive portions in their algorithms offloaded to the device.

For applications that are memory bound, with kernels running for a very short time, but spend more time in data exchanges, the effects of pinning GPU to a higher frequency might be less pronounced or even detrimental.

The maximum fused GPU frequency, which is the theoretical HW maximum frequency, can be obtained using sysfs handles as noted below:

```
for (( i=1; i<$num_cards; i++ ))
do
    for (( j=0; j<$num_tiles; j++ ))
    do
        cat /sys/class/drm/card$i/gt/gt$j/rps_RP0_freq_mhz;
    done
done
```

Users can also obtain the current maximum software frequency, that can be dynamically modified by user, via `rps_max_freq_mhz`:

```
for (( i=1; i<$num_cards; i++ ))
do
  for (( j=0; j<$num_tiles; j++ ))
  do
    cat /sys/class/drm/card$i/gt/gt$j/rps_max_freq_mhz;
  done
done
```

Default policy in Linux kernel mode driver (i915) for server platforms is to set the frequency request range such that: `rps_min_freq_mhz = rps_max_freq_mhz = rps_boost_freq_mhz = rps_RP0_freq_mhz`

So for many workloads, this policy might provide best performance out of the box.

Users can set the frequency using below sysfs handles provided by the DRM Linux kernel driver.

```
for (( i=1; i<$num_cards; i++ ))
do
  for (( j=0; j<$num_tiles; j++ ))
  do
    echo $desired_freq > /sys/class/drm/card$i/gt/gt$j/rps_min_freq_mhz;
    echo $desired_freq > /sys/class/drm/card$i/gt/gt$j/rps_max_freq_mhz;
    echo $desired_freq > /sys/class/drm/card$i/gt/gt$j/rps_boost_freq_mhz;
  done
done
```

Notes regarding frequency pinning:

- Firmware is final arbiter on granted frequency.
- Some throttling may occur for thermal/power budget reasons.
- Once frequency is pinned to a fixed value, there will be no dynamic scaling. \* For server platforms current i915 policy pins frequency to `rps_max_freq_mhz` at boot time.

## Switching Intel® Xe-link On/Off

Intel® Xe-link is a high-speed connectivity fabric hardware that provides accelerated data transfer capabilities for scale-up and scale-out operations. These are typically used for inter-GPU and inter-node data transfer operations for HPC applications deployed at cluster scale. However, for applications that don't use these capabilities, it could be beneficial to turn off the power to this resource selectively to allow for lower frequency throttling on the GPU compute engines.

Following command lines describe how to turn off power to Intel® Xe-link:

```
modprobe -r iaf;
for i in {0..$num_cards}; do
  cat /sys/class/drm/card$i/iaf_power_enable;
done;

for i in {0..$num_cards}; do
  echo 0 > /sys/class/drm/card$i/iaf_power_enable;
done;

for i in {0..$num_cards}; do
  cat /sys/class/drm/card$i/iaf_power_enable;
done
```

`$num_cards` can be obtained by listing `/sys/class/drm/` directory and noting how many `card*` sub-directories exist.

# Media Graphics Computing on GPU

We have focused on GPGPU optimization techniques and tools so far. When it comes to media graphics computing, a new set of optimization techniques and tools are also needed for maximizing utilization efficiency of the hardware, especially the dedicated units, for example, the media engines.

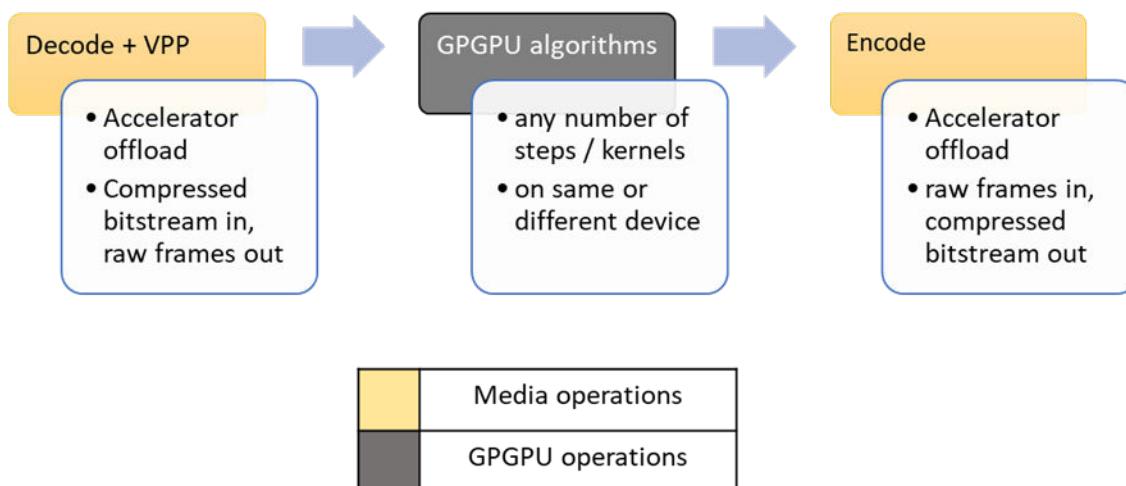
- Optimizing Media Pipelines
- Performance Analysis with Intel® Graphics Performance Analyzers

## Optimizing Media Pipelines

Media operations are ideal candidates for hardware acceleration because they are relatively large algorithms with well-defined inputs and outputs. Video processing hardware capabilities can be accessed via industry-standard frameworks, Intel Video Processing Library (Intel VPL), or low-level/operating system specific approaches like Video Acceleration API (VA-API) for Linux or Microsoft® DirectX® for Windows. Which path to choose depends on many factors. However, the basic principles like parallelization by multiple streams and maximizing data locality apply for all options.

The main differences between video processing and GPGPU work apply to all accelerator API options. Many typical GPGPU optimizations focus on optimizing how large grids of work are partitioned across multiple processing units. Hardware-accelerated media operations are implemented in silicon. They work in units of frames and usually work is partitioned by streams of frames.

Media optimization steps don't match the GPGPU workflow described in other sections. However, they can be easily added before or after GPGPU work. Media steps will supply inputs to or take outputs from GPGPU steps. For example:

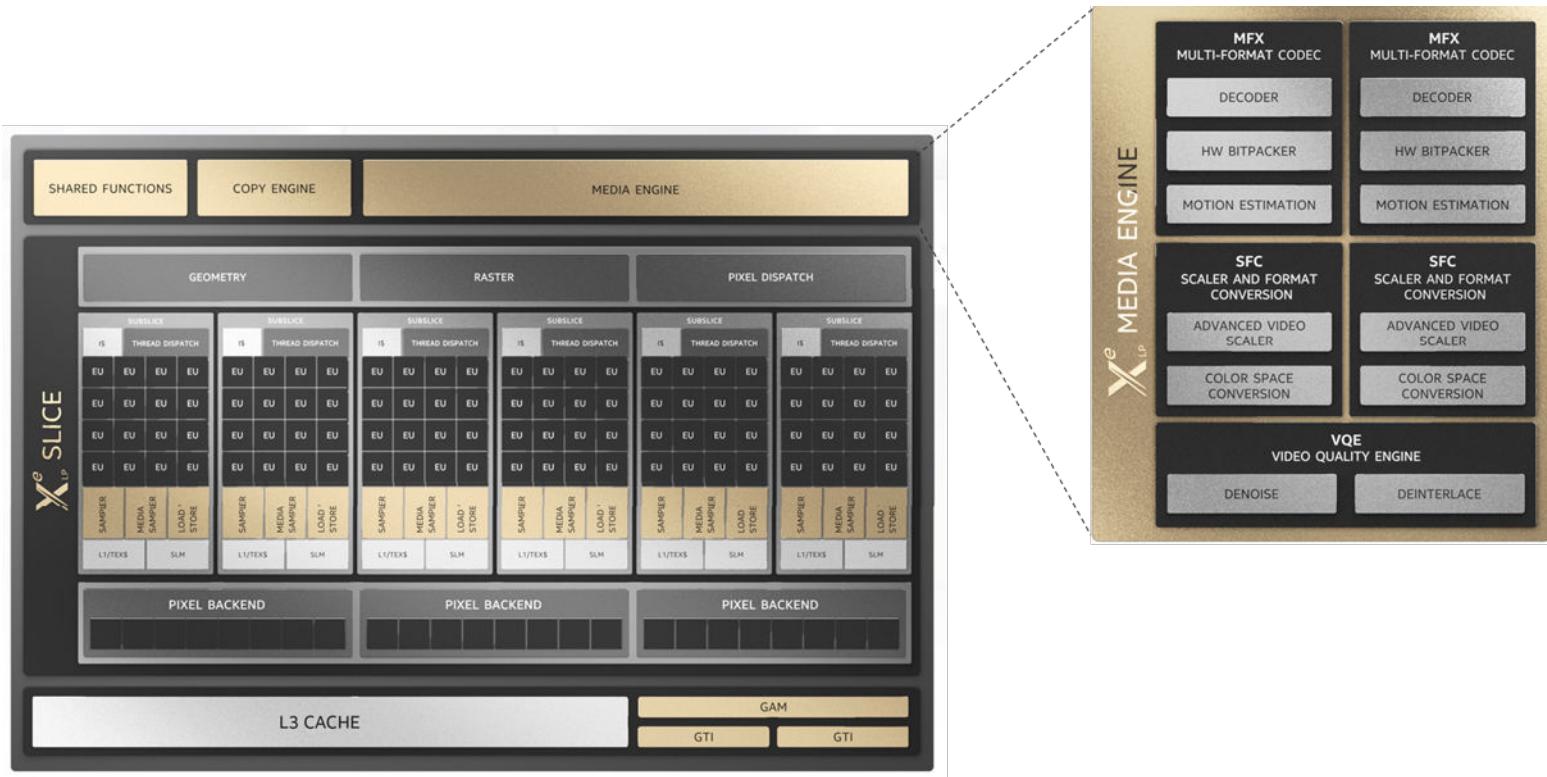


- Media Engine Hardware
- Media API Options for Hardware Acceleration
- Media Pipeline Parallelism
- Media Pipeline Inter-operation and Memory Sharing
- SYCL-Blur Example

Video streaming is prevalent in our world today. We stream meetings at work. We watch movies at home. We expect good quality. Taking advantage of this new media engine hardware gives you the option to stream faster, stream at higher quality and/or stream at lower power. This hardware solution is an important consideration for End-to-End performance in pipelines working with video data.

## Media Engine Hardware

As described in Intel® Xe Architecture section, [Xe- Intel® Data Center GPU Flex Series](#) and some other Intel® GPUs contain media engine which provide fully-accelerated video decode, encode and processing capabilities. This is sometimes called Intel® Quick Sync Video. The media engine runs completely independent of compute engines (vector and matrix engines).



Several components can be used by applications:

- MFX/Multi-format codec: hardware decode and encode. Some configurations include two forms of encode. 1) motion estimation + bit packing and 2) full fixed function/low power
- SFC/scaler and format conversion: resize (primarily intended for downscaling), conversion between color formats such as NV12 and BGRA
- Video Quality Engine: multiple frame processing operations, such as denoise and deinterlace.

This hardware has its own instruction queue and clock, so fully fixed function work can be very low power if configured to use low power pathways. This can also leave the slice capabilities on the GPU free for other work.

## Supported codecs

New codec capabilities are added with each new GPU hardware generation.

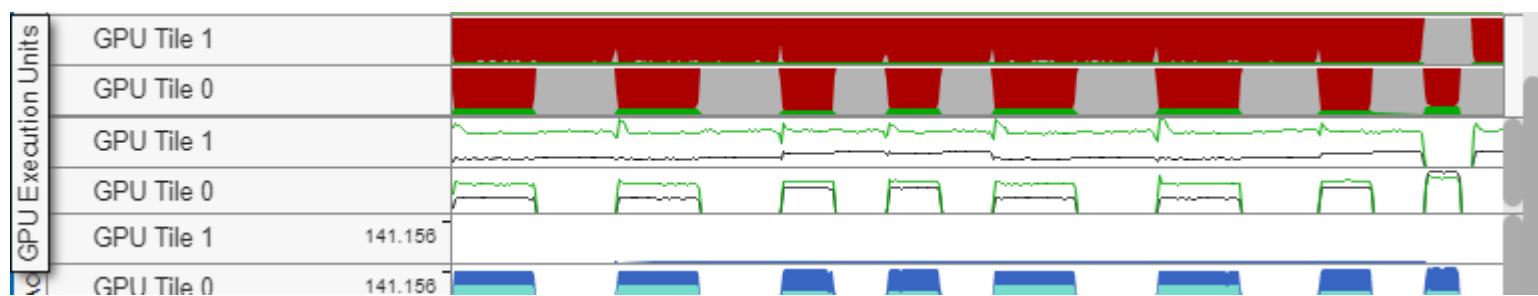
	AVC	MPEG 2	JPEG	VP8	HEVC 8-bit	HEVC 8-bit 422	HEVC 8-bit 444	HEVC 10-bit	HEVC 10-bit 422	HEVC 10-bit 444	HEVC 12-bit	HEVC 12-bit 422, 444	VP98-bit	VP98-bit 444	VP10-bit
	D/E*	D	D/E		D/E			D/E							
Intel® Core (BDW)	D/Es	D/Es	D	D											
Intel® Core (SKL)	D/E/Es	D/Es	D/E	D	D/Es										
Processor E3900 series (APL)	D/E/Es	D	D/E	D	D/Es			D					D		
Intel® Core (KBLx)	D/E/Es	D/Es	D/E	D/Es	D/Es			D/Es					D		D
Intel® Core (ICL)	D/E/Es	D/Es	D/E	D/Es	D/E/Es	D/Es	D/E	D/E/Es	D/Es	D/E			D/E	D/E	D
Processor X Series (EHL)	D/E	D	D/E	D	D/E	D	D/E	D/E	D	D/E			D/E	D/E	D
Intel® RKL/ADL, (DG1)	D/E/Es	D/Es	D/E	D (TGL only)	D/E/Es	D/Es	D/E	D/E/Es	D/Es	D/E	D/Es	D	D/E	D/E	D
	D/E	D/E	D/E	D	D/E	D/E	D/E	D/E		D/E	D/E	D	D/E	D/E	D

Note: in this table two kinds of encode are represented.

E=Hardware Encode via low power VDEnc

Es=Hardware Encode via (PAK) + Shader (media kernel +VME)

Intel® Arc A-series and Intel® Server GPU (previously known as Arctic Sound-M) add AV1 encode. This cutting edge successor to VP9 adds additional encode control for stack, segmentation, film grain filtering, and other new features. These increase encode quality at a given bitrate or allow a decrease in bitrate to provide increased quality.



## Media API Options for Hardware Acceleration

There are multiple ways to accelerate video processing on Intel® architecture (CPUs, GPUs). To choose the option that benefits you most, ensure your goals align with the tools you choose.

## Industry Standard Frameworks: FFmpeg, GStreamer, OpenCV, etc.

Used for

Full media solution w/ network protocols, container support, audio support, etc.

Easily move across accelerators and HW vendors

## Intel® oneAPI Video Processing Library (oneVPL)

Used for

Project focused on video elementary stream processing only

OS agnostic

## Low-level Hardware or OS-specific Solutions: VA-API/DXVA

Used for

Most control/direct integration with OS-specific graphics stack

Project is already based on VA-API/DXVA

Intel-optimized frameworks (ffmpeg/gstreamer)

Intel® oneAPI Video Processing Library (oneVPL)

Media SDK GPU

oneVPL GPU

libva/directx

Intel GPU Hardware

As shown above there are higher-level tools and lower-level tools. Do you need the extremely low-level control you can get with operating system specific tools like libva\* or DirectX\*? And do you have the extra time it takes to develop these low-level applications? Or is it more important to be able to easily port your code from Linux\* to Windows\* and save time by coding with higher level tools?

More details to help match the approach option to requirements are in the table below.

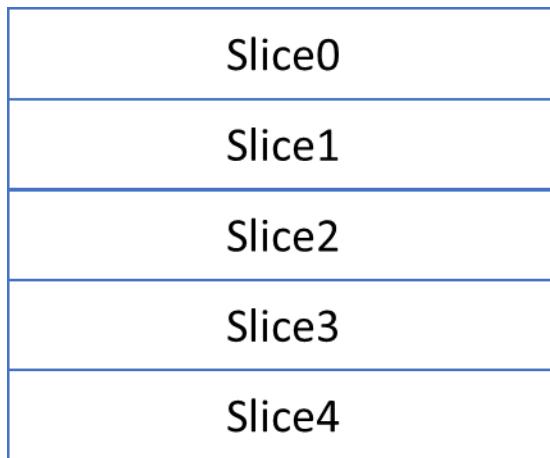
Intel Video Processing Library	Media Frameworks (FFmpeg & GStreamer)	Low-level/OS-specific solutions (Libva & DXVA)
Functionality	Elementary video stream processing with a limited set of frame processing operations	Full stack (network protocols, container support, audio support)
Level of control over hardware capabilities	Medium	Low
Portability	High	High

## Media Pipeline Parallelism

For GPGPU, parallelism focuses on concerns like how the ND range is partitioned and related edge conditions. Multiple accelerators can work on this partitioned space, executing the same algorithm over the entire grid (SIMD). This is not the case for encode/decode.

Instead of analyzing the internal implementation details of an encoder or decoder to find opportunities for parallelism as it processes each frame, in most cases the entire operation would be treated as a black box. Decode implementations for a codec are intended to be interchangeable, like substituting one box for another. Encode replacement is more complex, since effects of a broader range of parameters must be considered. However, the strategy is usually the same - swap the entire optimization to one best suited to the hardware instead of attempting to optimize hotspots/inner loops.

In theory, operations could parallelize by slice within frames:

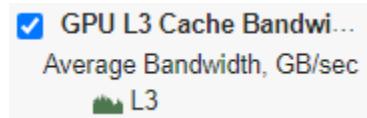


This is usually not practical. Since motion search cannot “see” across slice boundaries, overall compression quality is affected as the number of slices increase. Additional header bytes are required for slices as well.

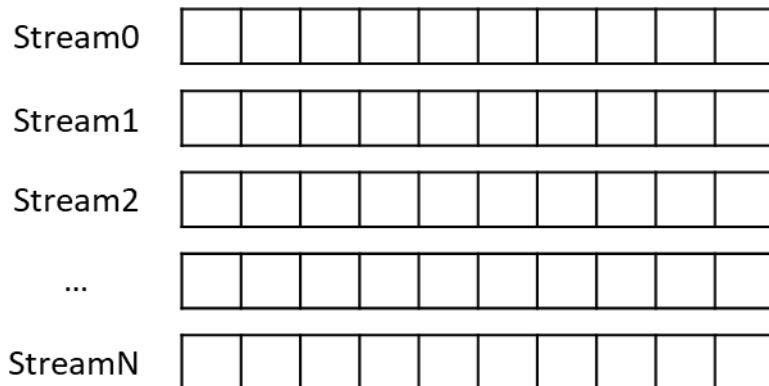
Single streams can be processed asynchronously, but this is also not scalable. Dependencies between frames prevent parallelism. Turning off these dependencies reduces quality at a given bitrate. Increasing the number of frames in flight also increases latency.



For single stream optimization, Deep Link hyper encode may simplify development. Deep Link hyper encode can provide a performance boost when one or more discrete GPUs are available on a system where integrated/processor graphics is also available by automatically coordinating work between integrated and discrete GPUs. Single stream performance can be improved by utilizing the capabilities of dGPU and iGPU together.



The best way to scale efficiently while preserving quality and reducing latency is to process multiple streams simultaneously. (Note: for non-realtime processing even a single stream can be processed in parallel as segments since frames will not have dependencies across segment/GOP boundaries.)



This approach provides ideal “embarrassing” parallelism which scales across accelerators. There are no dependencies across streams, so each accelerator can process as quickly as possible without coordination. For the Hyper Encode case, it is usually faster to schedule separate streams on iGPU and dGPU.

From a oneAPI perspective, these properties greatly simplify interoperability with SYCL. Media operations generally will not run “inside” kernels, which means there are fewer concerns at the API or development level. Media operations will either provide data for a kernel (act as a source), or they will work as a sink on data provided by a kernel. The main concern for performance is that the handoff between media operation and kernel implies synchronization and reduces opportunities to process asynchronously within a single stream. Processing multiple streams concurrently is the best workaround for this limitation.

## Optimizing Media Operations

Since the algorithms are implemented in hardware, the main concerns with media development are data locality, synchronization, and providing a pipeline of work to keep the hardware busy.

**Data locality:** keep frames on the GPU, avoiding copying to the CPU unnecessarily. Since the media engine is connected to the GPU memory hierarchy, data can be shared locally between slice and media engine components. From a GPGPU perspective these operations work on local GPU data. Frames can be shared between this hardware and execution units with low latency/zero copy. This is especially important for discrete GPUs, since moving raw frames across a PCI bus can be expensive.

**Synchronization:** Because the multiple hardware units can function independently, they can work asynchronously. For best performance, the application should force synchronization with CPU as infrequently as possible. Design algorithms so that the accelerator can proceed as far as it can without interrupt.

**Keeping the hardware busy:** If the instruction queue is not kept full, the engine clock will go down. It can take a few milliseconds to ramp up to full clock speed again.

## Media Pipeline Inter-operation and Memory Sharing

Media engine capabilities exposed in low-level OS-specific interfaces, such as:

- VA-API (Video Acceleration API) for Linux OS
- Microsoft DirectX® Video Acceleration for Windows OS

as well as various high-level media frameworks built on top of low-level interfaces, such as:

- Intel Video Processing Library (Intel VPL)
- FFmpeg and libav
- GStreamer

Each media framework defines own interfaces for device and context creation, memory allocation and task submission. Most frameworks also expose export/import interfaces to convert memory objects to/from other memory handles.

- High-level media frameworks (FFMpeg, GStreamer) support conversion to/from low-level media handles (VA-API and DirectX surfaces).

- Low-level media interfaces (VA-API, DirectX) support conversion to/ OS-specific general-purpose GPU memory handles such as DMA buffers on Linux and NT handles on Windows.
- Level-zero support conversion between DMA buffers / NT handles and USM device pointers.

Together these interfaces allow zero-copy memory sharing between media operations submitted via media frameworks and SYCL compute kernels submitted into SYCL queue, assuming the SYCL queue created on same GPU device as media framework and SYCL device uses Level-zero backend (not OpenCL backend).

Despite multiple stages of memory handles conversion (FFmpeg/GStreamer, VA-API/DirectX, DMA/NT, Level-Zero, SYCL), all converted memory handles refer to the same physical memory block. Thus writing data into one memory handle makes the data available in all other memory handles, assuming proper synchronization between write and read operations.

Below is reference to interfaces used for zero-copy buffer sharing between media frameworks and SYCL

1. (Linux) [VA-API to DMA-BUF](#)
2. (Windows) [DirectX to NT-Handle](#)
3. [DMA-BUF or NT-Handle to Level-zero](#)

The memory pointer created by Level-zero from DMA-BUF or NT-Handle (#3 above) is USM device pointer only accessible by SYCL kernels running on same GPU device as used for media memory allocation and media operations. This USM pointer is not accessible from host and not accessible from SYCL kernels running on CPU or other XPU devices.

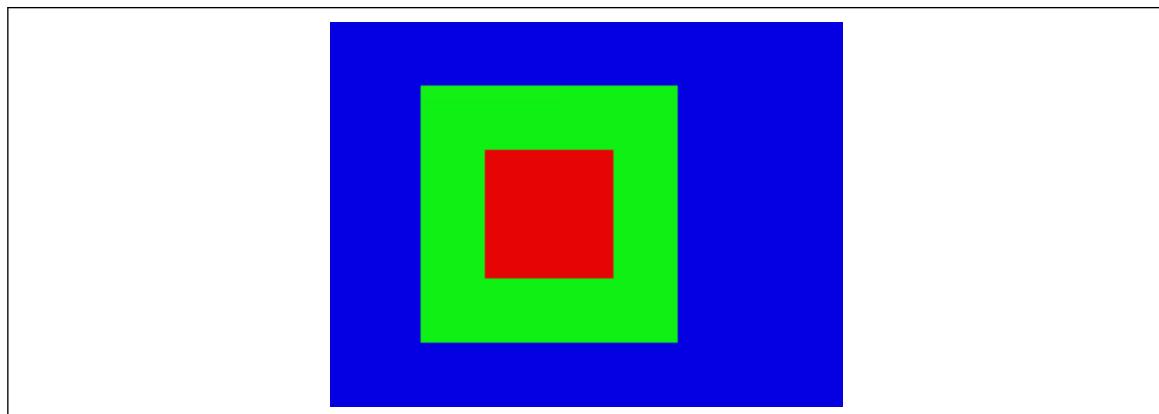
Example in next section demonstrates zero-copy buffer sharing between VA-API and SYCL using interfaces 1 and 3 from list above and synthetic video data (moving rectangle). For more advanced examples with FFmpeg/GStreamer video decode/encode on GPU media engine and SYCL kernels on GPU compute engines please refer to [Intel® DL Streamer memory interoperability API \(preview\)](#) and [Intel® DL Streamer samples](#)

## VA-API and SYCL Memory Sharing Example

The example

1. allocates shared VA-API surfaces and USM device pointers for NUM\_FRAMES frames
2. submits VA-API calls to draw moving rectangle on frames
3. submits SYCL kernels to draw sub-rectangle inside rectangle created by VA-API on step 2
4. synchronize all frames and write RGB data into file

Output frames generated by this example look like the picture below.



The example supports Linux OS and requires installation of the following additional packages besides oneAPI packages (installation example via **apt** package manager on Ubuntu OS).

```
sudo apt install intel-level-zero-gpu level-zero-dev
sudo apt install intel-media-va-driver-non-free libva-dev libva-drm2
```

and requires linkage with Level-zero and VA-API libraries

```
icpx -fsycl memory-sharing-with-media.cpp -lze_loader -lva -lva-drm
```

Example execution generates file `output.bgra` which could be directly played by some media players (ex, `ffplay`) or transcoded to compressed video format, for example using the following `ffmpeg` command:

```
ffmpeg -f rawvideo -pix_fmt bgra -s 320x240 -i output.bgra output.mp4
```

and then played by any media player, for example

```
ffplay output.mp4

// SYCL
#include <CL/sycl.hpp>

// SYCL oneAPI extension
#include <sycl/ext/oneapi/backend/level_zero.hpp>

// Level-zero
#include <level_zero/ze_api.h>

// VA-API
#include <va/va_drm.h>
#include <va/va_drmcommon.h>

#include <cstdio>
#include <fcntl.h>
#include <unistd.h>
#include <vector>

#define OUTPUT_FILE "output.bgra"

#define VAAPI_DEVICE "/dev/dri/renderD128"

#define FRAME_WIDTH 320
#define FRAME_HEIGHT 240

#define RECT_WIDTH 160
#define RECT_HEIGHT 160
#define RECT_Y (FRAME_HEIGHT - RECT_HEIGHT) / 2

#define NUM_FRAMES (FRAME_WIDTH - RECT_WIDTH)

#define VA_FORMAT VA_FOURCC_BGRA
#define RED 0xffff0000
#define GREEN 0xff00ff00
#define BLUE 0xff0000ff

#define CHECK_STS(_FUNC)
{
    auto _sts = _FUNC;
    if (_sts != 0) {
        printf("Error %d calling " #_FUNC, (int)_sts);
        return -1;
    }
}

VASurfaceID alloc_va_surface(VADisplay va_display, int width, int height) {
    VASurfaceID va_surface;
    VASurfaceAttrib surface_attrib{};
    surface_attrib.type = VASurfaceAttribPixelFormat;
    surface_attrib.flags = VA_SURFACE_ATTRIB_SETTABLE;
    surface_attrib.value.type = VAGenericValueTypeInteger;
```

```
surface_attrib.value.value.i = VA_FORMAT;
vaCreateSurfaces(va_display, VA_RT_FORMAT_RGB32, width, height, &va_surface,
                 1, &surface_attrib, 1);
return va_surface;
}

int main() {
    // Create SYCL queue on GPU device and Level-zero backend, and query
    // Level-zero context and device
    sycl::queue sycl_queue{sycl::ext::oneapi::filter_selector(
        "level_zero")}; // { sycl::gpu_selector() }
    auto ze_context = sycl::get_native<sycl::backend::ext_oneapi_level_zero>(
        sycl_queue.get_context());
    auto ze_device = sycl::get_native<sycl::backend::ext_oneapi_level_zero>(
        sycl_queue.get_device());

    // Create VA-API context (VADisplay)
    VADisplay va_display = vaGetDisplayDRM(open(VAAPI_DEVICE, O_RDWR));
    if (!va_display) {
        printf("Error creating VADisplay on device %s\n", VAAPI_DEVICE);
        return -1;
    }
    int major = 0, minor = 0;
    CHECK_STS(vaInitialize(va_display, &major, &minor));

    // Create VA-API surfaces
    VASurfaceID surfaces[NUM_FRAMES];
    for (int i = 0; i < NUM_FRAMES; i++) {
        surfaces[i] = alloc_va_surface(va_display, FRAME_WIDTH, FRAME_HEIGHT);
    }

    // Convert each VA-API surface into USM device pointer (zero-copy buffer
    // sharing between VA-API and Level-zero)
    void *device_ptr[NUM_FRAMES];
    size_t stride;
    for (int i = 0; i < NUM_FRAMES; i++) {
        // Export DMA-FD from VASurface
        VADRMPRIMESurfaceDescriptor prime_desc{};
        CHECK_STS(vaExportSurfaceHandle(va_display, surfaces[i],
                                         VA_SURFACE_ATTRIB_MEM_TYPE_DRM_PRIME_2,
                                         VA_EXPORT_SURFACE_READ_WRITE, &prime_desc));
        auto dma_fd = prime_desc.objects->fd;
        auto dma_size = prime_desc.objects->size;
        stride = prime_desc.layers[0].pitch[0] / sizeof(uint32_t);

        // Import DMA-FD into Level-zero device pointer
        ze_external_memory_import_fd_t import_fd = {
            ZE_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMPORT_FD,
            nullptr, // pNext
            ZE_EXTERNAL_MEMORY_TYPE_FLAG_DMA_BUF, dma_fd};
        ze_device_mem_alloc_desc_t alloc_desc = {};
        alloc_desc.stype = ZE_STRUCTURE_TYPE_DEVICE_MEM_ALLOC_DESC;
        alloc_desc.pNext = &import_fd;
        CHECK_STS(zeMemAllocDevice(ze_context, &alloc_desc, dma_size, 1, ze_device,
                                   &device_ptr[i]));

        // Close DMA-FD
        close(dma_fd);
    }
}
```

```
}

// Create VA-API surface with size 1x1 and write GREEN pixel
VASurfaceID surface1x1 = alloc_va_surface(va_display, 1, 1);
VAImage va_image;
void *data = nullptr;
CHECK_STS(vaDeriveImage(va_display, surface1x1, &va_image));
CHECK_STS(vaMapBuffer(va_display, va_image.buf, &data));
*(uint32_t *)data = GREEN;
CHECK_STS(vaUnmapBuffer(va_display, va_image.buf));
CHECK_STS(vaDestroyImage(va_display, va_image.image_id));

// VA-API call to fill background with BLUE color and upscale 1x1 surface into
// moving GREEN rectangle
VAConfigID va_config_id;
VAContextID va_context_id;
CHECK_STS(vaCreateConfig(va_display, VAProfileNone, VAEEntrypointVideoProc,
                        nullptr, 0, &va_config_id));
CHECK_STS(vaCreateContext(va_display, va_config_id, 0, 0, VA_PROGRESSIVE,
                        nullptr, 0, &va_context_id));
for (int i = 0; i < NUM_FRAMES; i++) {
    VAProcPipelineParameterBuffer param{};
    param.output_background_color = BLUE;
    param.surface = surface1x1;
    VARectangle output_region = {int16_t(i), RECT_Y, RECT_WIDTH, RECT_HEIGHT};
    param.output_region = &output_region;
    VABufferID param_buf;
    CHECK_STS(vaCreateBuffer(va_display, va_context_id,
                            VAProcPipelineParameterBufferType, sizeof(param),
                            1, &param, &param_buf));
    CHECK_STS(vaBeginPicture(va_display, va_context_id, surfaces[i]));
    CHECK_STS(vaRenderPicture(va_display, va_context_id, &param_buf, 1));
    CHECK_STS(vaEndPicture(va_display, va_context_id));
    CHECK_STS(vaDestroyBuffer(va_display, param_buf));
}

#endif
// Synchronization is optional on Linux OS as i915 KMD driver synchronizes
// write/read commands submitted from Intel media and compute drivers
for (int i = 0; i < NUM_FRAMES; i++) {
    CHECK_STS(vaSyncSurface(va_display, surfaces[i]));
}
#endif

// Submit SYCL kernels to write RED sub-rectangle inside GREEN rectangle
std::vector<sycl::event> sycl_events(NUM_FRAMES);
for (int i = 0; i < NUM_FRAMES; i++) {
    uint32_t *ptr = (uint32_t *)device_ptr[i] +
                    (RECT_Y + RECT_HEIGHT / 4) * stride + (i + RECT_WIDTH / 4);
    sycl_events[i] = sycl_queue.parallel_for(
        sycl::range<2>(RECT_HEIGHT / 2, RECT_WIDTH / 2), [=](sycl::id<2> idx) {
            auto y = idx.get(0);
            auto x = idx.get(1);
            ptr[y * stride + x] = RED;
        });
}

// Synchronize all SYCL kernels
```

```

sycl::event::wait(sycl_events);

// Map VA-API surface to system memory and write to file
FILE *file = fopen(OUTPUT_FILE, "wb");
if (!file) {
    printf("Error creating file %s\n", OUTPUT_FILE);
    return -1;
}
for (int i = 0; i < NUM_FRAMES; i++) {
    CHECK_STS(vaDeriveImage(va_display, surfaces[i], &va_image));
    CHECK_STS(vaMapBuffer(va_display, va_image.buf, &data));
    fwrite(data, 1, FRAME_HEIGHT * FRAME_WIDTH * 4, file);
    CHECK_STS(vaUnmapBuffer(va_display, va_image.buf));
    CHECK_STS(vaDestroyImage(va_display, va_image.image_id));
}
fclose(file);
printf("Created file %s\n", OUTPUT_FILE);

// Free device pointers and VA-API surfaces
for (int i = 0; i < NUM_FRAMES; i++)
    zeMemFree(ze_context, device_ptr[i]);
vaDestroySurfaces(va_display, surfaces, NUM_FRAMES);

return 0;

```

## SYCL-Blur Example

For a working Intel Video Processing Library (Intel VPL) example which ties several of these concepts together (currently only for a single stream), see `sycl-blur`. This sample shows memory interoperation between video APIs and Intel VPL as the frame is input, manipulated and output using the following steps.

- Set up SYCL
- Set up an Intel VPL session
- Initialize Intel VPL VPP
- Loop through frames
  - Read the frame from a file
  - Run VPP resize/colorspace conversion on the GPU
  - Get access to the GPU surface, convert to USM
  - Run SYCL kernel (blur) on the GPU
  - Output the frame to a file

Find this sample here:

<https://github.com/oneapi-src/oneAPI-samples/tree/master/Publications/GPU-Opt-Guide/memory-sharing-with-media>

In this example, you can see that the interaction between Intel VPL and SYCL is at a frame level. Intel VPL provides a frame then the SYCL kernel processes it. For the OS environment where zero-copy capabilities are enabled in L0 (Linux), the libva frame data is made available to SYCL as USM. Instead of copying the libva raw frame to a new USM surface, it is possible for the app to work with the frame on the GPU as a libva surface then start working with the same memory as if it were USM.

To keep this example simple there are many design simplifications which currently limit its ability to fully showcase the benefits of zero copy.

- Raw frames are read from disk and written to disk - this sets the overall frame rate
- VPP data is read in as system memory and converted to video memory
- The pipeline is synchronized at each frame

However, zero copy is the core concept which can be built into a high performance application.

## Performance Analysis with Intel® Graphics Performance Analyzers

### Introduction

Intel® Graphics Performance Analyzers (Intel® GPA)

Intel® GPA is a performance analysis tool suite for analysis of applications run on single and multi-CPU platforms as well as single and multi-GPU platforms. It offers detailed analysis of data both visually and via scripting.

Intel® GPA consists of 4 GUI tools and a command line tool.

- GUI Tools
  - Graphics Monitor - hub of Intel® GPA tools for selecting options and starting trace and frame captures.
  - System Analyzer - live analysis of CPU and GPU activity.
  - Graphics Trace Analyzer - deep analysis of CPU/GPU interactions during a capture of a few seconds of data.
  - Graphics Frame Analyzer - deep analysis of GPU activity in one or more frames.
- Command Line Interface Tool
  - Intel® GPA Framework - scriptable command line interface allowing the capture and analysis of extensive data from one or more frames.

This chapter focuses on the functionality of Graphics Frame Analyzer for the deep view it provides into GPU behavior.

(Note: Intel® GPA Framework can be used to capture the same data as it is the backend of Graphics Frame Analyzer. In addition Intel® GPA Framework can be used to automate profiling work.)

#### Graphics Frame Analyzer Features

Some of the useful features of Graphics Frame Analyzer are:

- Immediately see which frame of a set of frames takes longest.
- Use Advanced Profiling Mode to automatically calculate your application's hottest bottlenecks based both on pipeline state and GPU metrics so that optimizing for one frame may optimize multiple parts of your application.
- Geometry - wire frame and solid:
  - in seconds you can see if you have drawn outside of your screen view.
  - View the geometry at any angle, dynamically.
- Textures
  - Visualize textures draw call by draw call.
  - See if a draw call with a long duration is drawing an insignificant part of the frame.
- Shaders
  - See which shader code lines take the most time.
  - See how many times each DXR (DirectX Raytracing) shader is called, as well as shader Vector Engine occupancy.
- Render State Experiments - at the push of a button simplify textures or pixels or disable events to help locate the causes of bottlenecks, immediately seeing the changes in the metrics and other data.

#### Supported APIs

- Direct3D 11
- Direct3D 12 (including DirectX 12 Ultimate)
- Vulkan

## Execution Unit Stall, Active and Throughput

Graphics Frame Analyzer is a powerful tool that can be used by novice and expert alike to take a quick look at frame duration, API calls, resource data and event data. Understanding more about the meaning of each data element levels you up making it easier to root cause performance issues.

Execution Stall, Execution Active, Execution Throughput

Knowing how to interpret the interrelationships of these 3 data elements can take you much further in your ability to understand the interworking of your applications with respect to the GPU(s).

EU, XVE (X<sup>e</sup> Vector Engine), and XMX

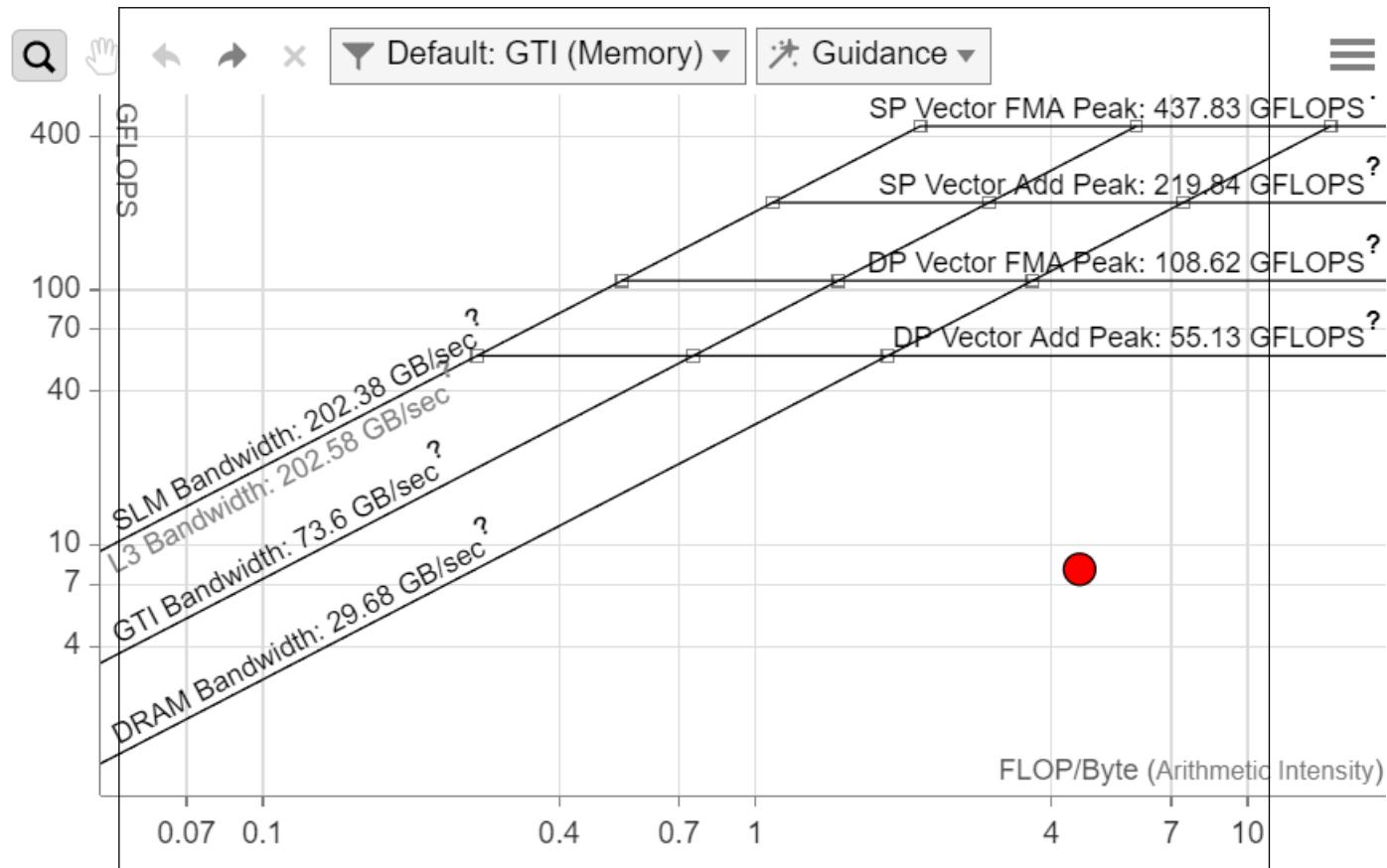
As discussed in the \* Intel® X<sup>e</sup> GPU Architecture \* section of this document, in X<sup>e</sup>-LP and prior generations of Intel® GPUs the EU - execution unit - is the compute unit of the GPU. In X<sup>e</sup>-HPG and X<sup>e</sup>-HPC we introduced the X<sup>e</sup>-core as the compute unit. For these latter platforms each X<sup>e</sup>-core consists of ALUs (arithmetic logic units) - 8 vector engines (XVE) and 8 matrix extensions (XMX).

In Graphics Frame Analyzer, if you are running on X<sup>e</sup>-LP or earlier architecture, you will see EU Active, EU Stall and EU Throughput data labels. On newer architecture you will see XVE Active, XVE Stall and XVE Throughput data labels. Here we use X<sup>e</sup>-LP as our reference architecture, thus we will refer to the EU. But understand that whether it is the EU or the XVE, the Stall/Active/Throughput relationships affect performance in the same ways.

Understanding these 3 data elements

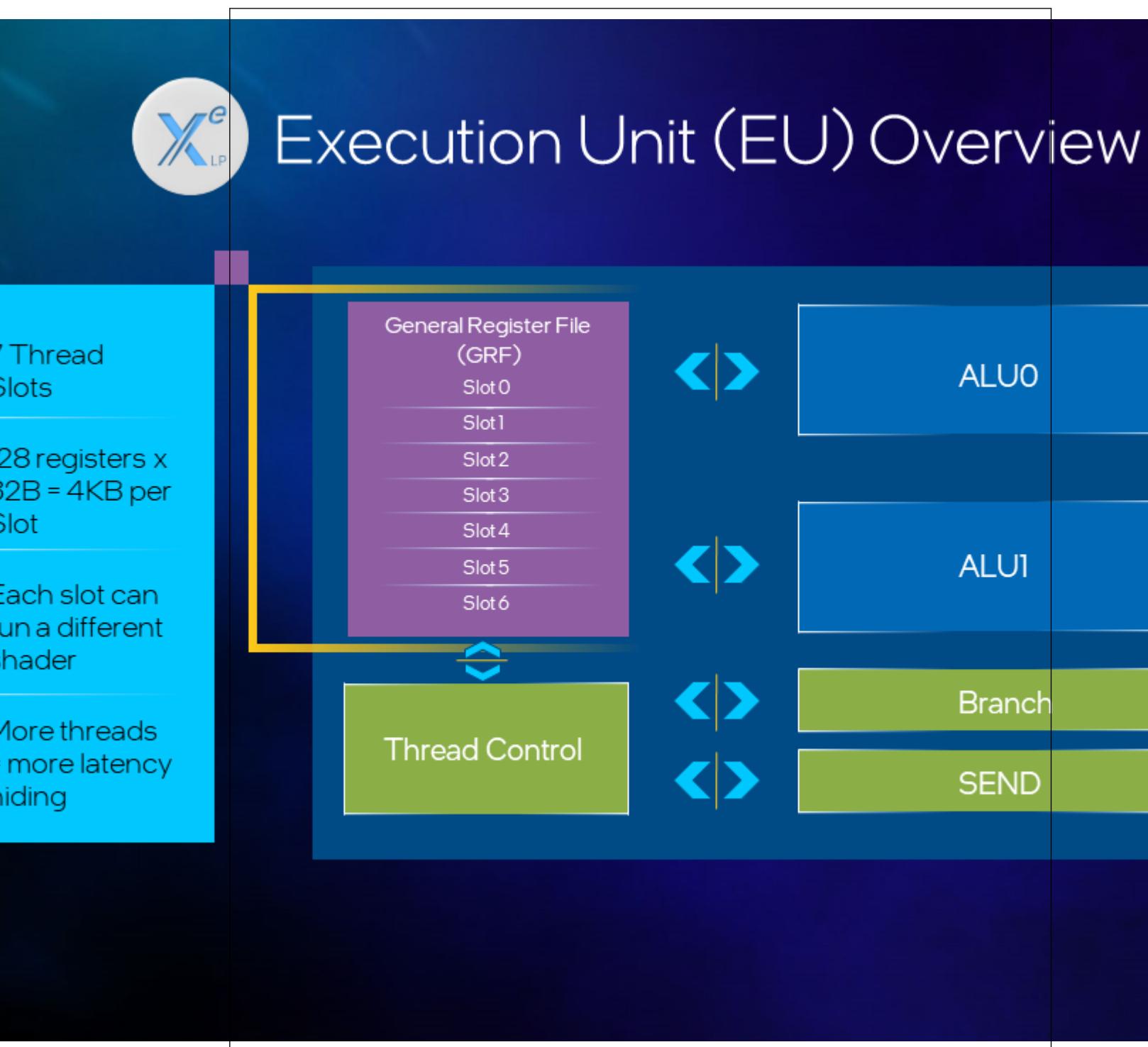
First, let's see what it looks like to drill down from the entire X<sup>e</sup>-LP unit with its 96 EUs into a single EU. The General Register File (GRF) on each EU of this particular GPU holds 7 threads. Figure 1 shows the zoom in from the entire GPU to a single EU.

### Zooming in on a single EU



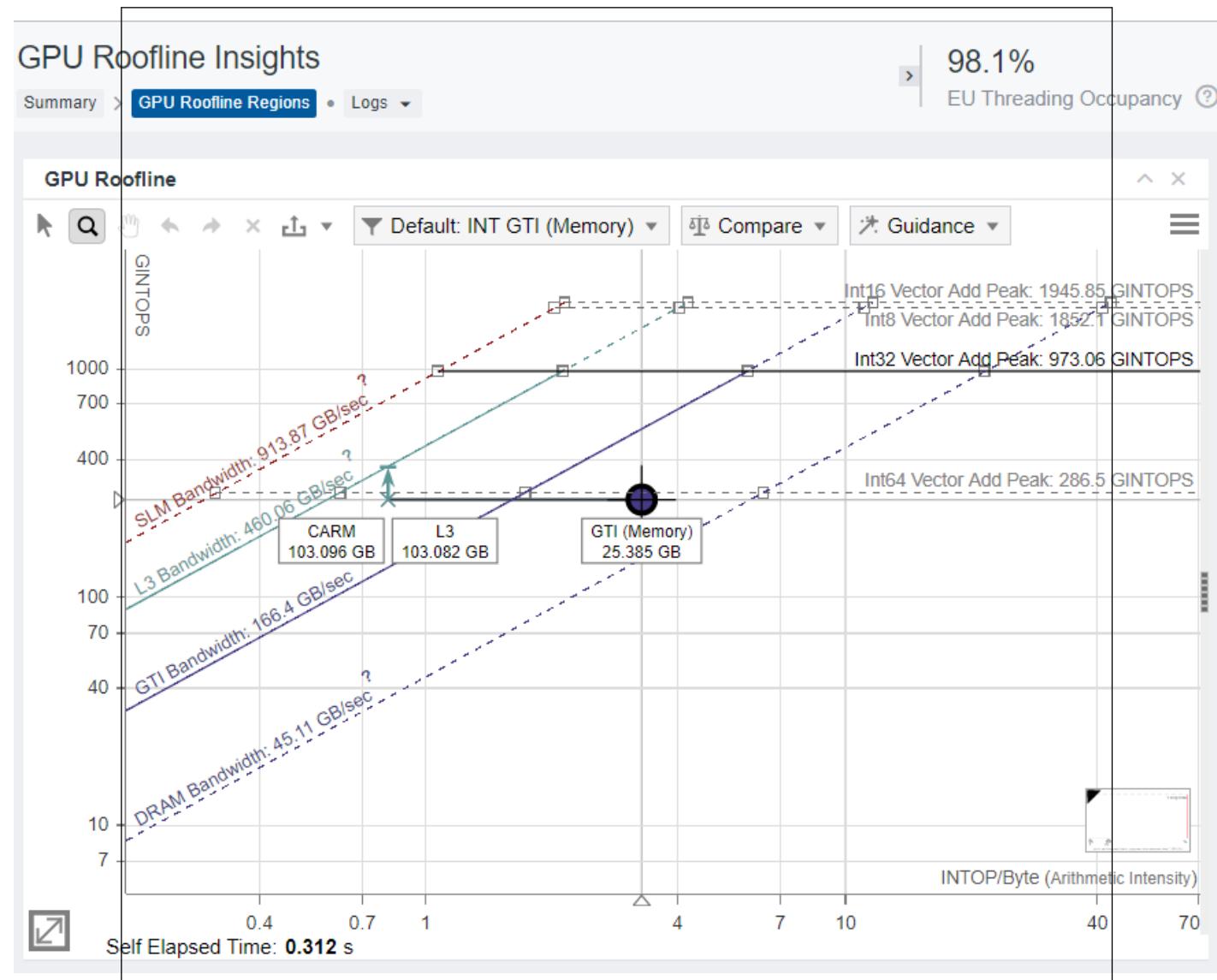
Let's take a closer look at the details of the EU. In Figure 2, of the elements shown, we will focus primarily on the 7 thread slots of the GRF, addressing the importance of the thread interactions with the SEND unit and the 2 ALUs.

**View of the GRF, the ALUs and the Thread Control, Branch and SEND units.**



Now let's look at a threading scenario. Figure 3 shows the contents of the GRF. We see that the GRF of this EU is loaded with only one thread in this instant. We see that single thread executing for some quantity of time. This means that one or both of the ALUs are invoked to execute instructions. At some point that thread needs to either read or write data, which activates the SEND unit. While the thread waits for the read/write to complete, the EU is stalled - it has a thread loaded but nothing to compute during that time.

### The GRF contains a single thread.



Augmenting this scenario, in Figure 4 there is a second thread in the EU. If there is a second thread loaded into the GRF of this EU, then, at the time when the first thread invokes the SEND unit, instead of stalling execution, the EU begins executing the instructions of the second thread. When that second thread invokes a

command requiring the SEND unit, the EU becomes stalled until the first thread is able to continue. Had there been a third thread in this EU or if the first SEND returned sooner, all latency may have been hidden, resulting in thread latency, but no stall during this time for this EU.

**The ALUs do nothing (no execution of instructions) while the EU waits for data to be read or written.**

GPU	Compute Task	CARM (EU <-> Data Port) »		EU Array «			EU Threading
		Total, GB	L3 Cache Line Utilization	Active	Stalled	Idle	
[Outside any task]		0.000	0.0%	0.2%	0.3%	99.6%	0.4%
zeCommandListAppendMemoryCopyReg		0.000	0.0%	21.5%	34.3%	44.2%	39.6%
zeCommandListAppendBarrier		0.000	0.0%	0.0%	0.0%	100.0%	0.0%
matrixMultiply2<int, (unsigned long)2048>		103.096	100.0%	47.8%	52.2%	0.0%	98.3%

### Terminology

For the following definitions the resulting data calculated

- is the average across all EUs;
- consists of both the full frame data and the data for the selected portion of the frame. The selection may be a single call or a set of calls, or even a set of frames.

### Idle

EU Idle is the percentage of time when no thread is loaded.

### Active

EU Active is the percentage of time when ALU0 or ALU1 were executing some instruction. It should be as high as possible; a low value is caused either by a lot of stalls or EUs being idle.

### Stall

EU Stall is the percentage of time, when one or more threads are loaded but none of them are able to execute because they are waiting for a read or write.

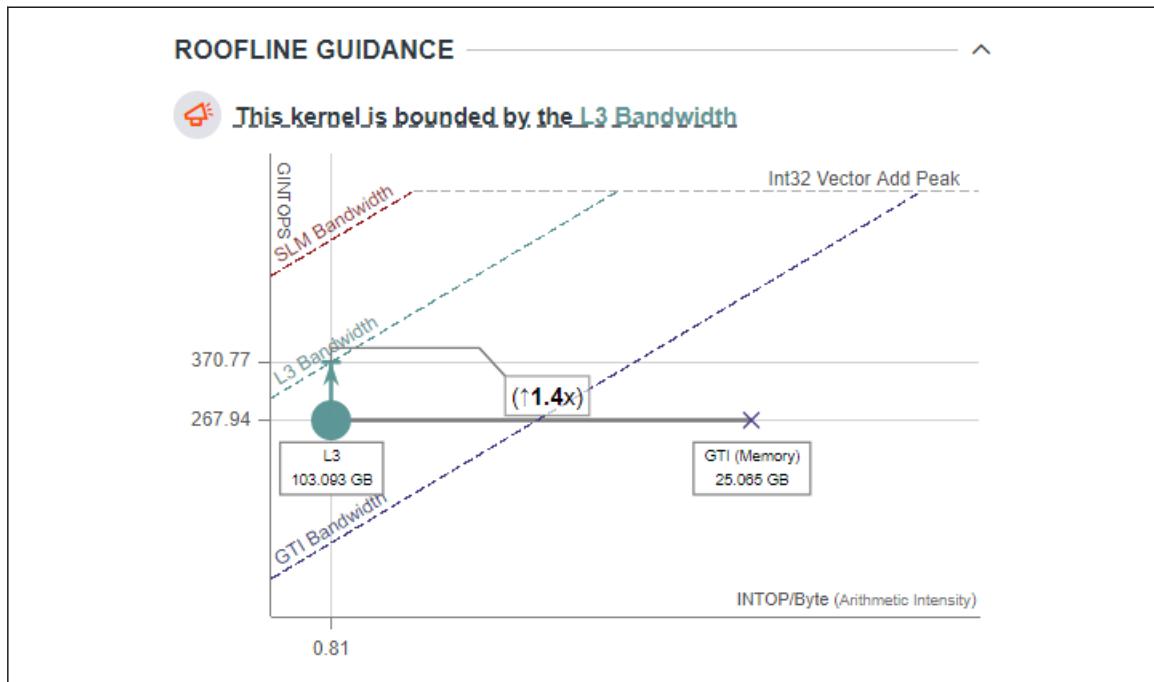
### Thread Occupancy

EU Thread Occupancy is the percentage of occupied GRF slots (threads loaded). This generally should be as high as possible. If the EU Thread Occupancy value is low, this indicates either a bottleneck in preceding HW blocks, such as vertex fetch or thread dispatch, or, for compute shaders it indicates a suboptimal SIMD-width or Shared Local Memory usage.

If only a single thread is executing on each of the 96 EUs, then 1 of 7 slots/EU is occupied, resulting in thread occupancy of 1/7 (14%).

If 2 EUs have no threads loaded (they are idle) but the other 94 EUs have 6 threads loaded, we have occupancy =  $(0 + 0 + 6*94)/672 = 84\%$ .

The Thread Occupancy values you will see in Graphics Frame Analyzer indicate the average occupancy across all EUs over the time selected. Though other hardware may have a different number of EUs or XVEs, the calculations are over all execution units. For example, below, on the left, you see a frame where over the entire frame duration of 6ms, though thread occupancy fluctuated during that 6ms, the average over that time for all 96 EUs is 77%. We can also look at thread occupancy over a part of the frame duration. Below, on the right, we select the 3 most time-consuming draw calls and see that during the 1.9ms that it took to execute these bits of the application, thread occupancy is 85.3%.

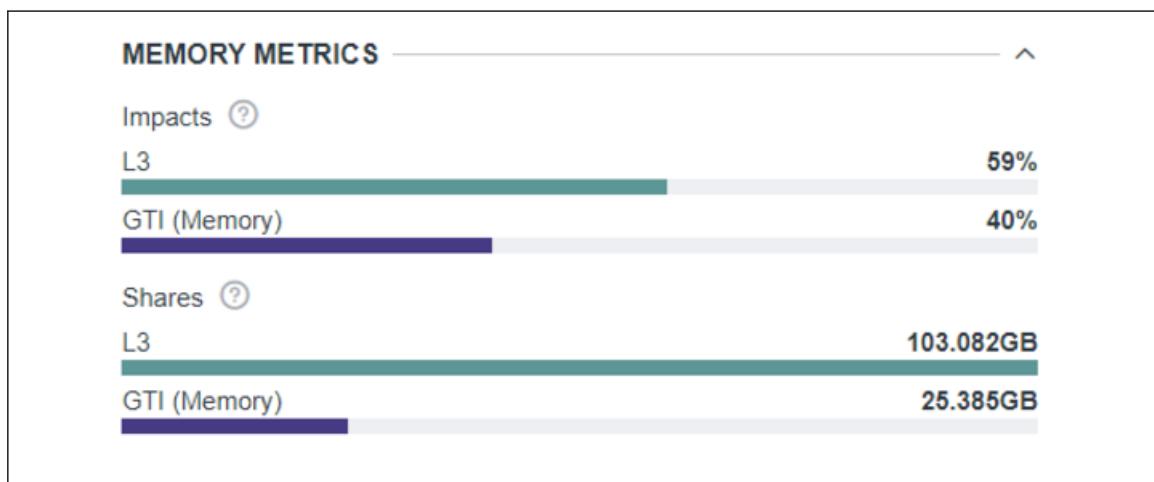


## Graphics Frame Analyzer

View this data in Intel® GPA's Graphics Frame Analyzer. For usage, see our 8 short videos and/or their corresponding articles. [Video Series: An In-Depth Look at Graphics Frame Analyzer \(intel.com\)](#)

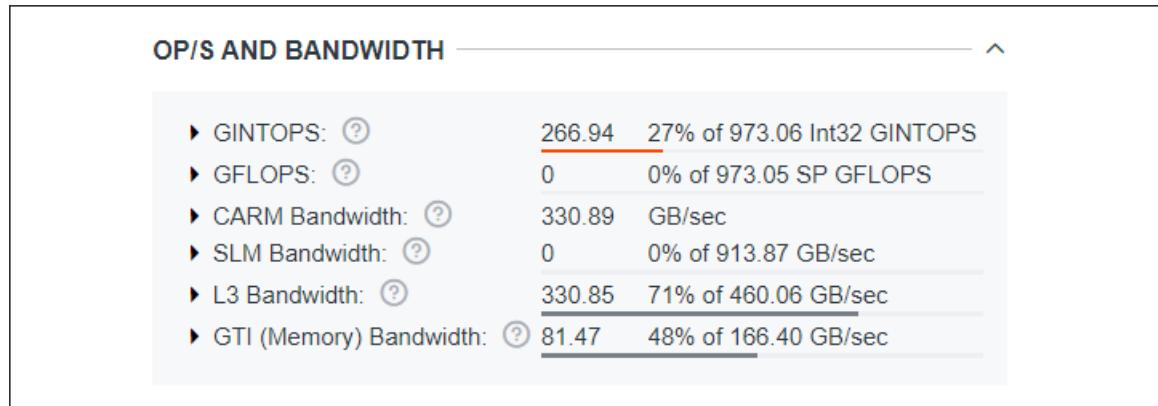
In Graphics Frame Analyzer after opening a frame, you will see a view such as that in Figure 5. If you look at the data just after opening the frame, you will see data percentage values for the entire frame. That means the percentages averaged over all 96 EUs over the frame time for data such as Active, Stall and Throughput.

**Data values averaged across all EUs over the entire frame time.**



You can then select a single draw call or a set of calls to see that same data recalculated for the part of the frame you have selected.

**After making a selection, in this case calls 91, 94 and 95, the data will be recalculated to represent the data for only those calls.**



Additionally, if you captured a stream, it will open in multi-frame view. From there you can select a single frame or multiple frames. If you select multiple frames the data calculated will be the aggregate of the data from all selected frames.

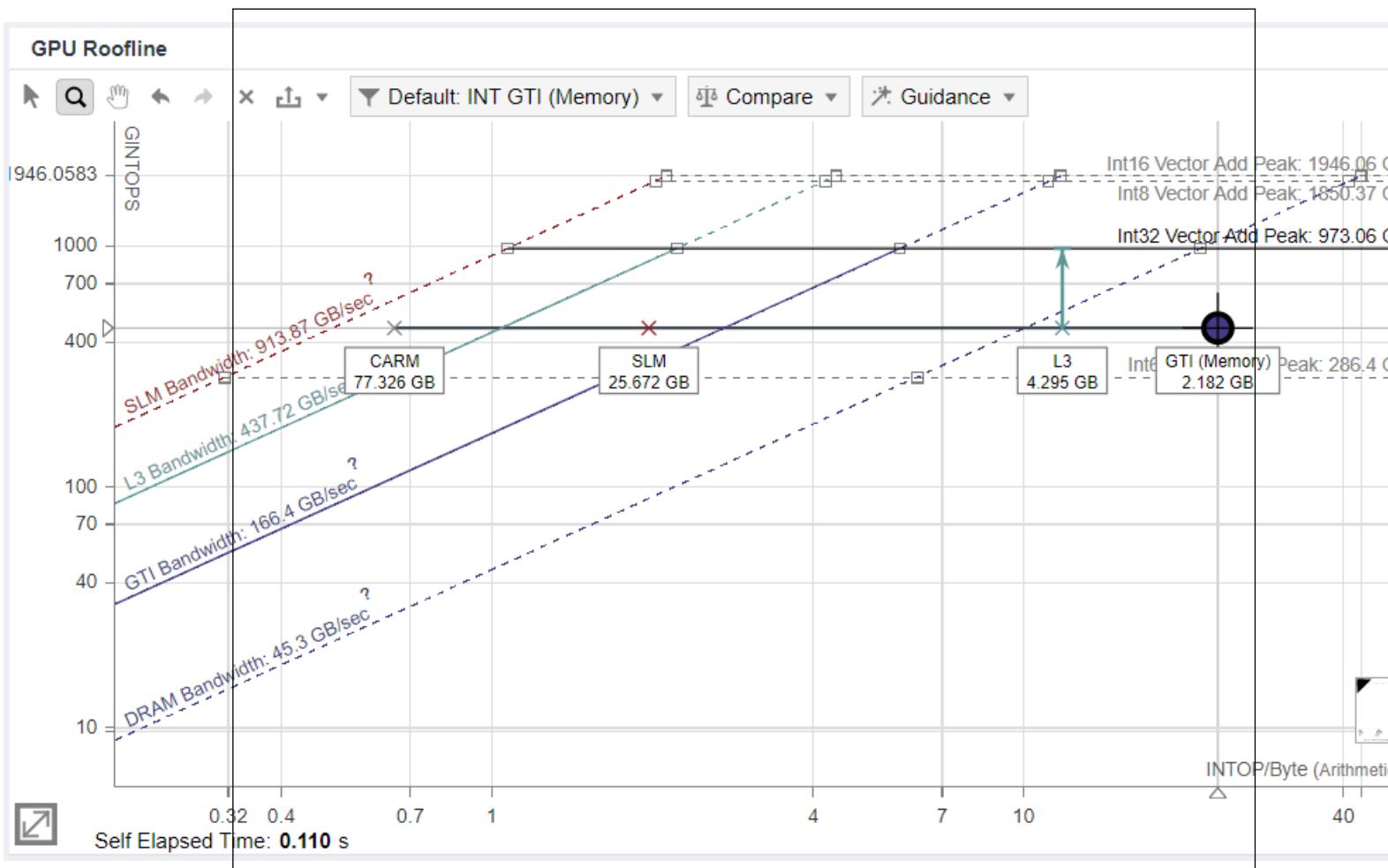
While it is important to understand how the GPU works and what the metrics mean for efficient profiling, you don't need to analyze each draw call in your frame manually in order to understand the problem type. To help with this sort of analysis, Intel® GPA provides automatic hotspot analysis - Advanced Profiling Mode.

#### Hotspot Analysis

Now that we have some understanding of the EU architecture, let's look at how that knowledge manifests in the profiler.

When you enable Advanced Profiling Mode Graphics, Graphics Frame Analyzer delineates bottlenecks by bottleneck type and pipeline state. This categorization provides the additional benefit of a fix for one issue often fixing not only the local issue, but rather an entire category of issues.

In Graphics Frame Analyzer enable Hotspot Analysis by clicking on the button on the top left of the tool - shown in Fig 7. The Bar Chart across the top then shows the bottlenecks, and the API Log in the left panel changes to show the bottleneck types. When you click on a bottleneck the metrics viewer will show more details about the bottleneck, with metrics descriptions and suggestions to alleviate the bottleneck.

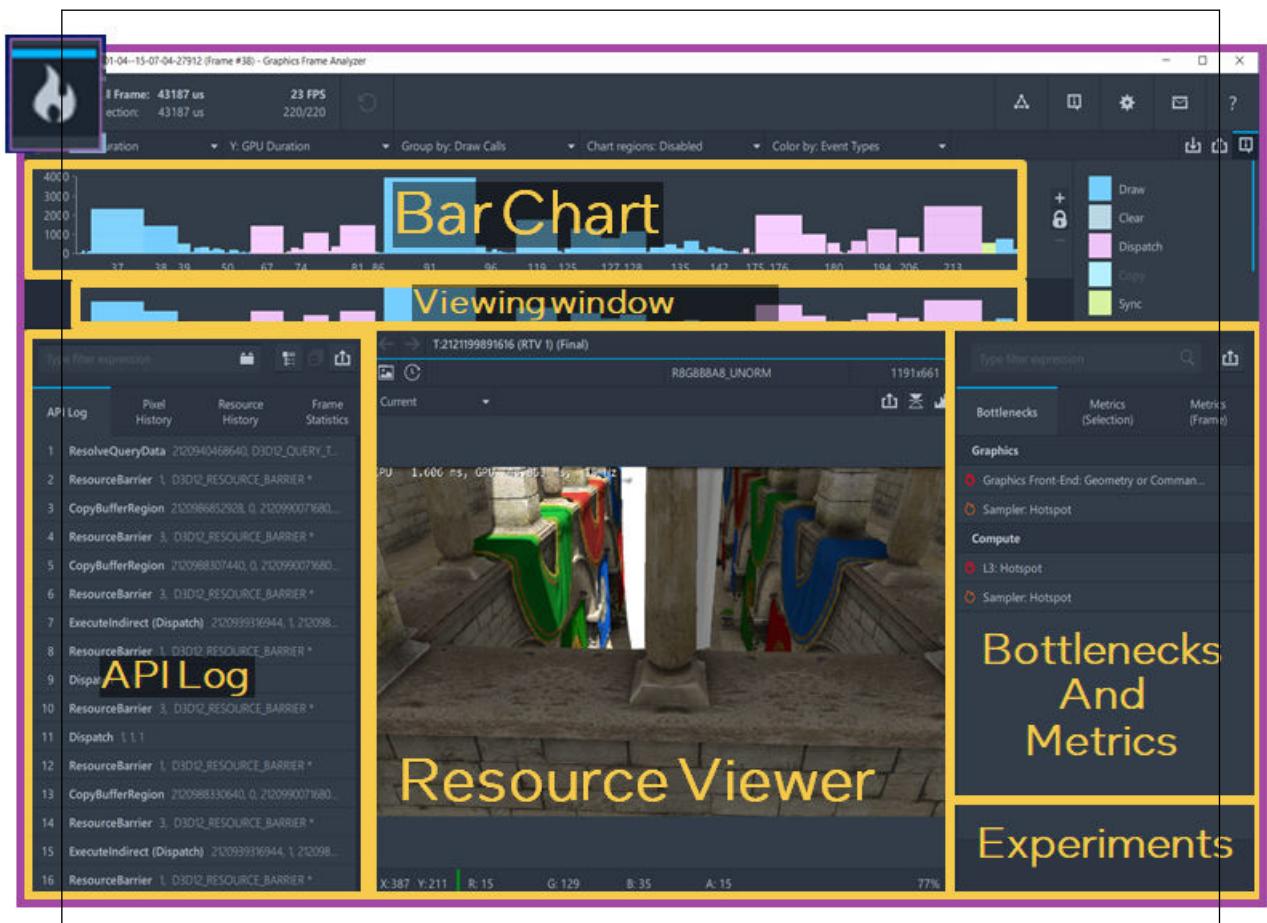


Hotspot: L2 Cache

#### Characterization of an L2 Cache Hotspot

When the application has high thread occupancy, close to 90%, that is good. But if the high thread occupancy is coupled with stall, greater than 5-10%, you may have an L2 Cache bottleneck.

With a frame open in Graphics Frame Analyzer, look at the Metrics Viewer Panel on the right, enlarged in Fig. x. Occupancy is more than 90%, but there is still a stall in the EU, which means that EU threads are waiting for some data from memory.



## Shader Profiler

For further analysis use the Shader Profiler to see per-instruction execution latencies. As you already know latency is not always equal to stall. However, an instruction with higher latency has a higher probability to cause a stall. And, therefore, when dealing with an EU Stall bottleneck, Shader Profiler gives a good approximation of what instructions most likely caused the stall.

### Enable Shader Profiler

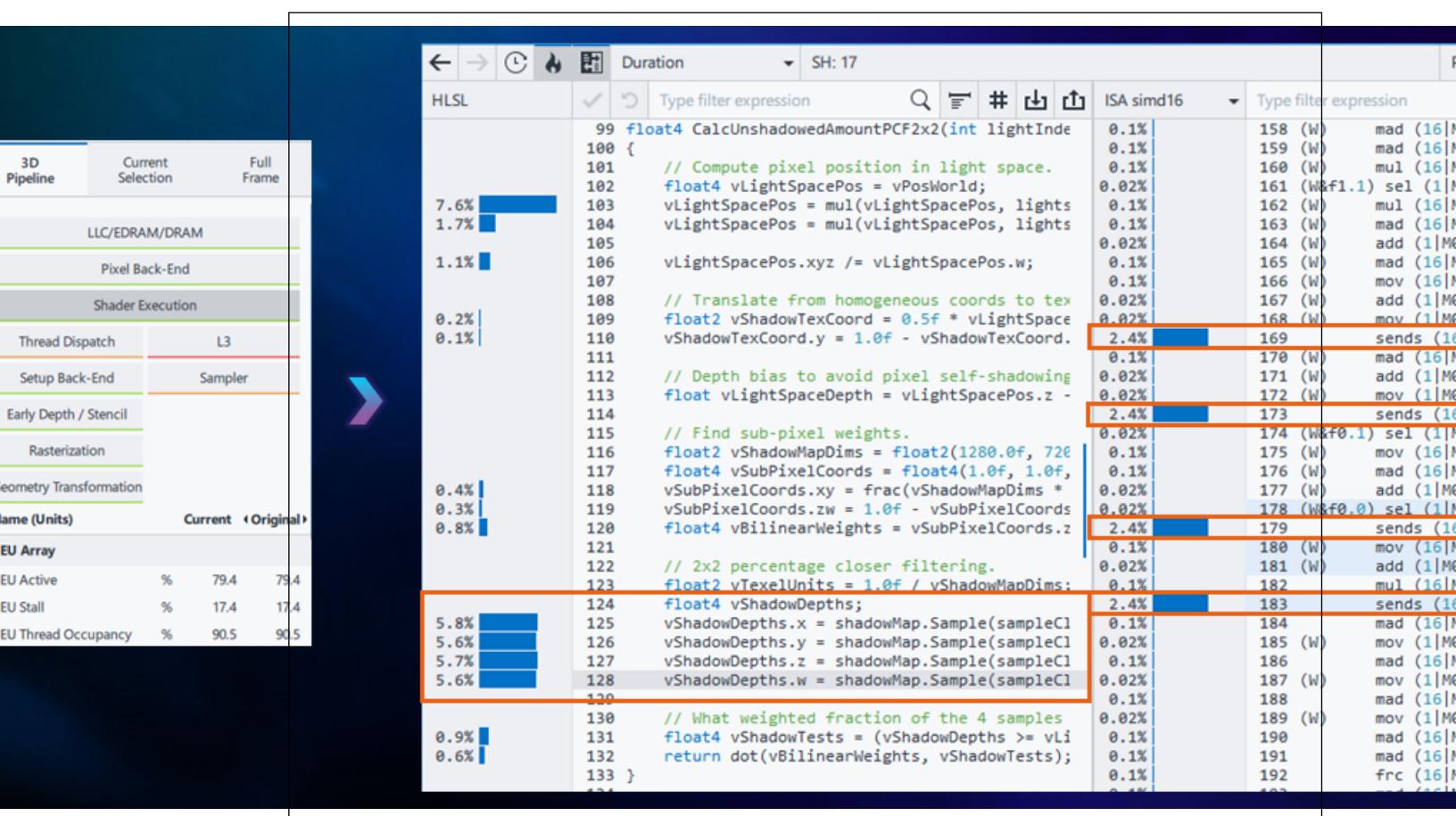
Access the shader profiler by doing the following. Click on any shader in the Resource List, in this case SH:17, to display the shader code in the Resource Panel. Then click the flame button at the top of the Resource Pane to see the shader code with the lines taking the most time annotated with the timings, toggle between execution count and duration (percentage of frame time consumed).

### Map Shader Code to Assembly Code

For a potential L2 Cache bottleneck, you will also want to see the assembly code, where you will find the send commands from the Send Unit. Click the



button in the Resource Panel above the shader code to see the mapping from the shader code to the assembly code.



### Identify the Root Cause of the L2 Bottleneck

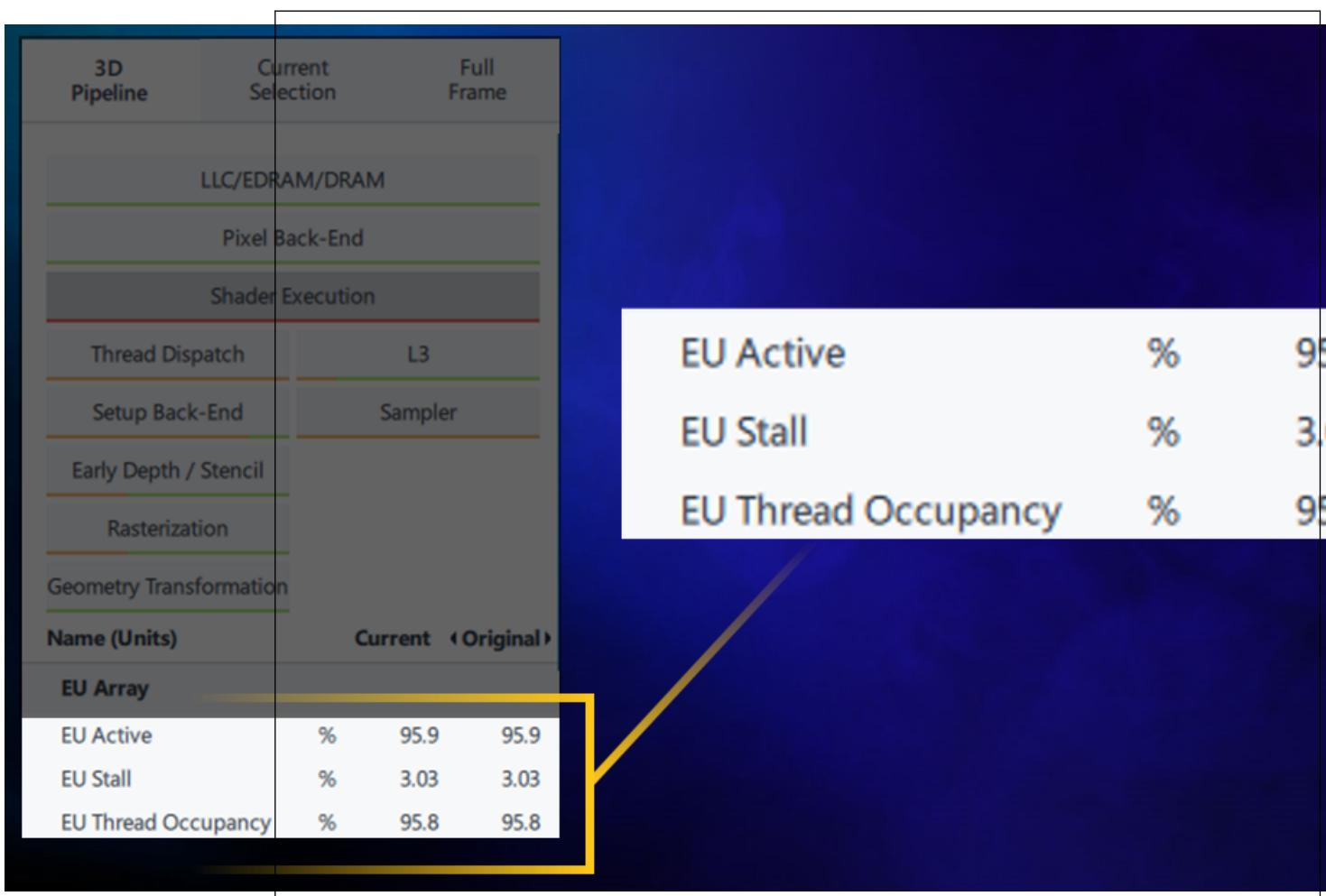
To find the root cause of an L2 Cache bottleneck, scroll through the assembly code, looking for the send instructions with the longest duration. Then identify which shader source portions caused them.

In the case of the application being profiled in Fig x, above, the CalcUnshadowedAmountPCF2x2 function which samples from ShadowMap and reads the constant buffer is cause of this bottleneck.

### Hotspot: Shader Execution

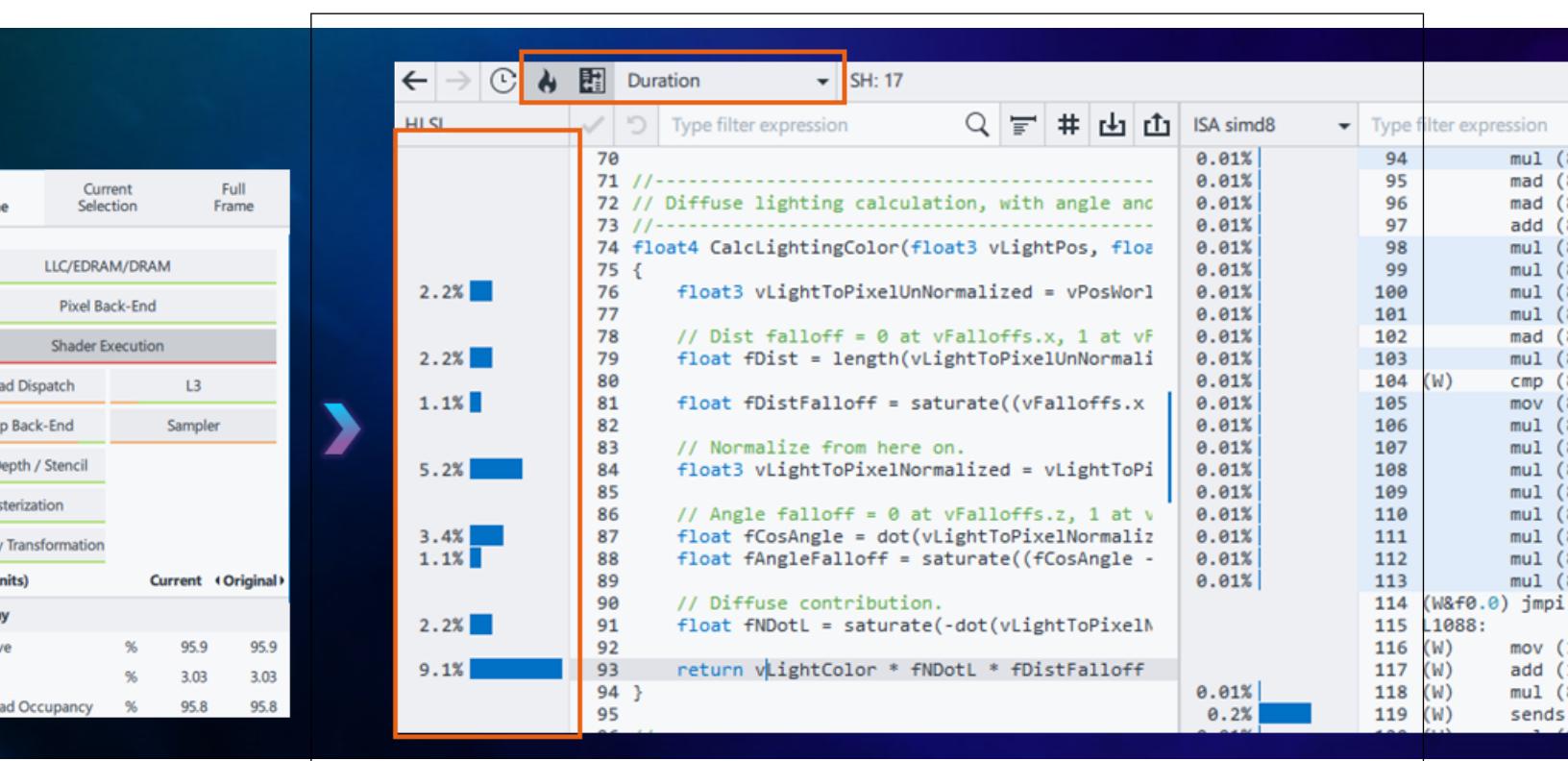
#### Characterization of a Shader Execution Bottleneck

A Shader Execution bottleneck is characterized by very high thread occupancy and very low stall time. These are good. However, if the application reduced execution time, it is necessary to optimize the shader code itself.



### Identify the Root Cause of the Shader Execution Bottleneck

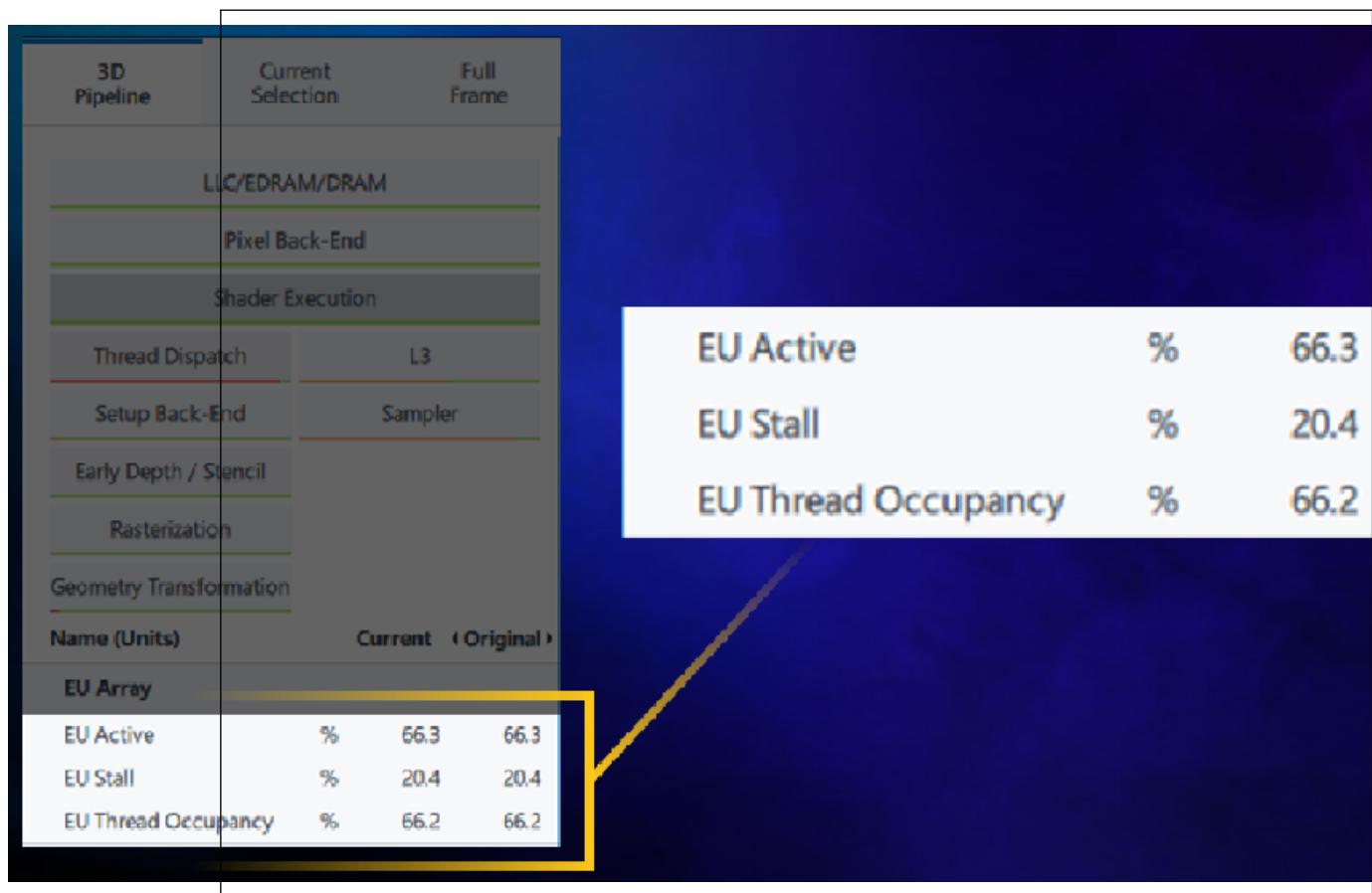
For a shader execution bottleneck, it is necessary to analyze the hotspots in shader source code caused by arithmetic operations. Find these by toggling to Duration Mode in the shader profiler, then scroll through the code to find the lines of shader code that take exceedingly long. CalcLightingColor does calculations involving both simple and transcendental operations. Fig x shows that this function in this single shader consumes about 20% of the total frame time. In order to resolve this bottleneck this algorithm must be optimized.



### Hotspot: Thread Dispatch

#### Characterization of a Thread Dispatch Bottleneck

In this final example of hotspot analysis there is a sequence of draw calls which have a Thread Dispatch bottleneck. In this particular case we have a rather high stall rate (20%) and low thread occupancy (66%). As stated earlier, low occupancy may be caused by an insufficient number of threads loaded on the EU. Thus, instead of directly fixing stall time in shader code, it is necessary, instead, to increase the overall EU Occupancy.

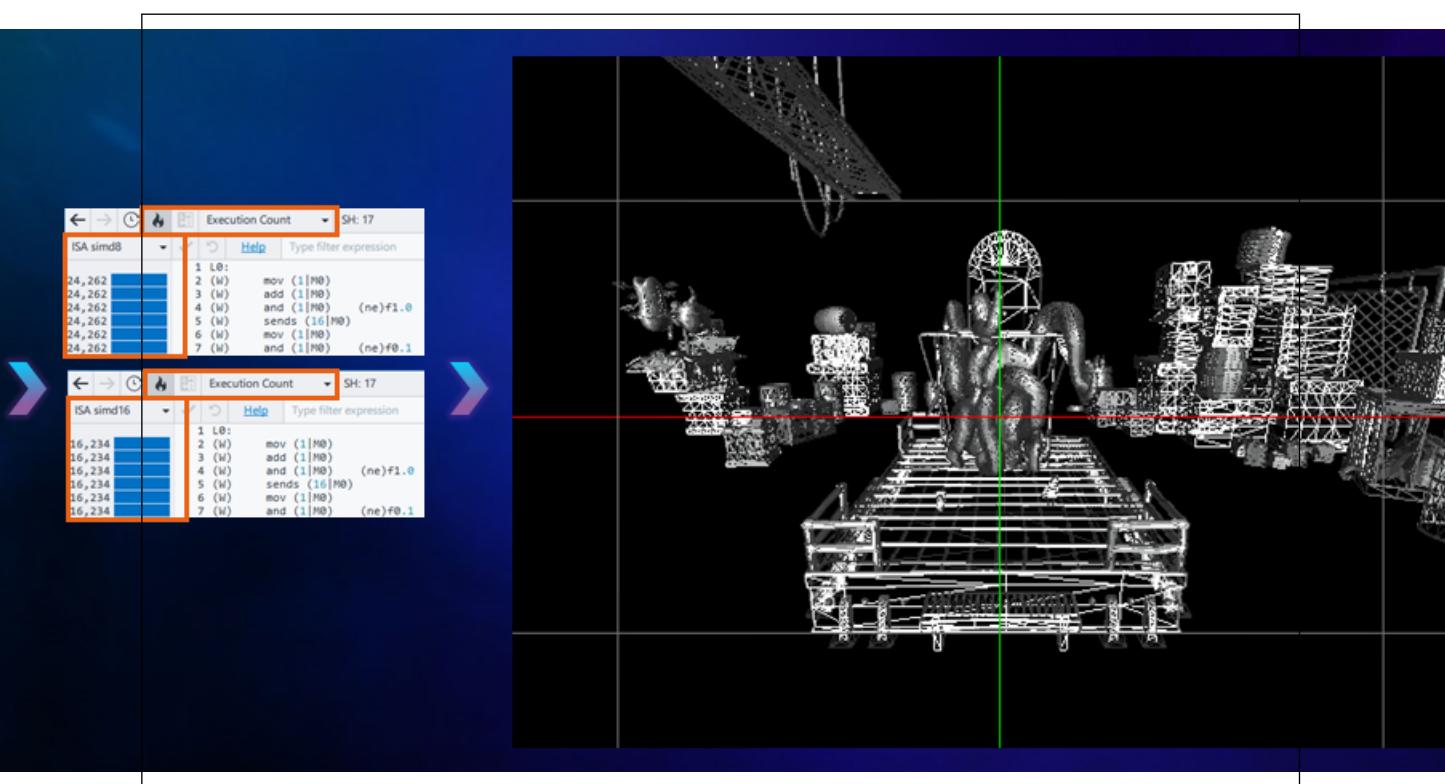


Identify the Root Cause of the Thread Dispatch Bottleneck

Which is better, SIMD8 or SIMD16?

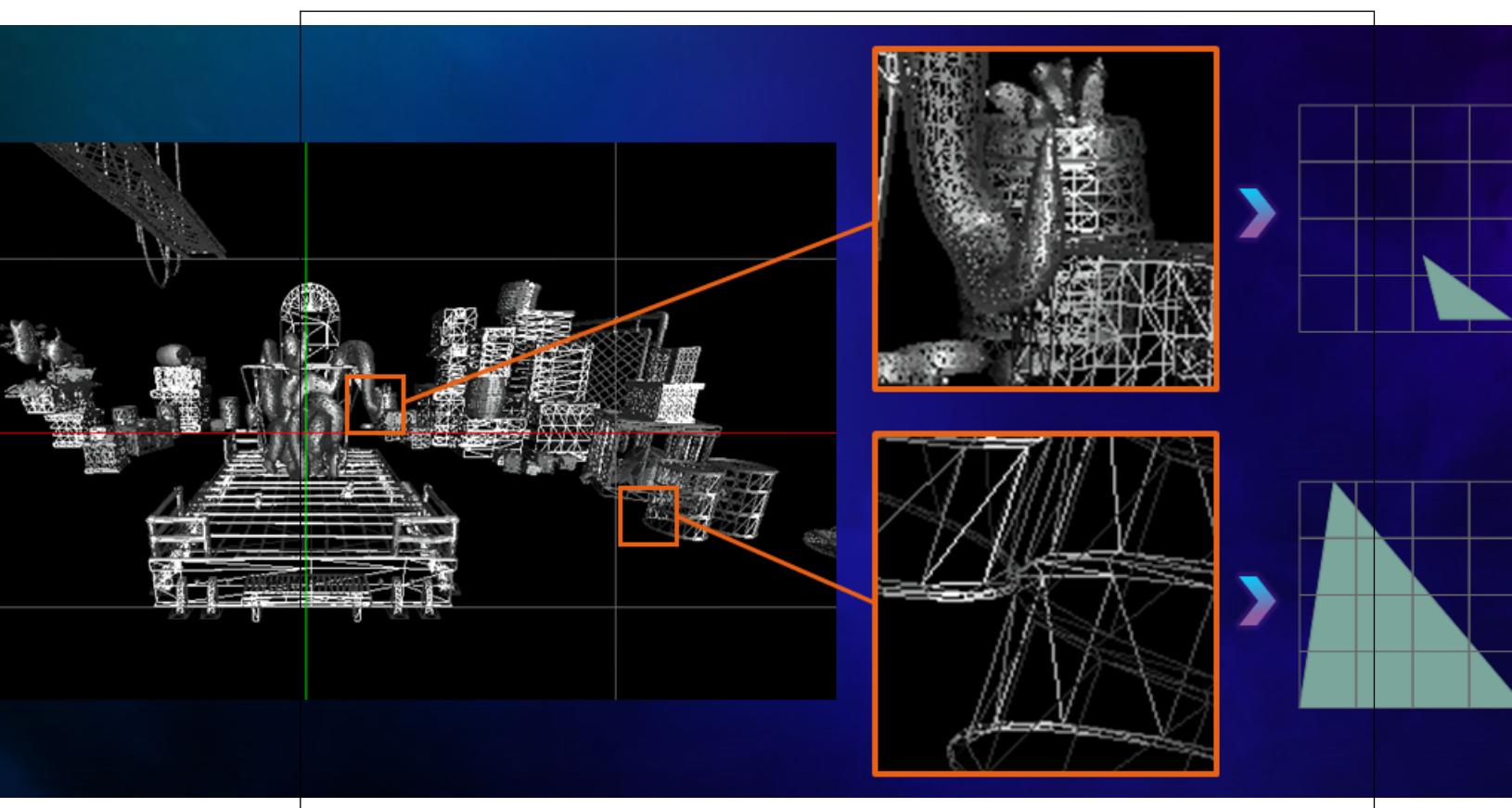
Open Shader Profiler, but in Execution Count mode rather than Duration mode which shows how many times each instruction was executed. In Fig x notice that the pixel shader has been compiled into both SIMD8 and SIMD16. Shader Profiler shows that each instruction in the SIMD8 version was executed 24,000 times, while instructions in SIMD16 were executed 16,000 times - a 1.5 times difference!

It is preferable to have more SIMD16 threads, as they perform twice as many operations per single dispatch, compared to SIMD8 threads. Why so many SIMD8 dispatches? And why should there be 2 SIMD versions for the Pixel Shader?



### Examine the Geometry

The geometry for these draw calls is rather fine-grained. The observed anomaly is a result of how the GPU handles pixel shading. The shader compiler produced two SIMD versions, SIMD8 and SIMD16. This is required so that the pixel shader dispatcher can choose which one to execute based on the rasterization result. It is important to know that hardware does not shade pixels one at a time. Instead shading happens in groups. A single pixel shader hardware thread shades 16 pixels at a time for SIMD16. With SIMD16, if a primitive is rasterized into very few or just a single pixel, then the GPU will still shade 16 pixels and will discard all those which were unnecessary. Therefore, in order to discard less, the pixel shader dispatcher schedules SIMD8 for very small primitives. This is the case here. A large number of highly-detailed geometry (many small primitives) produced a huge number of SIMD8 invocations. As you may guess, in order to fix such a performance problem you need to use geometry LODs in your game



## Summary

Bottleneck Type	Characterization
L2 Cache	High occupancy High stall
Shader Execution	High occupancy Low stall
Thread Dispatch	High stall Low occupancy

As shown above, different scenarios require different approaches. At times it is best to speed up CPU work to fully populate the GPU. Other times it is best to optimize shader code. And still others it might be best to change formats, dimensions or layouts of primitives. For each scenario, Graphics Frame Analyzer facilitates analysis of resources to assist developers to make informed decisions about how to optimize frame rate of their applications.

For more ways to optimize GPU performance using Intel® GPA, see [Intel® GPA Use Cases](#) as well as [Deep Dives and Quick Tips](#).

## References

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 5.2 Specification](#)
- [OpenMP API 5.1 Examples](#)
- [Data Parallel C++, by James Reinders et al](#)
- [SYCL 2020 Specification](#)
- [oneAPI Level Zero Specification](#)

## 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.