Optimizing SYCL workloads on Aurora and Sunspot

Varsha Madananth

Agenda

- Overview of Aurora
- Device hierarchy
  - Implicit vs Explicit Scaling
- Software Stack on GPU
  - SYCL programming model
  - Compilation Workflow
- Occupancy on GPU
  - Reduce stalls
  - Concurrent execution of work on GPU
- Submitting kernels to GPU
  - Regular command list vs Immediate Command list
- Specialization constants
- Floating point Accuracy

Aurora System

Intel-HPE machine arriving at Argonne in 2023

Sustained Performance ≥ 1Exaflops DP

Compute Nodes

2 Intel Xeon CPU Max Series processors(Sapphire Rapids)

64GB HBM on each, 512GB DDR5 each;

6 Intel Data Center GPU Max Series(Ponte Vecchio [PVC

128GB HBM on each, RAMBO cache on each;

CPU-GPU Interconnect

CPU-GPU: PCIe;

GPU-GPU: Xe Link

ALL-TO-ALL CONNECTIVITY WITHIN NODE

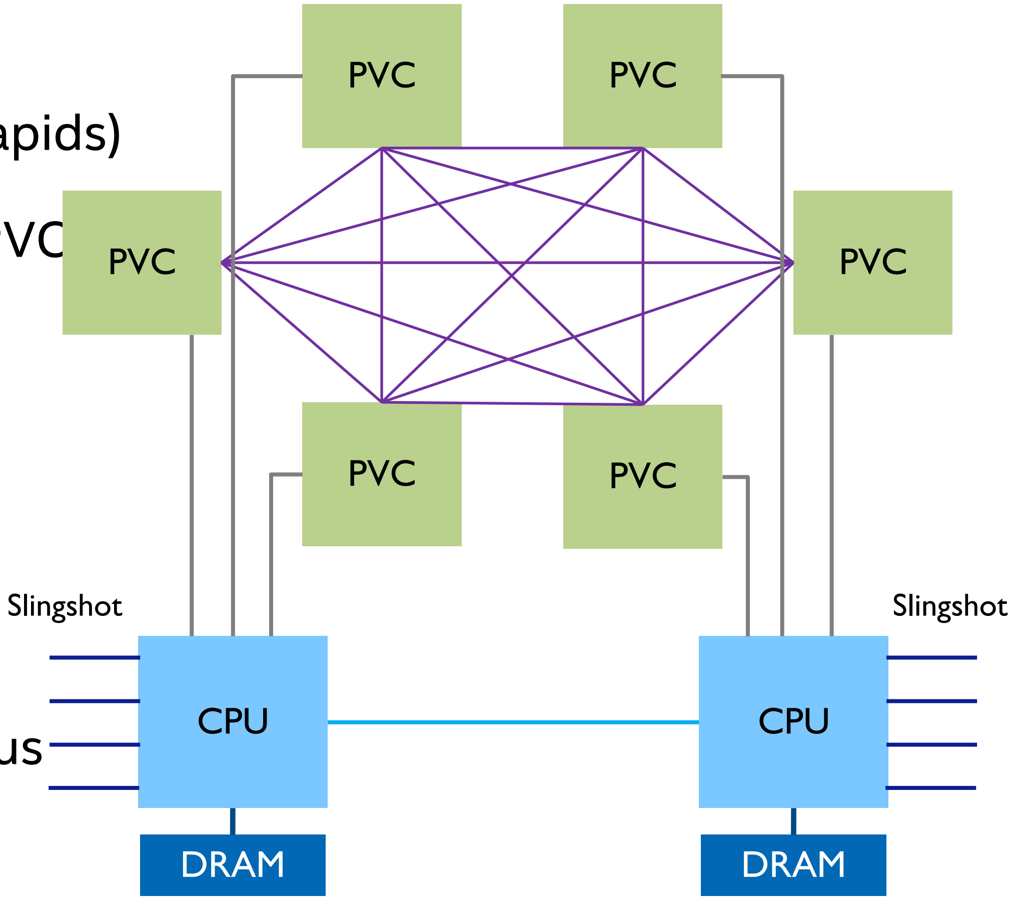
Filesystem

High Performance Storage - Distributed Asynchronous Object Store (DAOS)

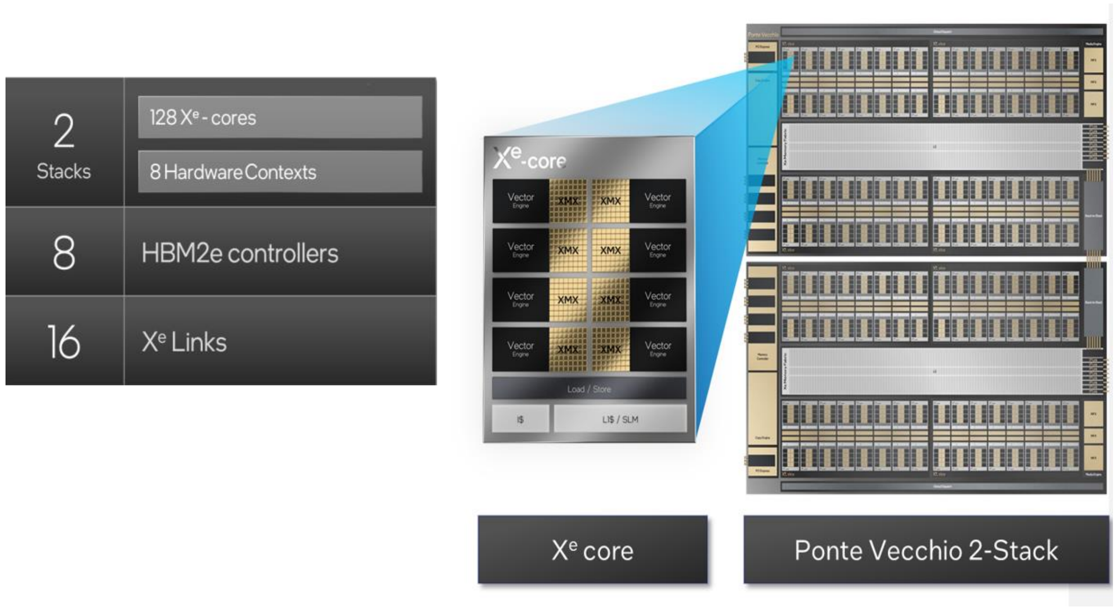
≥ 230 PB of storage capacity; ≥ 25 TB/s

Lustre

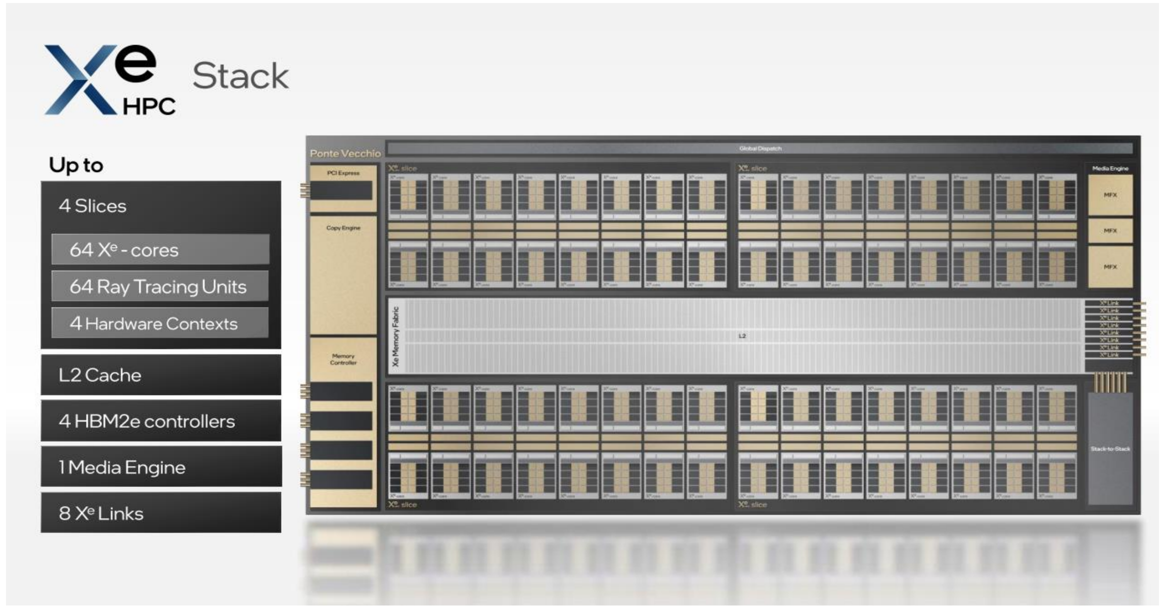
150PB of storage capacity; ~1 TB/s



Intel GPU Architecture for Aurora



2 Stacks	128 X^e - cores 8 Hardware Contexts
8	HBM2e controllers
16	X^e Links

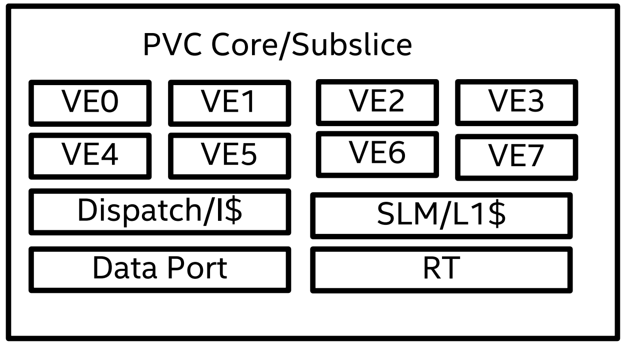


X^e HPC Stack

Up to

- 4 Slices
  - 64 X^e - cores
  - 64 Ray Tracing Units
  - 4 Hardware Contexts
- L2 Cache
- 4 HBM2e controllers
- 1 Media Engine
- 8 X^e Links

- ❑ Vector Engines execute SIMD math & load/store
- ❑ Each Vector Engine services multiple HW threads, issuing one thread instruction per clock tick
- ❑ Multiple Vector Engines form a Core, sharing one memory load/store unit
- ❑ Multiple Cores form a Slice
- ❑ Four Slices form a Stack
- ❑ Two Stacks form a PVC

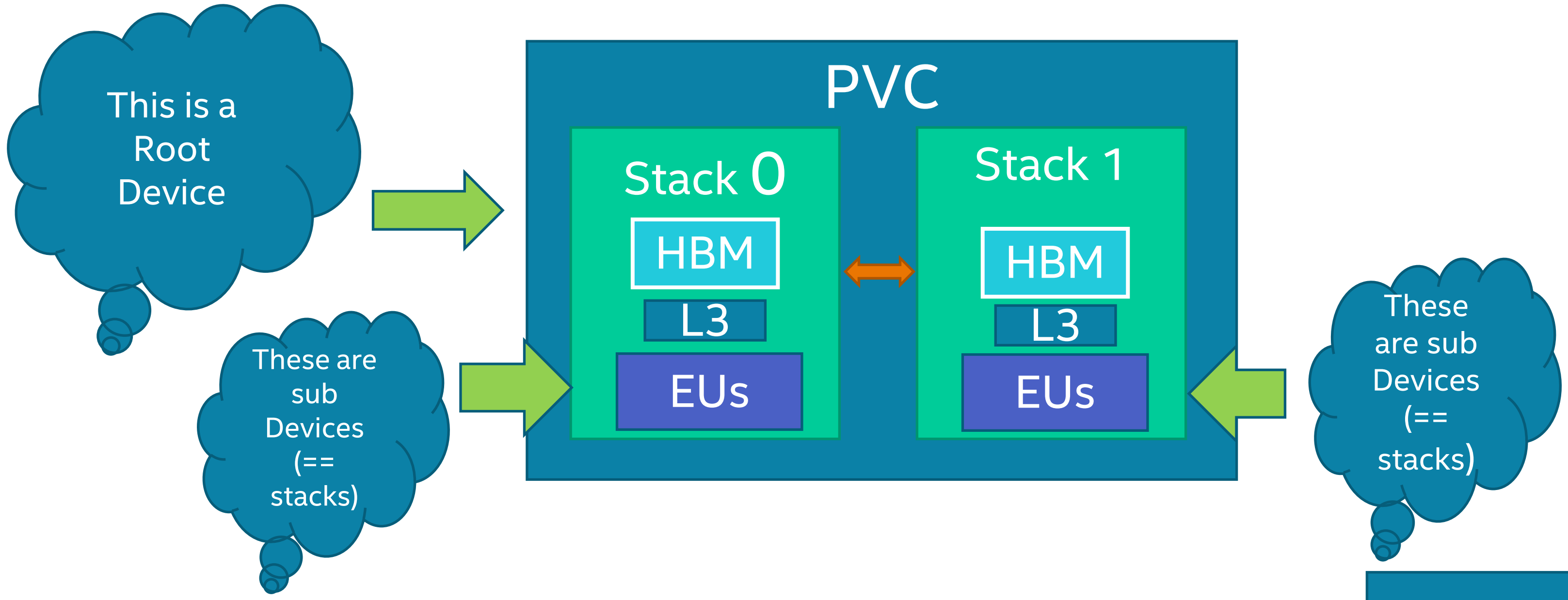


EU = Vector Engine
Sub-Slice = Core
Slice = Slice
Tile = Stack

Total Threads = #_slices * #_cores_per_slice * #_ve_per_core * #_threads_per_ve
(3,584 = 4 * 14 * 8 * 8)

*Argonne/Intel/HPE Business/Procurement Sensitive - DO NOT DISTRIBUTE

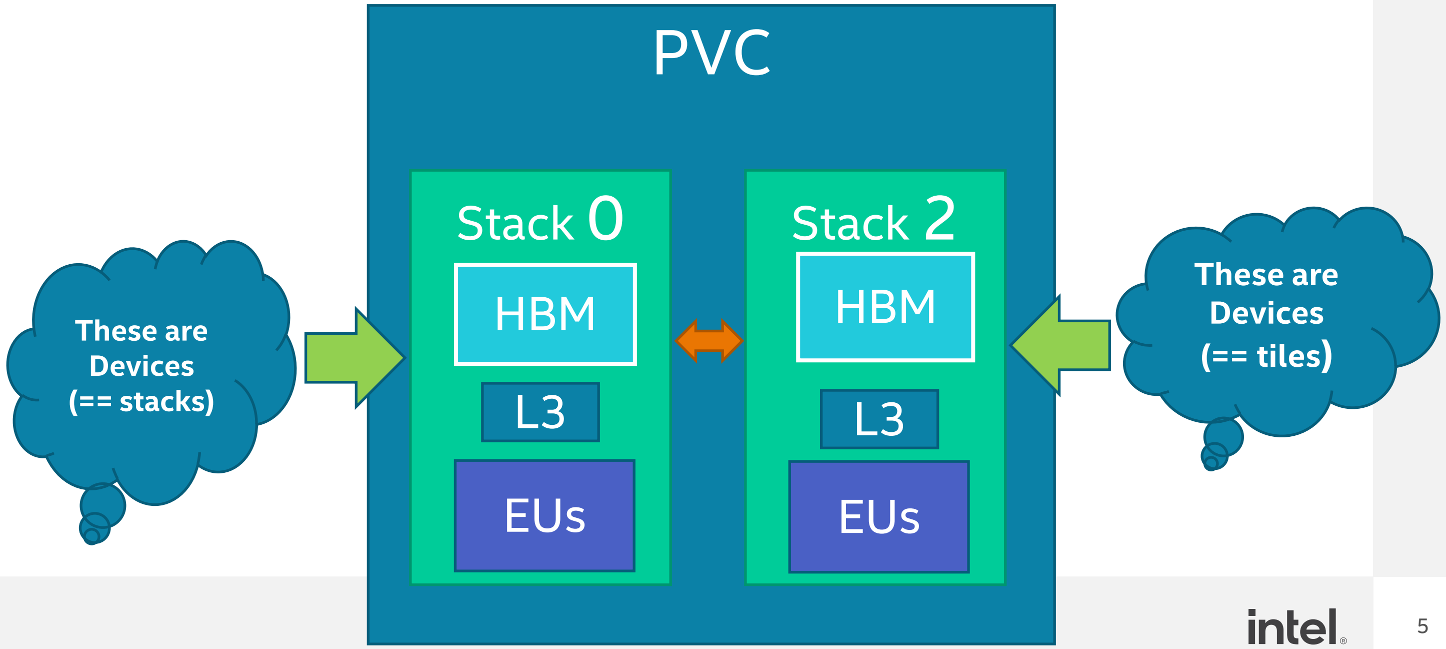
Devices and sub-devices


This is a Root Device
PVC
Stack 0
HBM
L3
EUs
Stack 1
HBM
L3
EUs
These are sub Devices (== stacks)
These are sub Devices (== stacks)

A root-device is built using multiple sub-devices, also known as stacks. These stacks form a shared memory space which allows to treat a root-device as a monolithic device without the requirement of explicit communication between stacks.

Each stack is exposed as a device.

Programmer can take direct control over work group distribution and memory placement.



Implicit vs Explicit Scaling

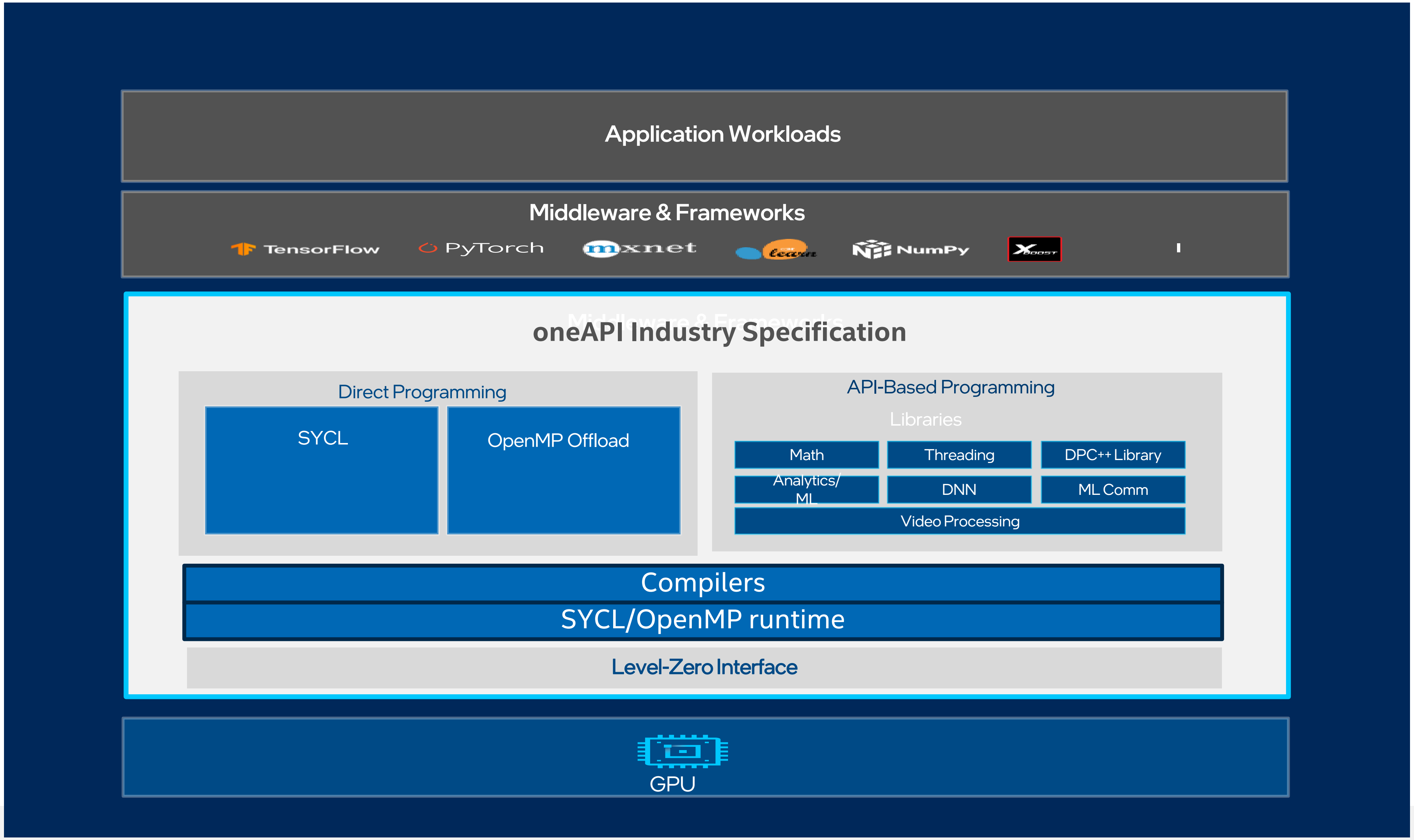
- Implicit Scaling provides a mechanism to automatically distribute work across multiple stacks.
  - No extensions required, works for non-multi-tile-aware applications.
  - Driver automatically distributes work and allocations across underlying resources
- Explicit Scaling: Application manually distributes work and allocations across underlying resources.
- Same trade-off as for other NUMA systems:
  - Implicit requires attention to memory placement, work scheduling, etc.
  - Explicit requires an “extra” level of decomposition

Affinity Mask to expose devices

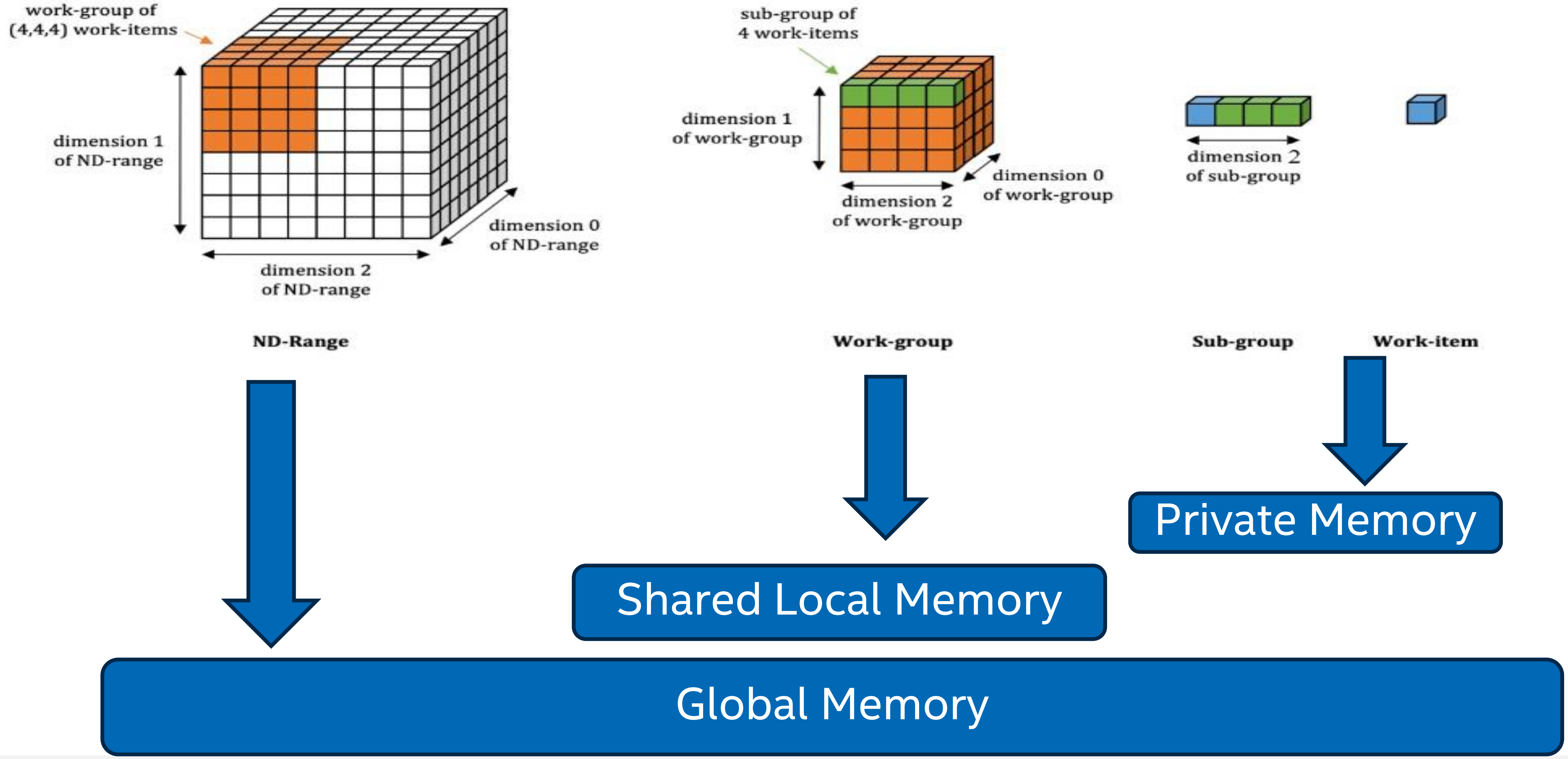
- Affinity mask allows an application to expose devices or sub-devices
  - ZE_AFFINITY_MASK = *0, 1,2,3,4,5*: all parent devices and stacks are reported (same as default):
  - ZE_AFFINITY_MASK = *0*: only parent device 0 is reported as device handle 0, with all its stacks reported as sub-device handles:
  - ZE_AFFINITY_MASK = *1*: only parent device 1 is reported as device handle 0, with all its stacks reported as sub-device handles:
  - ZE_AFFINITY_MASK = 0.0: only tile 0 in parent device 0 is reported as device handle 0:
  - ZE_AFFINITY_MASK = *1.0, 1.1*: only parent device 1 is reported as device handle 0; with its tiles 1 and 2 reported as its sub-devices 0 and 1, respectively:
  - Target device using MPI ranks
    - Target each MPI rank to a tile using env variable -
      - export ZE_AFFINITY_MASK=$gpu_id.$tile_id
      - Explicit scaling - 1 rank targets tile 0, and the other rank targets tile 1
      - Implicit scaling – 1 rank targets 1 GPU with 2 stacks.
    - On sunspot gpu_tile_compact.sh maps multiple ranks to each stack.
      - mpiexec -np ${NTOTRANKS} -ppn ${NRANKS} -d ${NDEPTH} --cpu-bind depth -envall gpu_tile_compact.sh ./myBinaryName
  - Partition by affinity domain –
    - The root-device, corresponding to the whole GPU, can be partitioned to 2 sub-devices(each sub-device corresponding to a physical stack exposed as separate device)
    - try {

```
      vector<device> SubDevices = RootDevice.create_sub_devices<
      cl::sycl::info::partition_property::partition_by_affinity_domain>(
      cl::sycl::info::partition_affinity_domain::numa);
   }
```

Software Stack on PVC

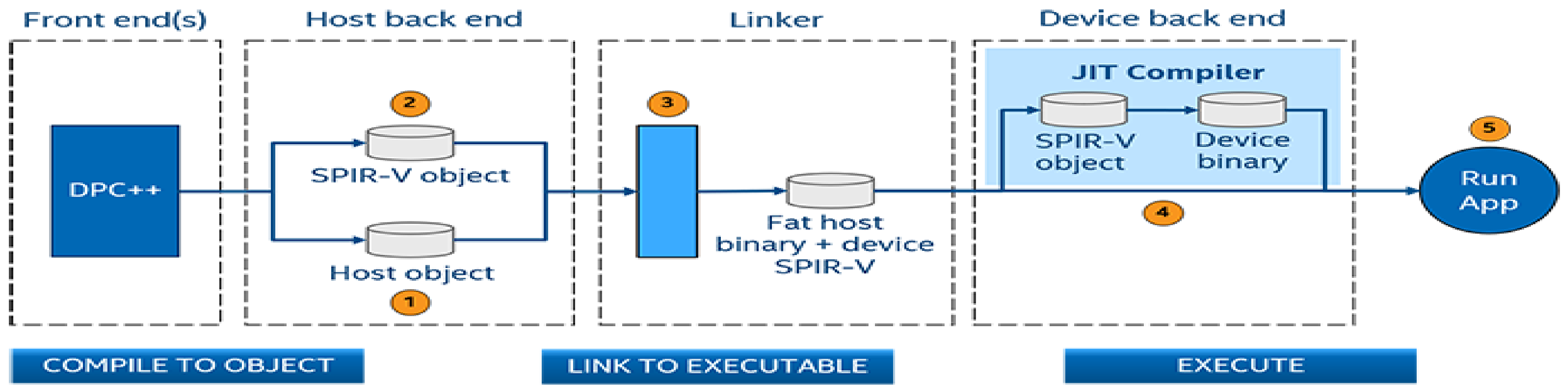


SYCL programming model

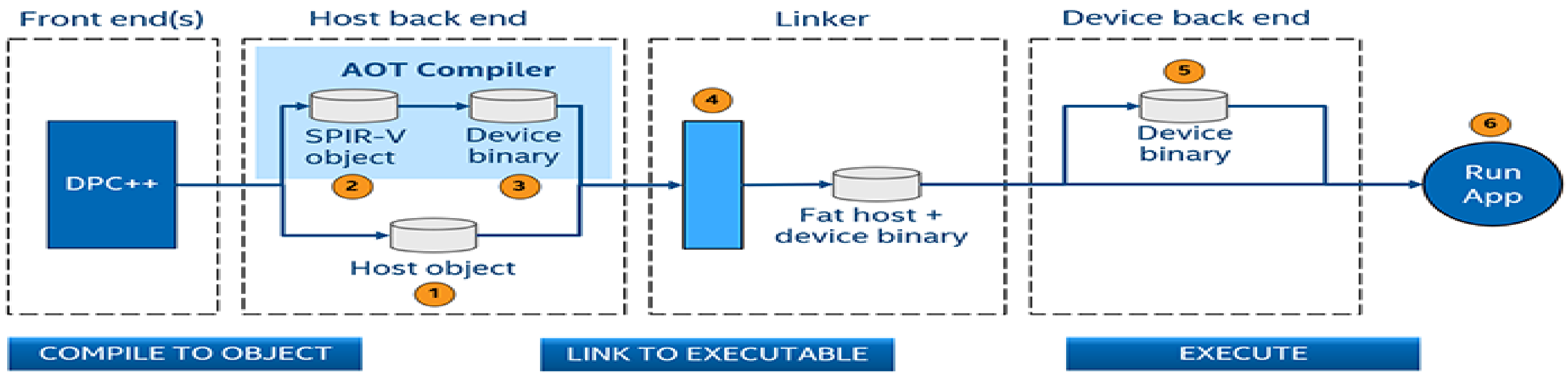


Compilation workflow

JIT Compilation



AOT Compilation



General principles of offloading kernels to GPU

- Data parallel parts of your applications
- Make effective utilization of GPUs hardware
  - Offload large enough problem to minimize the data transfer overhead from CPU to GPU.
  - Reduce back and forth data transfer from CPU and GPU.
- Non divergent control flow
  - Minimize inactive threads in a subgroup.
- Data Access Pattern
  - Contiguous memory access
- Optimize for lower latency on chip memory usage.

Occupancy

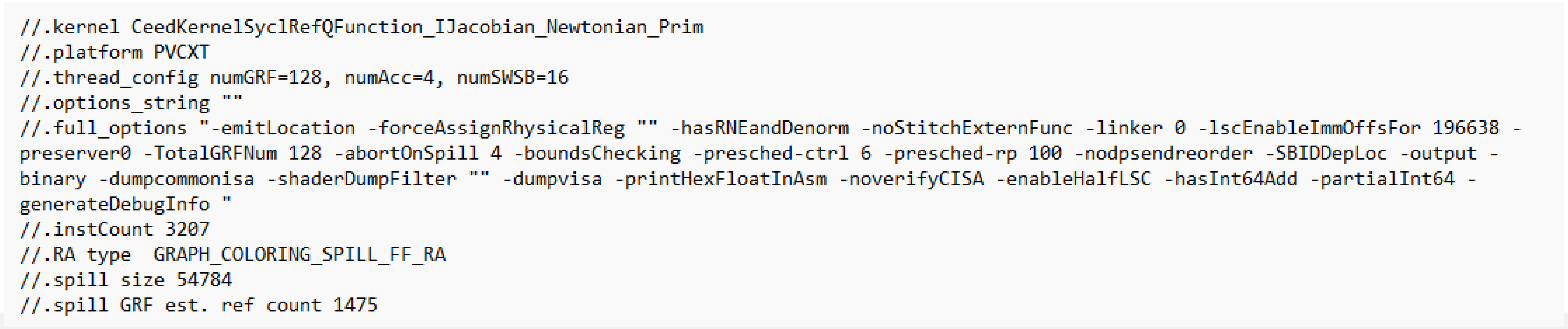
- It is a measure of how much of its capacity the GPU is utilizing.
- Several factors effect the occupancy of GPU
  - Workgroup size
  - Limitation on max wg size imposed by h/w or gpu driver
  - Amount of memory in SLM per WG.
  - Subgroup.
  - Register size
- Choosing the Work group size is important to fully utilize the GPU's thread which effects the occupancy
  Work group size = Threads x SIMD sub-group size <= 1024
- 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.
- Vtune gpu–hotspots analysis can be used to runtime occupancy of the GPU.

Improving performance by decreasing register pressure

- By default , PVC has 128 64-byte registers allocated per thread .
  - There are 8 threads per VE .
- Registers spills can be expensive. Improve register usage per thread by
  - Increasing the registers available per thread to 256.
  - Running in lower simd width
- Increasing the register size, the number of threads available per EU to 4.
  - Not enough threads to hide latency. .
- By default, the subgroup size generated by compiler is 32.

Inspecting register spills

- Building with AOT shows warnings on spills at each kernel
- Inspected assembly by setting the following env variables –
  - IGC_DumpToCurrentDir=1/IGC_DumpToCustomDir=<pwd>, IGC_ShaderDumpEnable=1
  - For more IGC env variables - https://github.com/intel/intel-graphics-compiler/blob/master/documentation/configuration_flags.md



```
//.kernel CeedKernelSyclRefQFunction_IJacobian_Newtonian_Prim
//.platform PVCXT
//.thread_config numGRF=128, numAcc=4, numSWSB=16
//.options_string ""
//.full_options "-emitLocation -forceAssignRhysicalReg "" -hasRNEandDenorm -noStitchExternFunc -linker 0 -lscEnableImmOffsFor 196638 -
preserver0 -TotalGRFNum 128 -abortOnSpill 4 -boundsChecking -presched-ctrl 6 -presched-rp 100 -nodpsendreorder -SBIDDepLoc -output -
binary -dumpcommonisa -shaderDumpFilter "" -dumpvisa -printHexFloatInAsm -noverifyCISA -enableHalfLSC -hasInt64Add -partialInt64 -
generateDebugInfo "
//.instCount 3207
//.RA type  GRAPH_COLORING_SPILL_FF_RA
//.spill size 54784
//.spill GRF est. ref count 1475
```

Setting subgroup size

- Environment variable
  - export IGC_ForceOCLSIMDWidth=16|32
- Setting subgroup size per kernel basis
  - Kernel Property - [intel::reqd_sub_group_size(16|32)]]

```
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;
});
});
```

Setting GRF modes

- GRF Mode Specification at Command Line (applies at the application Level)
  - -ze-opt-large-register-file: Forces IGC to select large register file mode for ALL kernels
  - -ze-opt-intel-enable-auto-large-GRF-mode: Enables IGC to select small/large GRF mode on a per-kernel basis based on heuristics
  - Default: IGC picks small GRF mode for ALL kernel

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

    AOT - icpx -fsycl -fsycl-targets=spir64_gen -Xsycl-target-backend "-device pvc -options -ze-opt-large-register-file" test.
- Register Allocation Mode for SYCL - Per-kernel specification
  - `#include <sycl/ext/intel/experimental/kernel_properties.hpp>`
  - `set_kernel_properties(kernel_properties::use_large_grf);`

```
cgh.parallel_for<class FillBuffer>( NumOfWorkItems, [=](sycl::id<1> WIid)
{ set_kernel_properties(kernel_properties::use_large_grf);
// Fill buffer with indexes
Accessor[WIid] = (sycl::cl_int)WIid.get(0);
});
```

Concurrent execution of kernels

- Queues are out of order by default
- Multiple kernels can run simultaneously on a single queue
  - unless explicitly wait on events/queues or dependencies between accessors.
  - in-order queues property is used
    - sycl::property_list q_prop{sycl::property::queue::in_order()};
    - sycl::queue q1(d_selector, q_prop);
- To effectively utilize full machine compute resources,
- No guarantee that they will execute concurrently, depends on the available resources
- Submit kernels to multiple queues.
- Oversubscribing MPI ranks on a stack

Modes of submitting work to GPU

The Level Zero API provides two modes of submitting work to the GPu

Regular command lists

- Kernel launch (e.g. zeCommandListAppendLaunchKernel) and submission (zeCommandQueueExecuteCommandList) are decoupled.
- Submissions can be batched on the host, i.e., many operations may be collected in a command list and then submitted together, thus dividing the submission cost across many operations.
- Number of batches (zeCommandQueueExecuteCommandLists) is less than the number of commands (zeCommandListAppendLaunchKernel).
- If kernel time is very small, use regular command list with batching.
- SYCL_PI_LEVEL_ZERO_USE_IMMEDIATE_COMMANDLISTS=0
- SYCL_PI_LEVEL_ZERO_BATCH_SIZE=<>

Immediate command lists

- Kernel launch and submission occur together.
- Immediate command list provide low latency submissions of work to GPU.
- Kernel time is long enough to overlap the submission overhead.
- SYCL_PI_LEVEL_ZERO_USE_IMMEDIATE_COMMANDLISTS=1

Specialization constants

- specialization constants are constants values can be set dynamically during execution of the application
- The values of these constants are fixed when a kernel is invoked, and they do not change during the execution of the kernel.
- Specialization constants must be declared using the specialization_id class
- Setting and getting the value of a specialization constant
    - Functions of class handler – set_specialization_constant, get_specialization_constant

template<auto& SpecName> *void set_specialization_constant(typename std::remove_reference_t<decltype(SpecName)>::value_type value);*	*template<auto& SpecName>* *typename std::remove_reference_t <decltype(SpecName)> ::value_type* *get_specialization_constant();*

Example of using specialization constants

```
#include <sycl/sycl.hpp>
using namespace sycl; // (optional) avoids need for "sycl::" before
SYCL names

using coeff_t = std::array<std::array<float, 3>, 3>;

// Read coefficients from somewhere.
coeff_t get_coefficients();

// Identify the specialization constant.
constexpr specialization_id<coeff_t> coeff_id;

void do_conv(buffer<float, 2> in, buffer<float, 2> out) {
  queue myQueue;

  myQueue.submit([&](handler &cgh) {
    accessor in_acc { in, cgh, read_only };
    accessor out_acc { out, cgh, write_only };
```

```
    // Set the coefficient of the convolution as constant.
    // This will build a specific kernel the coefficient available as
literals.
    cgh.set_specialization_constant<coeff_id>(get_coefficients());

    cgh.parallel_for<class Convolution>(
      in.get_range(), [=](item<2> item_id, kernel_handler h) {
        float acc = 0;
        coeff_t coeff = h.get_specialization_constant<coeff_id>();
        for (int i = -1; i <= 1; i++) {
          if (item_id[0] + i < 0 || item_id[0] + i >= in_acc.get_range()[0])
            continue;
          for (int j = -1; j <= 1; j++) {
            if (item_id[1] + j < 0 || item_id[1] + j >=
in_acc.get_range()[1])
              continue;
            // The underlying JIT can see all the values of the array
returned
            // by coeff.get().
            acc += coeff[i + 1][j + 1] *
                   in_acc[item_id[0] + i][item_id[1] + j];
          }
        }
        out_acc[item_id] = acc;
      });
  });
```

Floating point Accuracy

- 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.
- By default, fp-model=fast is used for host and device.

Floating Point Semantics	**Apply to Host and Device Compilations**	**Apply to Host Compilation Only**	**Apply To Device Compilation Only**
Precise	-fp-model=precise	-fp-model=precise and specify -Xsycl-target-frontend "-fp-model=fast"	-Xsycl-target-frontend "-fp-model=precise"
Fast-math	-fp-model=fast (default)	-fp-model=fast and specify -Xsycl-target-frontend "-fp-model=precise"	-fp-model=precise and specify -Xsycl-target-frontend "-fp-model=fast"
Relaxed-Math (native instructions)	Applies to device only	Applies to device only	-Xsycl-target-backend "-options -cl-fast-relaxed-math"

Summary

- Identify parts of your applications that can be offloaded to your application
  - Intel offload advisor can help identify such kernels in your applications
- Make the data resident on GPU as much as possible.
- Tune applications to be less memory bound and more compute bound
  - By improving the occupancy of the kernels
  - Avoid register spills
  - Avoid branch divergence to reduce inactive lanes in a subgroup
  - Avoid cross tile memory access
- Make sure GPU has enough work to fully utilize the GPU
  - Oversubscribing GPU with multiple streams of work
  - Hide memory latency with work
- Use profiling tools to understand performance of kernels
  - GPU Analysis with Vtune™ Profiler
  - Intel® Advisor GPU Analysis
  - Tools inside PTI-GPU - https://github.com/intel/pti-gpu
    - onetrace - host and device tracing tool for OpenCL(TM) and Level Zero backends with support of DPC++ (both for CPU and GPU) and OpenMP* GPU offload;
    - oneprof - GPU HW metrics collection tool for OpenCL(TM) and Level Zero backends with support of DPC++ and OpenMP* GPU offload;
    - ze_tracer - "Swiss army knife" for Level Zero API call tracing and profiling (former ze_intercept);
    - gpuinfo - provides basic information about the GPUs installed in a system, and the list of HW metrics one can collect for it;
    - sysmon - Linux "top" like utility to monitor GPUs installed on a system;

Questions?

Upcoming Learning paths

- SYCL Work-Group Mapping and GPU Occupancy Calculation
- Optimizing GPU Memory Allocation and Movement using SYCL

References

- Intel® oneAPI GPU Optimization Guide
- Intel® oneAPI Programming Guide
- ALCF Aurora
- Level Zero Specification document
- SYCL™ 2020 specification
- ISO3DFD Code Walkthrough