Tuning for Aurora

Kalyan Kumaran & Abhishek Bagusetty
Argonne Leadership Computing Facility

{kumaran, abagusetty}@anl.gov

Aurora Overview

Intel GPU
Intel® Data Center GPU Max Series

Intel Xeon Processor
4th Gen Intel XEON Max Series CPU with High Bandwidth Memory

Platform
HPE Cray-Ex

Racks - 166
Nodes - 10,624
CPUs - 21,248
GPUs – 63,744

Interconnect
HPE Slingshot 11
Dragonfly topology with adaptive routing
Network Switch:
25.6 Tb/s per switch (64 200 Gb/s ports)
Links with 25 GB/s per direction

Peak FP Performance
≧ 2 Exaflops DP

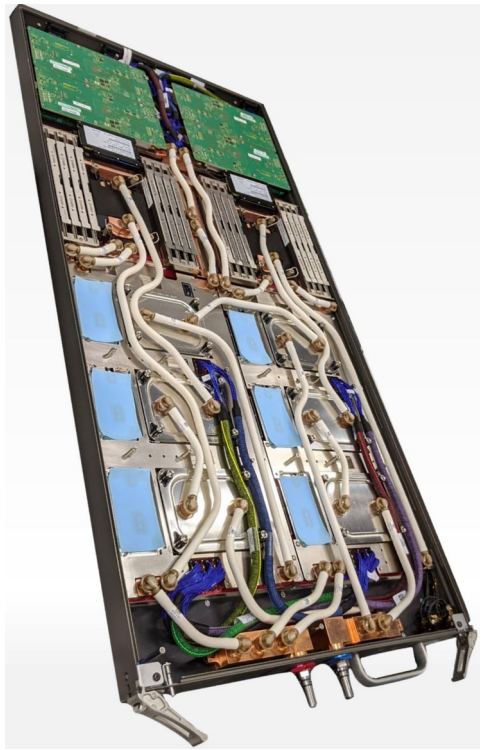
Memory
10.9PB of DDR @ 5.95 PB/s
1.36PB of CPU HBM @ 30.5 PB/s
8.16PB of GPU HBM @ 208.9 PB/s

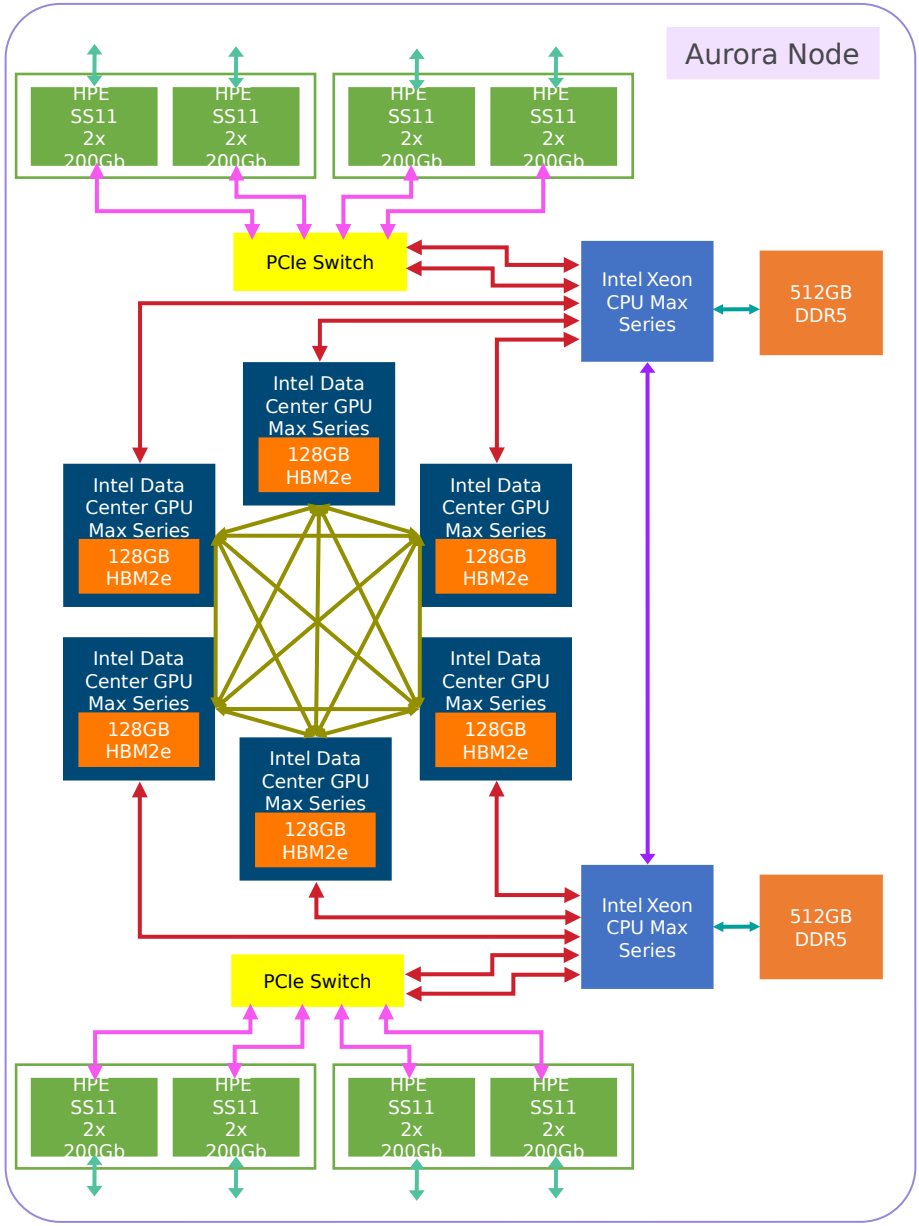
Network
2.12 PB/s Peak Injection BW
0.69 PB/s Peak Bisection BW

Storage
230PB DAOS Capacity
31 TB/s DAOS Bandwidth

Aurora Exascale Compute Blade

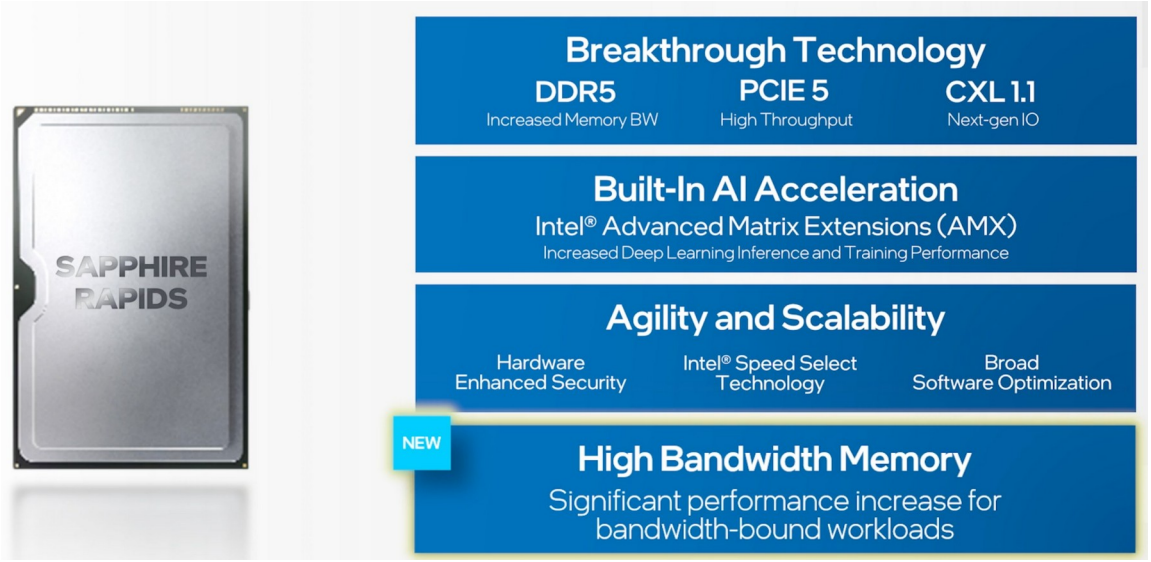
NODE CHARACTERISTICS
6 GPUs - Intel Data Center GPU Max Series
2 CPUs - Intel Xeon CPU Max Series
768 GB GPU HBM Memory
19.66 TB/s Peak GPU HBM BW
128 GB CPU HBM Memory
2.87 TB/s Peak CPU HBM BW
1024 GB CPU DDR5 Memory
0.56 TB/s Peak CPU DDR5 BW
≧ 130 TF Peak Node DP FLOPS
200 GB/s Max Fabric Injection
8 NICs





4th Gen Intel® Xeon Max Series CPU with HBM (Sapphire Rapids)

XEON DESCRIPTION	
Vector Extension	AVX-512
Threads (#)	2
Total HBM Memory (GB)	64
Peak HBM Memory BW (TB/s)	1.43
Total DDR5 4400 Memory (GB)	512
Peak DDR5 4400 Memory BW (TB/s)	0.28

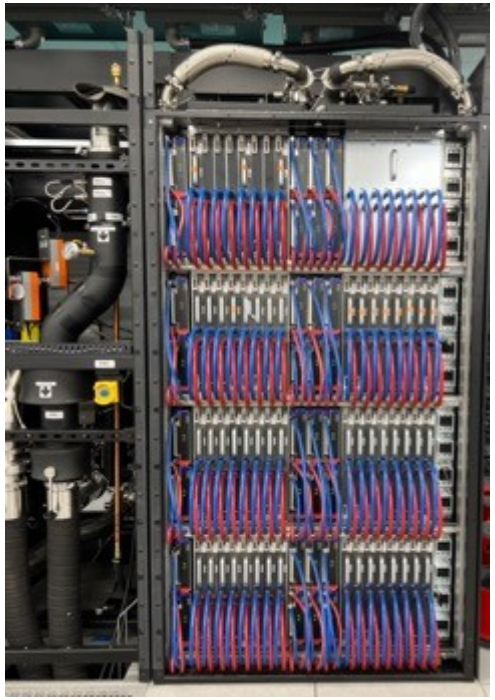


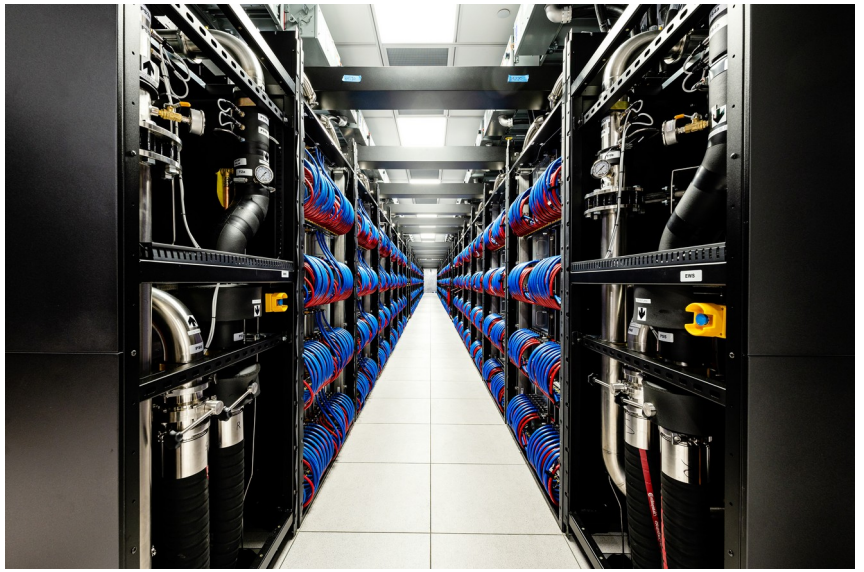
Where we are with Aurora today

Argonne Leadership Computing Facility

The Status of Aurora

- Aurora deployment is underway
- All of the Aurora system is installed at ANL
- ***Except -*** the installation of compute blades *which is nearly complete*
- Targeting early user in Q3 2023





Aurora Applications Status at Single Node Scale

Running
Running
Running
Partially Running
Porting in Progress

Application	Status
XGC	
NAMD	
FloodFillNetwork	
HACC	
QUDA	
OpenMC	
Flash-X/Thornado	
NWChemEX	
AMR-Wind	
CANDLE/UNO	
HARVEY	
NekRS	
LAMMPS	
GENE	
FusionDL	
MadGraph	
BerkelyGW	
PHASTA	
MFIX-Exa	
Chroma	
cctbx	

Application	Status
MILC	
QMCPACK	
E3SM-MMF	
SW4	
DCMesh	
LATTE	
Grid	
GAMESS	
NYX	
Uintah	
Data Driven CFD	
DarkSkyMining	
Flow Based Generative Model	
Nalu-Wind	
GEM	
RXMD-NN	
mb_aligner	
spiniFEL	
Multi-Grid Parameter Opt.	
FastCaloSim	

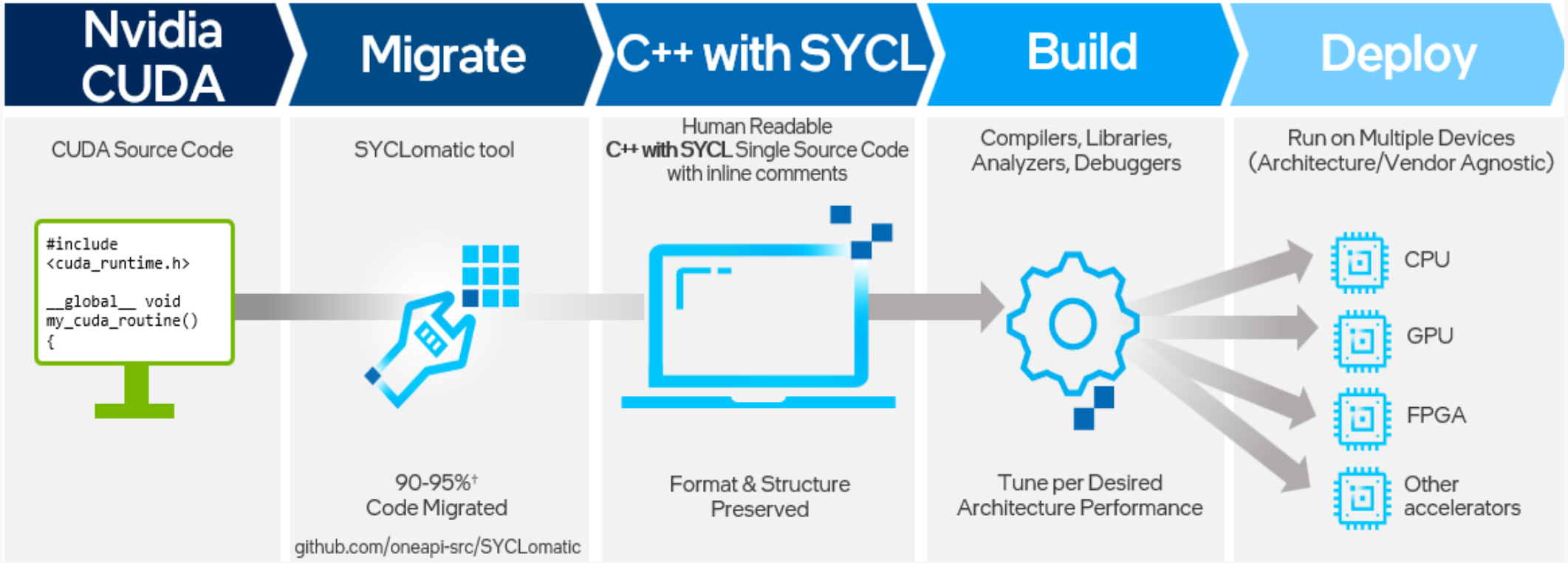
Performance Optimizations for Aurora

Argonne Leadership Computing Facility

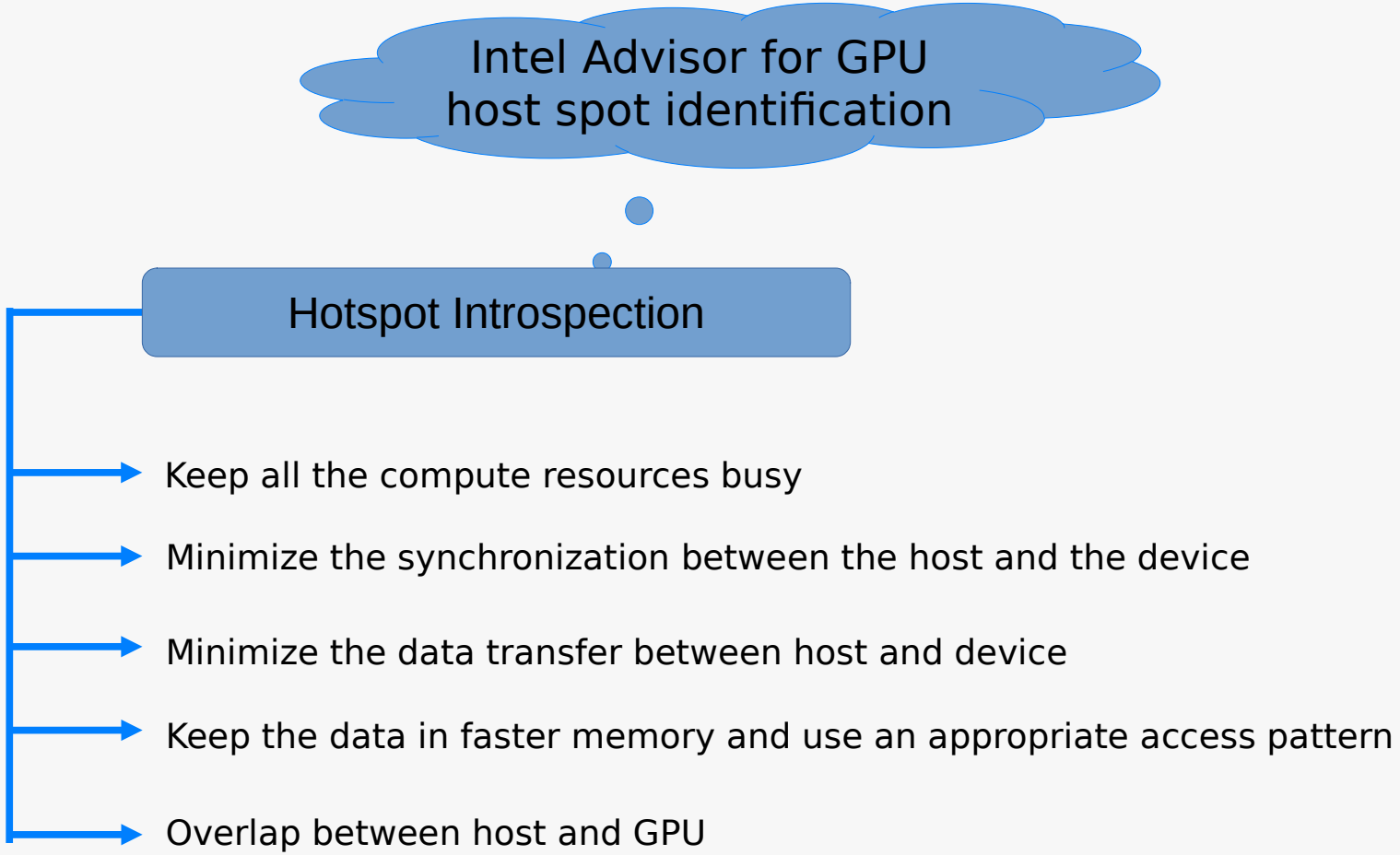
Porting Strategy from CUDA Projects to DPC++

SYCLomatic: A New CUDA*-to-SYCL* Code Migration Tool

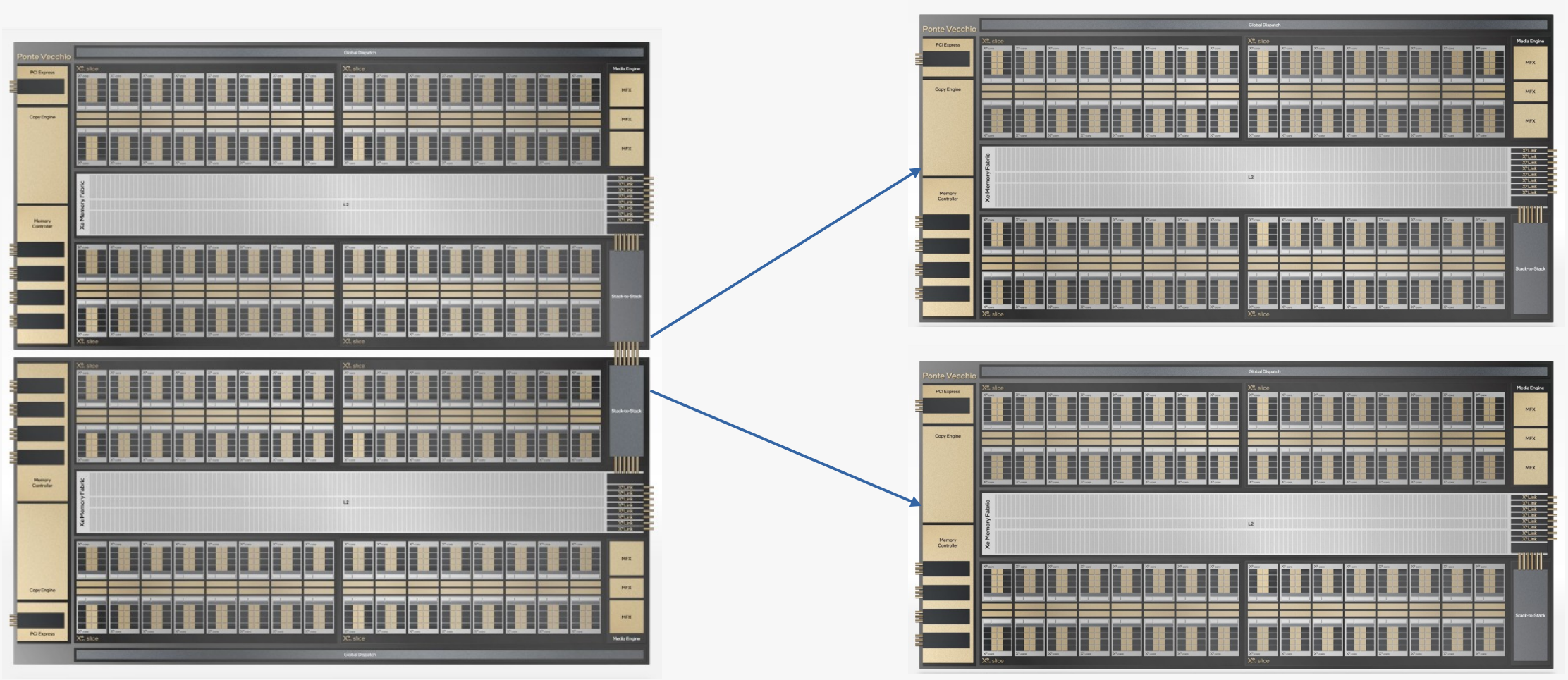
(previously referred to as DPCT, DPC++ Compatibility Tool)



Phases of Optimization for Aurora



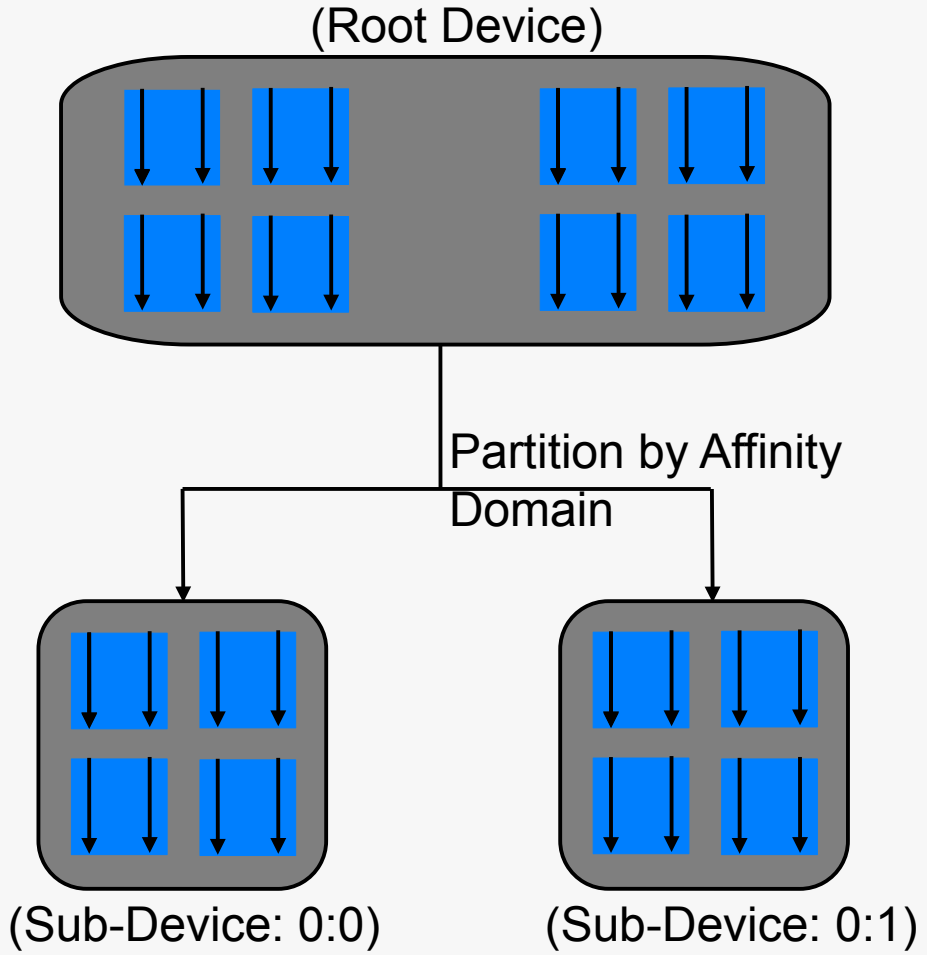
Architecture of Intel Data Center GPU Max (PVC)



- ✓ Intel PVC-2Stacks GPU can be used a single “root” device
- ✓ Can also be partitioned into 2 “virtual” sub-device

Intel Data Center Max “sub-devices”

- Consider creating sub-devices for each root/parent device to achieve better occupancy & through-put
- A sub-device represents a collection of **execution** and **memory** resources
- A sub-device is still a device:
  - It can do anything a device can do…
  - …including creating sub-(sub-) devices!
- Sub-devices are a very powerful – Abstractions for both CPU & GPU



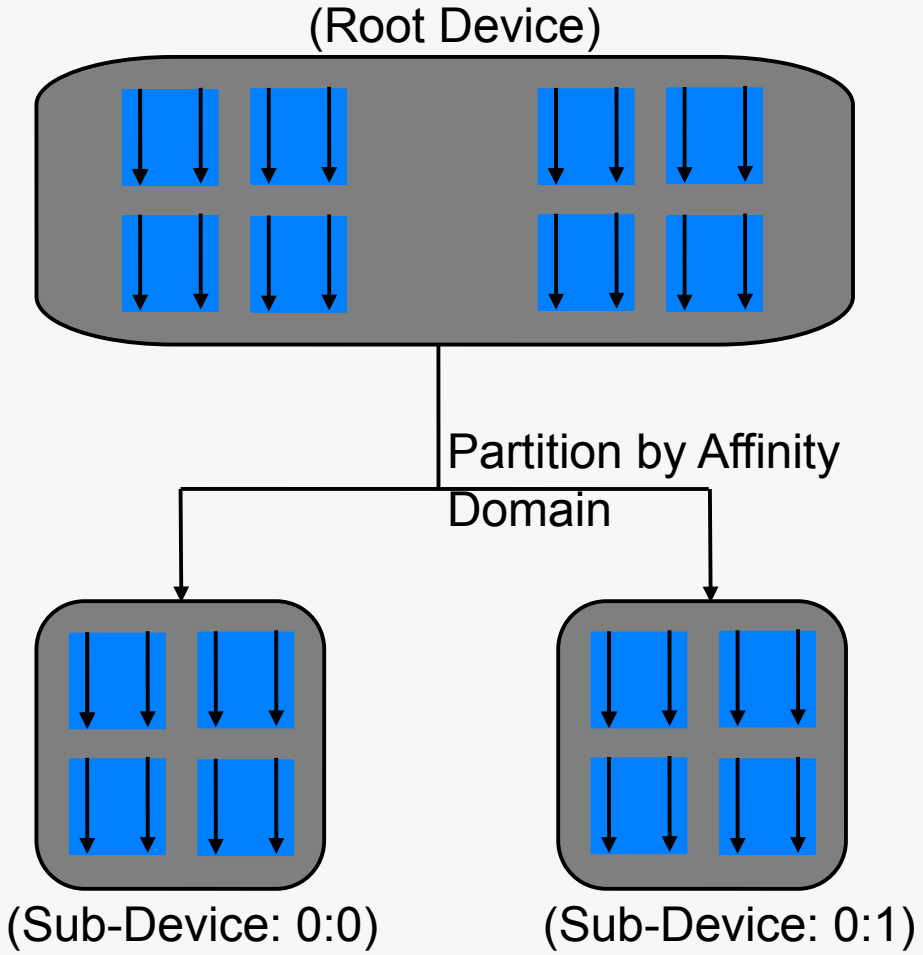
How to create “sub-devices”

- Two ways: runtime environment variables & via programming language heuristics

- Runtime environment variable:
  (to use whole root device, aka Implicit-scaling)
  `ZE_AFFINITY_MASK=#ROOT_DEVICE_ID`
  (i.e., `ZE_AFFINITY_MASK=0,1,...` for multi-GPUs)
- (to partition into sub-device, aka Explicit-scaling)
  `ZE_AFFINITY_MASK=#ROOT_DEVICE_ID:#SUB_DEVICE_ID`
- (i.e., `ZE_AFFINITY_MASK=0:0,0:1,1:0,1:1,…`)

- SYCL programming heuristics to parition “root” device:

```
// Get a handle to the default “device”
sycl::device root = sycl::device(sycl::default_selector{});
std::vector<sycl::device> devices;
if (has_numa_domains(root)) {
  // If root device has NUMA domains, split into sub-devices
  devices = root.create_sub_devices<by_affinity_domain>(numa);
}
else {
  // Otherwise, use the root device as-is
  devices = std::vector<sycl::device>{root};
}
```



Exploit SIMD with Sub-groups

- PVC supports two sub-group sizes: 16 and 32
- “Best” choice of sub-group size depends on many factors:
  - *Correctness*: Wise to use 32 during initial migration from CUDA
- *Register Pressure*: Smaller SIMD can help to alleviate register pressure
  - ✓ 128 registers * 512 bits / 16 lanes * 32 bits = 64 KiB / 512 bits = 128 scalar registers per lane
  - ✓ 128 registers * 512 bits / 32 lanes * 32 bits = 64 KiB / 1024 bits = 64 scalar registers per lane
- *Available Work*: Smaller SIMD ⇒ Less work required per EU thread
- *Instruction Overheads*: Larger SIMD ⇒ More compute per instruction
- Rule of Thumb: Aim for SIMD32, drop down to SIMD16 for complex code.

Exploit SIMD with Sub-groups

SIMD width can be set on per-kernel basis via kernel attribute "[[sycl::reqrd_sub_group_size()]]"

```
q.parallel_for(sycl::nd_range<1>{N}, sycl::nd_item<1> it)
[[sycl::reqd_sub_group_size(16)]] {
  // Get sub-group handle
  sycl::sub_group sg = it.get_sub_group();

  // Get work-item ID inside the sub-group
  // Migration from CUDA: (it.get_local_id() % 32)
  auto lane_id = sg.get_local_id();

  // Get sub-group ID within the work-group
  // Migration from CUDA: (it.get_local_id() / 32)
  auto sub_group_id = sg.get_group_id();
}
```

Attribute syntax will eventually be replaced by this extension:
https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/experimental/sycl_ext_oneapi_kernel_properties.asciidoc

Evaluate Register Spills vs Occupancy Tradeoff

- Request 256 registers per thread (“large GRF mode”) via environment variable:

```
export SYCL_PROGRAM_COMPILE_OPTIONS="-ze-opt-large-register-file"
```

- Trade-off between two factors:
  - *Register Pressure*: 2x registers per thread ⇒ Fewer spills/fills
  - *Occupancy*: 2x register footprint ⇒ 2x fewer threads ⇒ Exposed latency

- Rule of Thumb: Aim for 128 GRF, increase if you see spills.

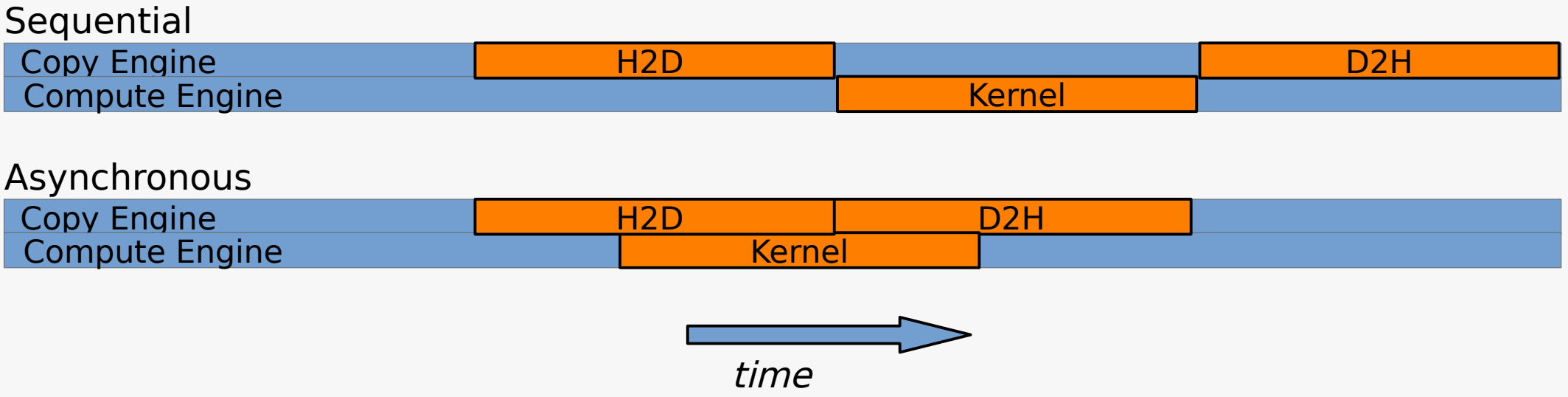
Avoid Indirect Register Access

- If algorithm requires a generic shuffle, it may be faster to use SLM

```
// sycl::select_from_group emulated via local memory
template <typename T>
inline T sub_group_exchange(sycl::sub_group sg, T value, int src) const
{
  T* slm_ptr = (T*) this->slm.get_pointer().get();
  const int offset = sg.get_group_id() * sg.get_max_local_range()[0];
  sycl::group_barrier(sg, sycl::memory_scope::sub_group);
  slm_ptr[offset + sg.get_local_id()] = value;
  sycl::group_barrier(sg, sycl::memory_scope::sub_group);
  return slm_ptr[offset + src];
}
```

Concurrent execution of work on GPU

- Execute several task (kernels, data-transfers, etc) in an out-of-order fashion



- Overlap computation with communication using SYCL & OpenMP

Concurreny via OpenMP directives

- OpenMP 4.5 standard now support asynchronous offloading
- `nowait` clause on target directives to achieve concurrent offloading

- Several host-threads launch “target-task” regions to GPU
- These task-regions may lead to asynchronous “concurrent” execution

```
#pragma omp parallel for
for (auto c: commands)
  #pragma omp target [...]
  {}
```

- A main-host thread launches several GPU “target-task” regions
- These target offload regions leads to asynchronous “concurrent” execution via “nowait” clause

```
for (auto c: commands)
  #pragma omp target [...] nowait
  {}

 #pragma omp taskwait
```

Concurreny via SYCL

- Concurrent execution can be possible via multiple “in-order SYCL queues” or a single Out-of-order SYCL queue

```
sycl::queue Q;
for (auto& c: commands)
  do_work(Q, c);
Q.wait();
```

- Default SYCL queue is out-of-order in nature

```
sycl::device D; sycl::context C(D);
std::vector<sycl::queue> in_orderQs;
for (auto _: commands)
  in_orderQs.push_back(sycl::queue(
C, D, sycl::property::queue::in_order{}));

// Submitings jobs
for (int i = 0; i < commands.size(); i++)
  do_work(in_orderQs[i], commands[i]);
for (auto &in_orderQ : in_orderQs)
  in_orderQ.wait();
```

- Submit work to several in-order SYCL queues. Similar to the practice with multiple CUDA streams
- Work submitted to each in-order SYCL queue executes independently

Query for “free” memory on GPU

- CUDA/HIP conveniently has `cudaMemGetInfo` API to query for total, free memory on GPU
- Recently DPC++ extension provides a functionality via a device descriptor: `sycl::ext::intel::info::device::free_memory`
- Motivation: Is there a way to obtain such information from Fortran OpenMP target offload

```
program main
  use iso_c_binding
  use omp_lib
  implicit none
  interface
     pure function get_free_memory_intel(d) result(n) bind(c)
       import c_int, c_size_t
       integer(kind=c_int), intent(in), value :: d
       integer(kind=c_size_t)                 :: n
     end function get_free_memory_intel
  end interface


  integer(kind=c_size_t) :: n
  integer(kind=c_int) :: device

  device = omp_get_num_devices() - 1

  n = get_free_memory_intel( device )
  print *, "n: ", n

end program main
```

*Note: Please set the env variable ZE_ENABLE_SYSMAN=1

```
#include <sycl/sycl.hpp>
#include <omp.h>
#include <map>
#include <iostream>

extern "C" {
  size_t get_free_memory_intel( int D )
  {
    omp_interop_t o = 0;
#pragma omp interop init(prefer_type("sycl"), targetsync: o) device(D)
    int err = -1;
    auto* sycl_device =  static_cast<sycl::device
   *>(omp_get_interop_ptr(o, omp_ipr_device, &err));
    assert (err >= 0 && "omp_get_interop_ptr(omp_ipr_device)");

    size_t freeMemory =
   sycl_device[0].get_info<sycl::ext::intel::info::device::free_memory>(
   );
    std::cout << "(root-dev) FreeMem (bytes): " << freeMemory <<
   std::endl;
#pragma omp interop destroy(o)

    return freeMemory;
  }
}
```

Query for “free” memory on GPU

```
program main
  use iso_c_binding
  use omp_lib
  implicit none
  interface
     pure function get_free_memory_intel(d) result(n) bind(c)
       import c_int, c_size_t
       integer(kind=c_int), intent(in), value :: d
       integer(kind=c_size_t)                 :: n
     end function get_free_memory_intel
  end interface


  integer(kind=c_size_t) :: n
  integer(kind=c_int) :: device

  device = omp_get_num_devices() - 1

  n = get_free_memory_intel( device )
  print *, "n: ", n

end program main
```

Listing1: main.F90

```
#include <sycl/sycl.hpp>
#include <omp.h>
#include <map>
#include <iostream>

extern "C" {
  size_t get_free_memory_intel( int D )
  {
    omp_interop_t o = 0;
#pragma omp interop init(prefer_type("sycl"), targetsync: o) device(D)
    int err = -1;
    auto* sycl_device =  static_cast<sycl::device *>(omp_get_interop_ptr(o,
   omp_ipr_device, &err));
    assert (err >= 0 && "omp_get_interop_ptr(omp_ipr_device)");

    size_t freeMemory =
   sycl_device[0].get_info<sycl::ext::intel::info::device::free_memory>();
    std::cout << "(root-dev) FreeMem (bytes): " << freeMemory << std::endl;
#pragma omp interop destroy(o)

    return freeMemory;
  }
}
```

Listing2: interop_sub.cpp

```
icpx -fsycl -fiopenmp -fopenmp-targets=spir64 interop_sub.cpp
ifx -i8 -fiopenmp -fopenmp-targets=spir64 -c main.F90
ifx -fsycl  -fiopenmp -fopenmp-targets=spir64 main.o interop_sub.o
```

- Language “interoperability” feature between OpenMP offload & SYCL/C++ provides several new capabilities

Language Interoperability: OpenMP vs SYCL

Your HPC applications is written in C++/OpenMP
Motivation: But you may want to be interfaced with SYCL

- ✓ Some SYCL API are more flexible than the OpenMP counterpart

- ✓ Some API only exist in SYCL
  - oneMKL APIs provide both an OpenMP and SYCL API, but SYCL APIs may provide enhanced functionality
  - oneDPL (CUDA's Thrust equivalent) with APIs in SYCL
  - You want to use SYCL to allocate memory (host, device, shared)

Source code :
https://github.com/argonne-lcf/HPC-Patterns/tree/main/sycl_omp_ze_interopt

Interoperability: OpenMP and SYCL

- ✓ Use `#pragma omp interop` to get "Native Handler" (OpenMP 5.1)
- ✓ Use the obtained handlers to create SYCL object (SYCL 2020)

```
ompDeviceId2Context.resize(omp_get_num_devices());
for (int D=0; D < omp_get_num_devices(); D++) {
     omp_interop_t o = 0;
     #pragma omp interop init(prefer_type("sycl"), targetsync: o) device(D)
     int err = -1;
     auto* sycl_context = static_cast<sycl::context *>(omp_get_interop_ptr(o, omp_ipr_device_context, &err));
     assert (err >= 0 && "omp_get_interop_ptr(omp_ipr_device_context)");
     auto* sycl_device =  static_cast<sycl::device *>(omp_get_interop_ptr(o, omp_ipr_device, &err));
     assert (err >= 0 && "omp_get_interop_ptr(omp_ipr_device)");
     #pragma omp interop destroy(o)

  •  // vector used to map a OpenMP device ID to a sycl context and device
     ompDeviceId2Context[D].sycl_context = sycl_context[0];
     ompDeviceId2Context[D].sycl_device = sycl_device[0];
 }
```

Note: It is important to ensure both OpenMP & SYCL uses the same "context" to ensure compatibility. Hence we explicitly use both SYCL device & SYCL context objects to create a SYCL queue. Though SYCL queues can be created with SYCL device object alone.

Source code :
https://github.com/argonne-lcf/HPC-Patterns/blob/main/sycl_omp_ze_interopt/interop_omp_ze_sycl.cpp

Interoperability: Kokkos & OpenMP

- Application written in OpenMP target offload interfaces with Kokkos (SYCL backend for device)

```
   ompDeviceId2Context.resize(omp_get_num_devices());
   for (int D=0; D < omp_get_num_devices(); D++) {
        omp_interop_t o = 0;
        #pragma omp interop init(prefer_type("sycl"), targetsync: o) device(D)
        int err = -1;
        auto* sycl_context = static_cast<sycl::context *>(omp_get_interop_ptr(o, omp_ipr_device_context, &err));
        assert (err >= 0 && "omp_get_interop_ptr(omp_ipr_device_context)");
        auto* sycl_device =  static_cast<sycl::device *>(omp_get_interop_ptr(o, omp_ipr_device, &err));
        assert (err >= 0 && "omp_get_interop_ptr(omp_ipr_device)");
        #pragma omp interop destroy(o)

    •   // vector used to map a OpenMP device ID to a sycl context and device
        ompDeviceId2Context[D].sycl_context = sycl_context[0];
        ompDeviceId2Context[D].sycl_device = sycl_device[0];
    }

  #ifdef KOKKOS_IMPL_SYCL_USE_IN_ORDER_QUEUES
   initialize( sycl::queue{ompDeviceId2Context[D].sycl_context, ompDeviceId2Context[D].sycl_device,
  sycl::property::queue::in_order{}} );
#else
   initialize(sycl::queue{ompDeviceId2Context[D].sycl_context, ompDeviceId2Context[D].sycl_device} );
#endif
```

Interoperability: oneDPL library & OpenMP

- oneDPL library provides a rich set of algorithms covering C++ Standard Template Library (STL), Parallel STL algorithms to be offloaded to GPUs via SYCL kernels (underneath)

```
#pragma omp target enter data map(to: data[0:N])
T* omp_data_gpu;
#pragma omp target data use_device_ptr(data) { data_gpu = data }
sycl::queue q = get_interopt_queue(); //Pseudo code for OpenMP Interop

//SYCL parallel STL algorithm using an OpenMP device pointer
std::sort(oneapi::dpl::execution::make_device_policy(q), omp_data_gpu, omp_data_gpu + N);
```

Interoperability: Memory pointers from SYCL & OpenMP

```
sycl::queue Q = get_intereopt_queue(); // Where the magic happens
T *ompMem = (T*) malloc(N*sizeof(T));
T *syclMem = sycl::malloc_device<T>(N,Q);
```

- OpenMP target using memory managed via SYCL APIs

```
#pragma omp target is_device_ptr(syclMem) map(from:ompMem[0:N])
for (size_t i=0 ; i < N; i++)
  ompMem[i] = syclMem[i];
```

- SYCL APIs using memory managed by OpenMP target regions

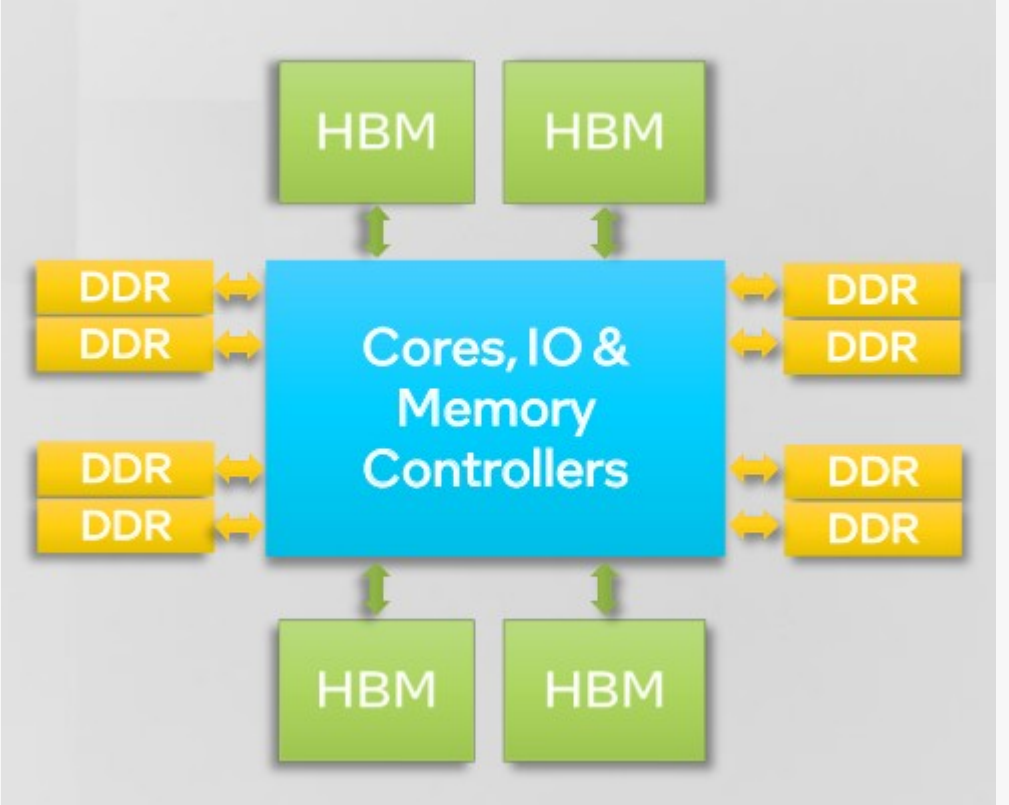
```
T* ompMem_gpu;
#pragma omp target enter data map(to:ompMem[0:N])
#pragma omp target data use_device_ptr(ompMem) { ompMem_gpu = ompMem }
Q.copy<T>(cpuMem, ompMem_gpu, N).wait();
```

Using DDR & HBM memory on Sapphire Rapids (SPR) CPU

Targeting high-bandwidth apps in HPC and AI

- ✓ Four HBM2e stacks on package
- ✓ 64 GB of total HBM capacity per socket
- ✓ 8 channels of DDR5

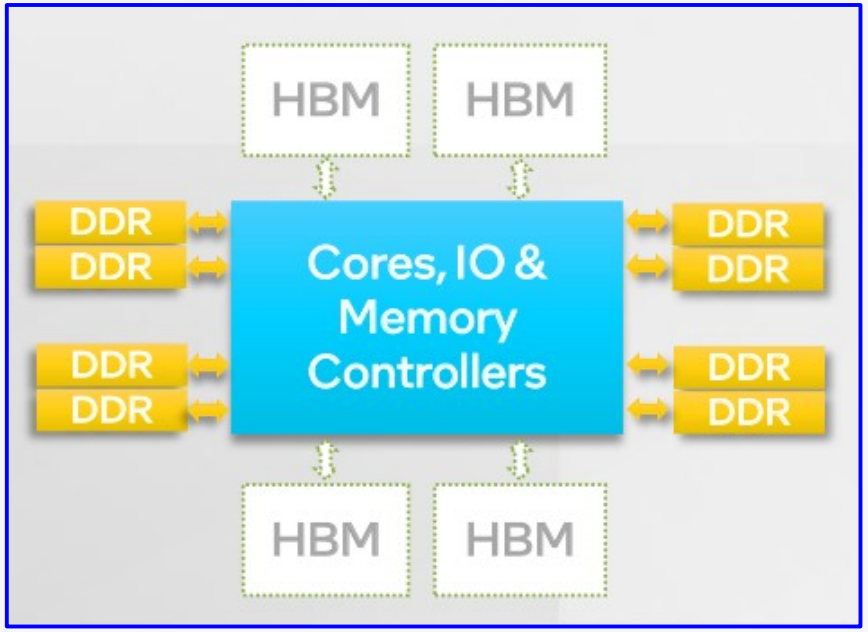
XEON DESCRIPTION	
Vector Extension	AVX-512
Threads (#)	2
Total HBM Memory (GB)	64
Peak HBM Memory BW (TB/s)	1.43
Total DDR5 4400 Memory (GB)	512
Peak DDR5 4400 Memory BW (TB/s)	0.28

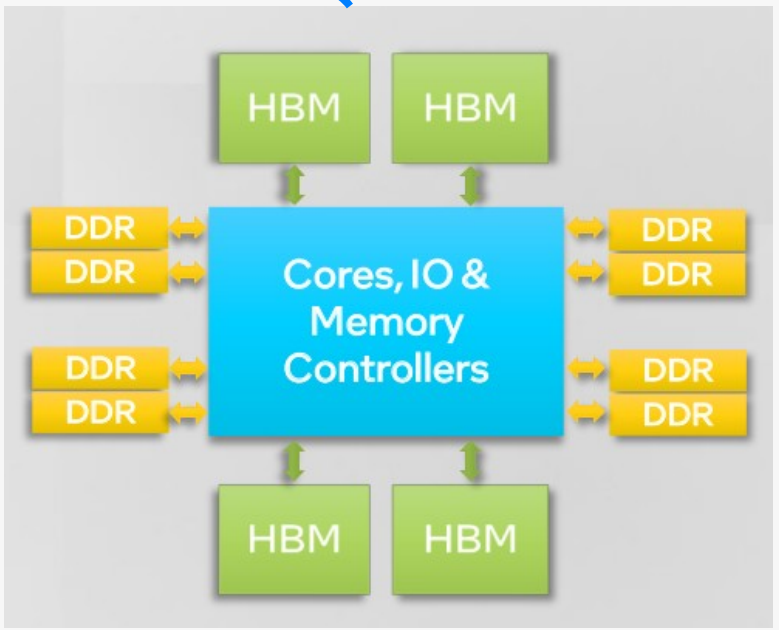


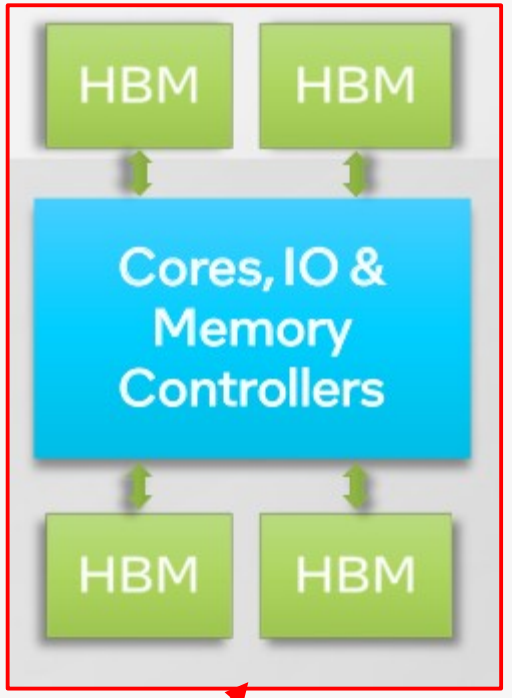
Schematic of Aurora Sapphire Rapids CPU: Configuration with DDR5 & HBM memory

Configuring memory on Sapphire Rapids

Configuration to use only (large, slow) DDR5 memory







Configuration to use only (small, fast) HBM2e memory

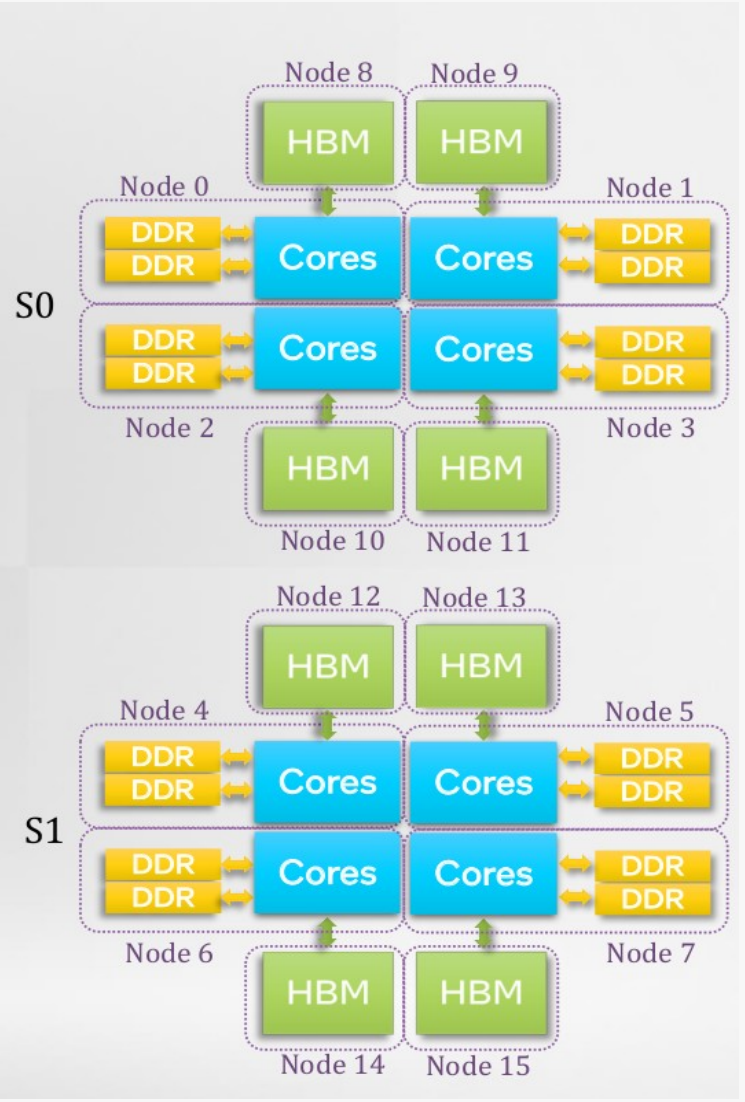
How to use HBM2e memory instead of DDR

- ✓ Applications have visibility to both DDR and HBM
- ✓ By default, applications use DDR memory

• Use numactl to place applications in HBM
- `numactl -m 8-15 mpirun –n 8 ./app`
- `numactl --preferred-many 8-15 mpirun –n 8 ./app`

• Alternatively use Intel MPI `I_MPI_HBW_POLICY` to place apps in HBM
- `mpirun –genv I_MPI_HBW_POLICY hbw_bind -n 8 ./app`
- `mpirun –genv I_MPI_HBW_POLICY hbw_preferred -n 8 ./app`

• Heap structures can be placed in HBM with program modification
- Either with `memkind` library or using `libnuma` APIs



Schematic of Aurora Sapphire Rapids CPU with 2 sockets: Numa configuration with DDR & HBM memory

How to use both DDR & HBM2e memory

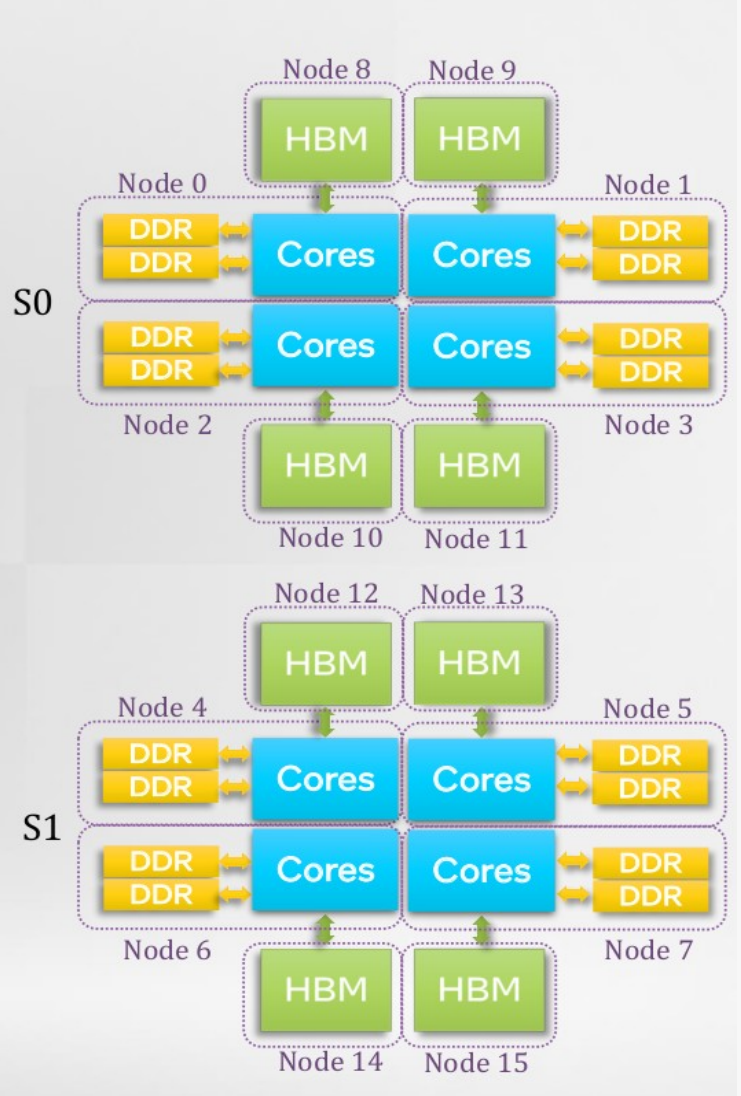
- Applications such as NWChemEx has explored the use of both DDR & HBM2e memory on Intel Sapphire Rapids CPUs
- Memory numa domains for DDR & HBM2e can be identified by their numa-node IDs and with `numactl -H`
- No code changes are required for DDR
- To target all the (de)allocations for HBM, use memkind library APIs.

```
#include <hbwmalloc.h>

if(hbw_check_available() == 0) { // returns zero if hbw_malloc is availiable.

  // To allocate
  hbw_set_policy(HBW_POLICY_BIND);
  Ptr = hbw_malloc(size_in_bytes);

  // To deallocate
  hbm_free(Ptr);
}
```

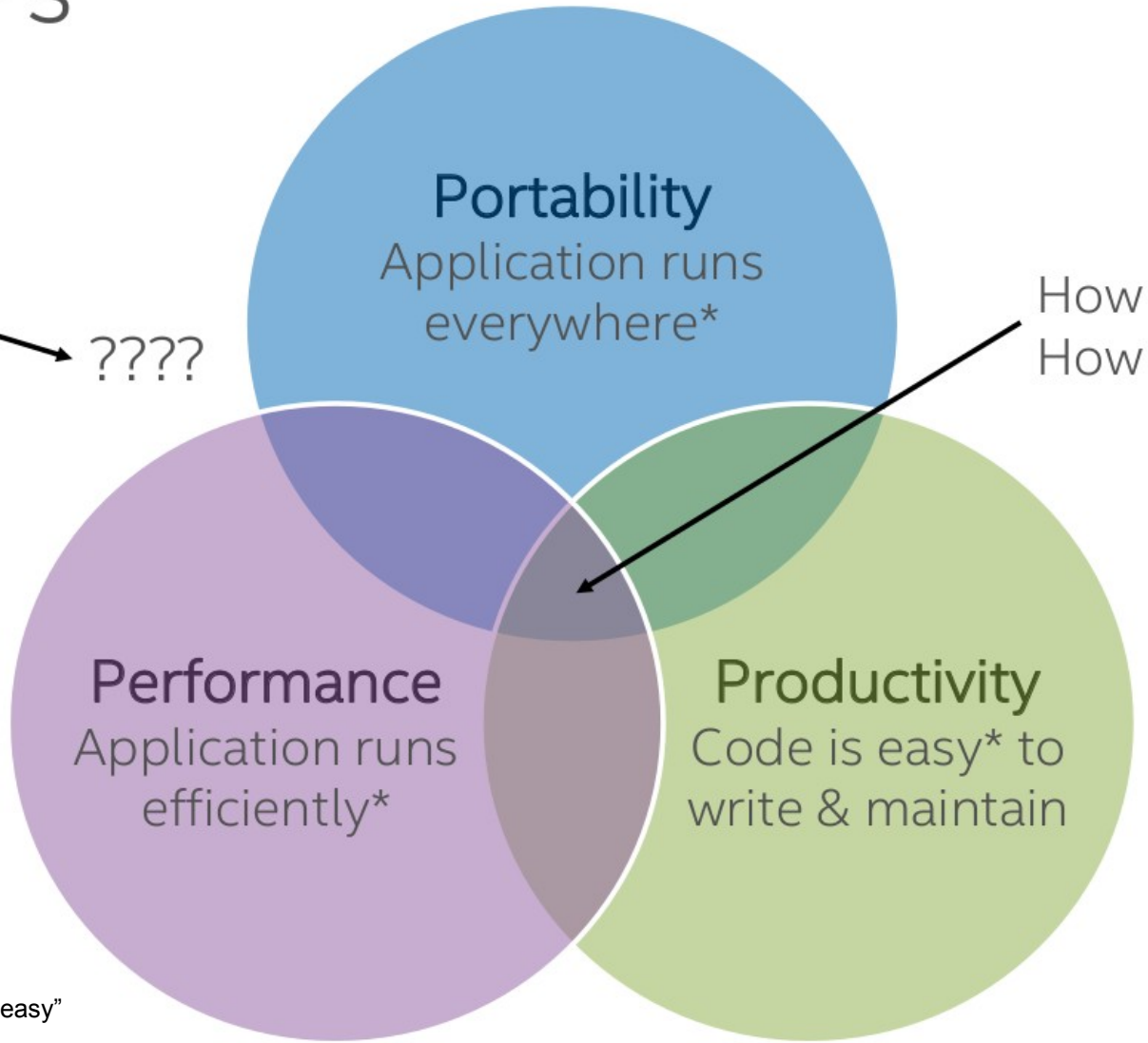


Schematic of Aurora Sapphire Rapids CPU with 2 sockets: Numa configuration with DDR & HBM memory

Performance Portability

The "Three Ps"

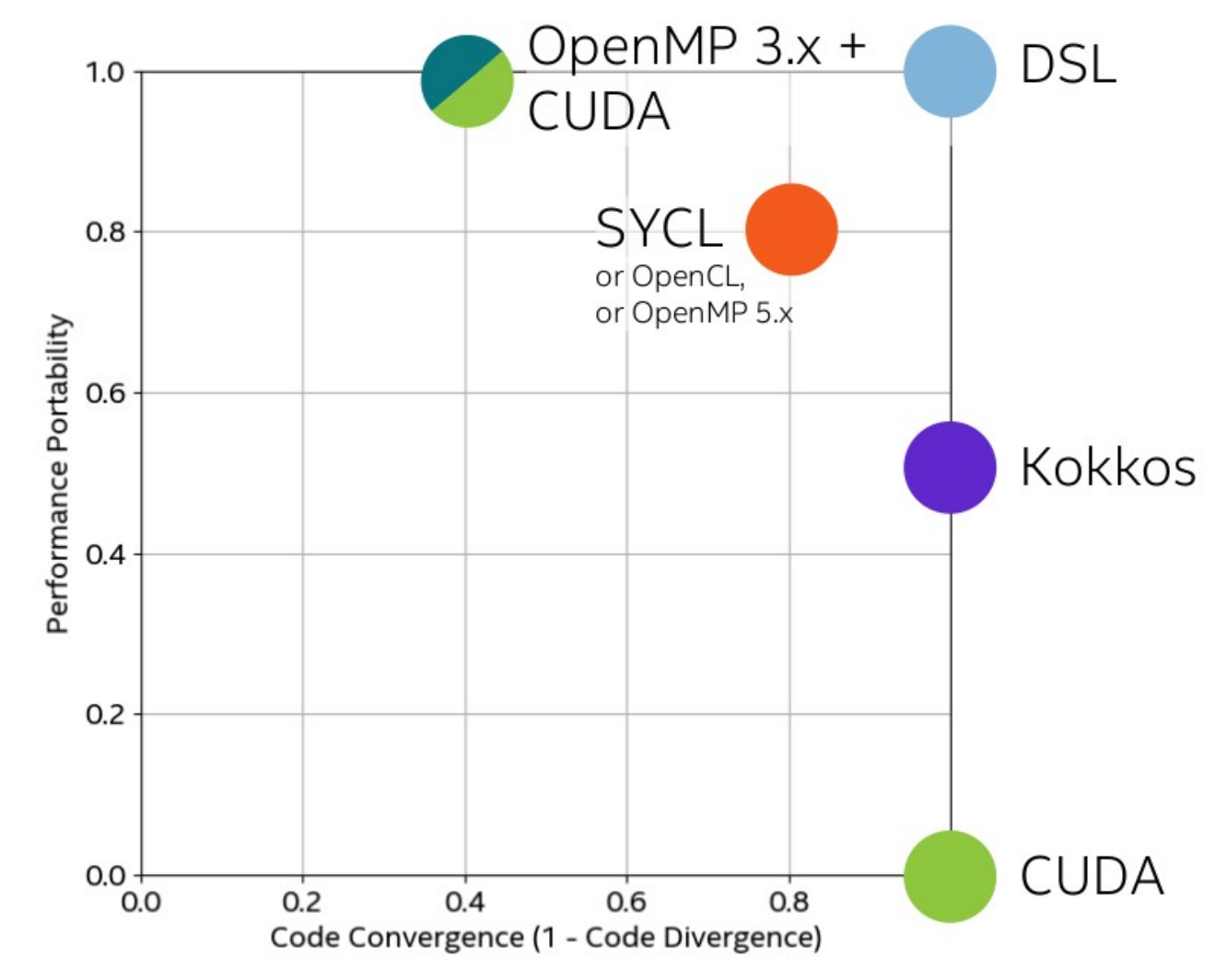
????



How do applications get here?
How can tools help?

* For some definition of "everywhere", "efficiently" and "easy"

Performance Portability



SC22 IXPUG BoF: Performance Portability in a Heterogeneous World – Pipe Dream? Are We Dreaming the Same Dream?

Tools to Analyze Performance of SYCL Applications

- ***Intel® VTune™ Profiler***. - Analyze the performance of an application. Identify the most time-consuming functions in the application, whether the application is CPU- or GPU-bound, how effectively it offloads code to the GPU

- ***SYCL_PI_TRACE=2*** environment variable. Provides a trace of all PI calls made with arguments and returned values.

- ***Dump of compiler-generated assembly for the device,*** 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
```

Tools to Analyze Performance of SYCL Applications

- ***Intel® VTune™ Profiler***. - Analyze the performance of an application. Identify the most time-consuming functions in the application, whether the application is CPU- or GPU-bound, how effectively it offloads code to the GPU

- ***SYCL_PI_TRACE=2*** environment variable. Provides a trace of all PI calls made with arguments and returned values.

- ***Dump of compiler-generated assembly for the device,*** 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
```

Tools to Analyze Performance of OpenMP Applications

- ***Intel® VTune™ Profiler***. - Analyze the performance of an application. Identify the most time-consuming functions in the application, whether the application is CPU- or GPU-bound, how effectively it offloads code to the GPU

- *LIBOMPTARGET_DEBUG=1* environment variable. LIBOMPTARGET_DEBUG controls whether or not debugging information from libomptarget.so will be displayed

- *LIBOMPTARGET_PLUGIN_PROFILE=T* environment variable. LIBOMPTARGET_PROFILE allows libomptarget.so to generate time profile output.

- ***Dump of compiler-generated assembly for the device,*** 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
```

Future Events for Aurora

https://www.alcf.anl.gov/events/optimizing-workloads-aurora-and-sunspot-examples-sycl-application

Webinar: Optimizing Workloads on Aurora and Sunspot: Examples with SYCL Application

Acknowledgements

- Early-users of Aurora Early Science Program (ESP) & Exascale Computing Project
- ALCF Performance Engineering Team, Operations, Catalysts
- Intel Center of Excellence at Argonne
- Open-source oneAPI, OpenMP & SYCL community

Scott Parker (ALCF)
Kevin Harms (ALCF)
Vitali Morozov (ALCF)
Servesh Muralidharan (ALCF)
Thomas Applencourt (ALCF)
Colleen Bertoni (ALCF)
Victor Anisimov (ALCF)
Yasaman Ghadar (ALCF)
JaeHyuk Kwack (ALCF)
Nevin Liber (ALCF)
Brian Homerding (ALCF)
Kris Rowe (ALCF)
Brice Videau (ALCF)

Michael D'Mello (Intel CoE)
Renzo Bustamante (Intel CoE)
Dahai Guo (Intel CoE)
Varsha Madananth (Intel CoE)
Brian Holland (Intel CoE)
Patrick Steinbrecher (Intel CoE)
John Pennycook (Intel)
Jaime Artega (Intel)

& Many more Intel, HPE engineers