Intel GPU Optimization Guide

Memory Allocation and Memory Movement

Learn about Optimizing Memory Allocation and Memory Movement

rakshith.Krishnappa@intel.com

Agenda

- Access Intel Data Center MAX GPUs using Intel Developer Cloud
- Intel GPU Optimization using SYCL
  - Unified Shared Memory vs Buffer Memory Model
  - Optimizing techniques for Buffer Memory Model
  - Optimizing techniques for Unified Shared Memory Model

Learning Objectives

- Access Intel Data Center GPU MAX using Intel Developer Cloud
- Understand Buffer Memory Model vs Unified Shared Memory Model
- Optimize memory copies by using proper Accessor permissions and properties
- Optimize memory copies by overlapping copies and executions.

SYCL Memory Model

SYCL offers several choices for managing memory on the device.

- Buffer Memory Model - 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.
- Unified Shared Memory Model (USM) - USM allows reading and writing of data with conventional pointers. You have choice of explicitly moving memory or let the runtime move memory implicitly.

SYCL Buffers Method

Buffer Memory Model with buffer created for data, and accessor for accessing data on device, and host_accessor to copy back the data from device to host.

Host memory setup → `int data[N];`

Host can initialize → `for(int i=0;i<N;i++) data[i] = 10;`

Create buffer → `sycl::buffer buf(data, sycl::range<1>(N));`

Create accessor → `sycl::accessor A(buf, h);`

Device can modify → `A[i] += 1;`

Host Accessor → `sycl::host_accessor ha(buf, sycl::read_only);`

Host has output → `for(int i=0;i<N;i++) std::cout << data[i] << " ";`

```
sycl::queue q;
int data[N];
for(int i=0;i<N;i++) data[i] = 10;

    sycl::buffer buf(data, sycl::range<1>(N));
    q.submit([&] (handler &h){
        sycl::accessor A(buf, h);
        h.parallel_for(N, [=](auto i){
            A[i] += 1;
        });
    });
sycl::host_accessor ha(buf, sycl::read_only);
for(int i=0;i<N;i++) std::cout << data[i] << " ";
```

Unified Shared Memory – Implicit Movement

malloc_shared enables accessing memory on the host and device with same pointer reference, memory movement happens implicitly.

```
sycl::queue q;
auto data =  sycl::malloc_shared<int>(N, q);
for(int i=0;i<N;i++) data[i] = 10;
q.parallel_for(N, [=](auto i){
        data[i] += 1;
}).wait();
for(int i=0;i<N;i++) std::cout << data[i] << " ";
sycl::free(data, q);
```

Setup Unified Shared Memory → `auto data = sycl::malloc_shared<int>(N, q);`

Host can initialize → `for(int i=0;i<N;i++) data[i] = 10;`

Device can modify → `data[i] += 1;`

Host has output → `for(int i=0;i<N;i++) std::cout << data[i] << " ";`

Unified Shared Memory – Explicit Movement

malloc_device enables allocating memory on the device, and memory should be copied explicitly between host and device using memcpy method.

Unified Shared Memory Setup

Host to Device copy

Device can modify

Device to Host copy

```
sycl::queue q;
int data[N];
for(int i=0;i<N;i++) data[i] = 10;
auto data_device =  sycl::malloc_device<int>(N, q);
q.memcpy(data_device, data, sizeof(int) * N).wait();
q.parallel_for(N, [=](auto i){
        data_device[i] += 1;
}).wait();
q.memcpy(data, data_device, sizeof(int) * N).wait();
for(int i=0;i<N;i++) std::cout << data[i] << " ";
sycl::free(data, q);
```

Which SYCL Memory Model should I use?

SYCL Buffers are powerful and elegant

- Use if the abstraction applies cleanly in your application, and/or buffers aren't disruptive to your development
- Useful when computations use with 2 or 3-dimensional data representations

USM provides a familiar pointer-based C++ interface

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

Hand-on Workshop

- Intel GPU Optimization using SYCL
  - Buffers vs USM
  - Optimization Techniques for Buffers
  - Optimization Techniques for USM

Optimization Techniques for Buffers

- Accessor access modes are very important to set to avoid unnecessary copies.
- Use sycl::no_init, if output buffer is write_only and initial data is not required in kernel computation.
- Avoid declaring buffers in loops.
- Avoid copying data back and forth between host and device.

Optimization Techniques for USM

- If possible, chunk data to allocate, copy and submit kernels for multiple chucks to overlap copies and computations.
- Use sycl::malloc_host for allocation on hosts and then copy to device allocations.
- Copy back only sections of data that is modified by device.

Resources

- SYCL Essentials training modules:
  - https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/DPC%2B%2B/Jupyter/oneapi-essentials-training
- Intel GPU Optimization Guide:
  - https://www.intel.com/content/www/us/en/develop/documentation/oneapi-gpu-optimization-guide/top.html
- SYCL Code Samples:
  - https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/DPC%2B%2B

Resources

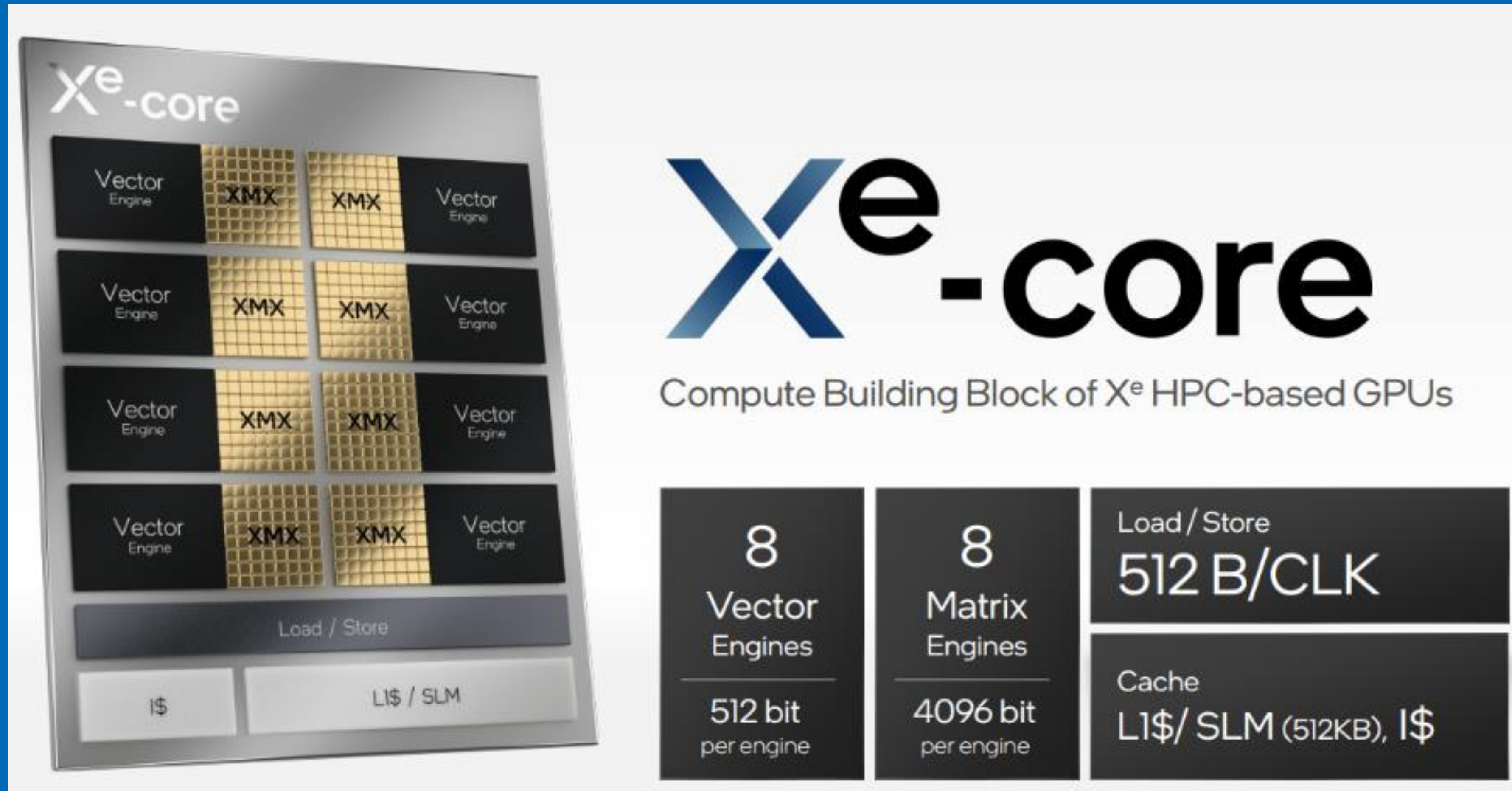
- Download and Install Intel oneAPI Compiler, Libraries and Tools:
  - https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit.html
- Build open source SYCL compiler:
  - https://github.com/intel/llvm
- SYCL Specification:
  - https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf

Intel Data Center GPU MAX Series

Intel's highest performing, highest density, general-purpose discrete GPU, which packs over 100 billion transistors into one package

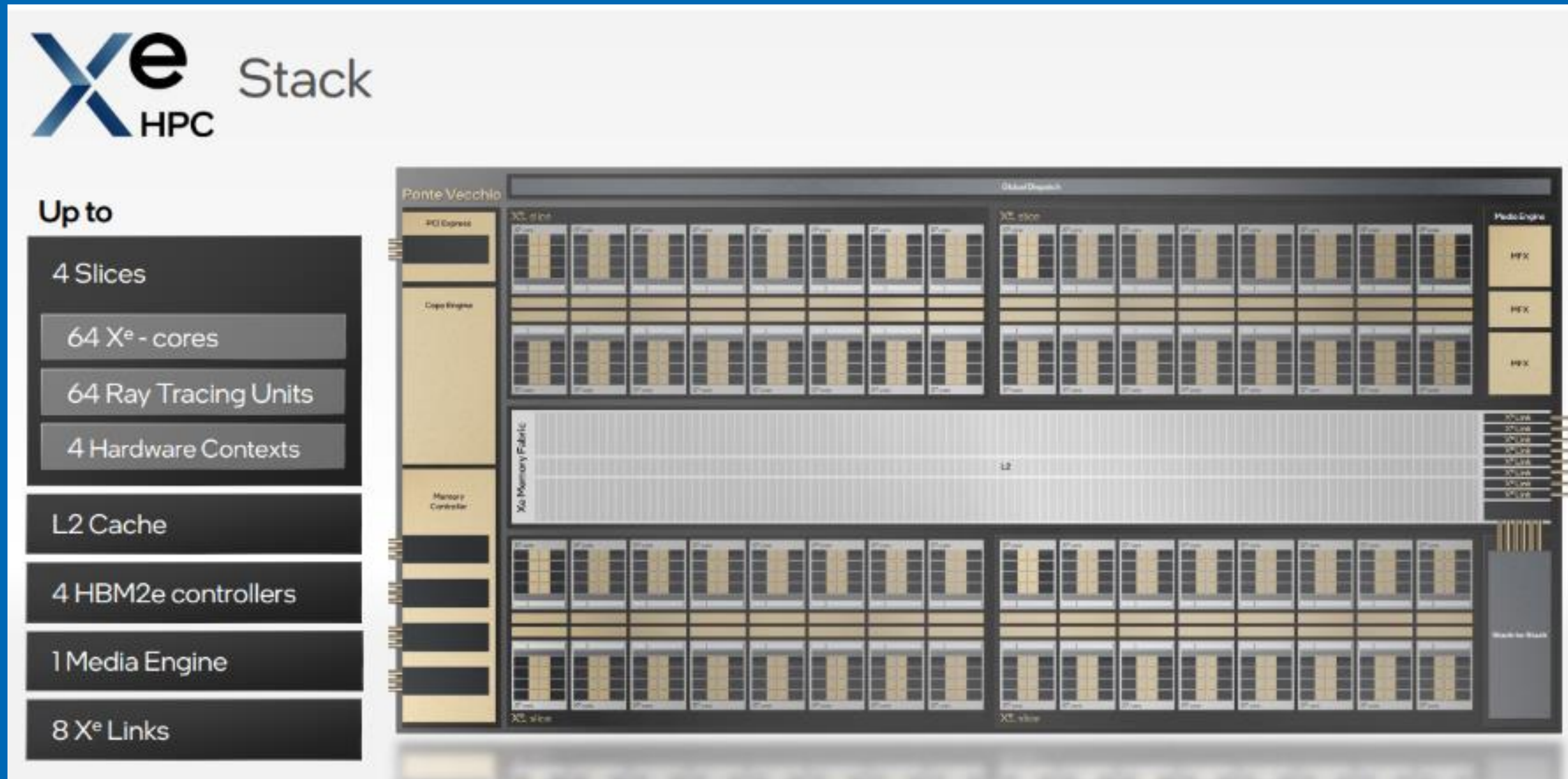
Xe Core

Building block of GPU with 8 vector engines, 8 matrix engines, SLM/L1 Cache



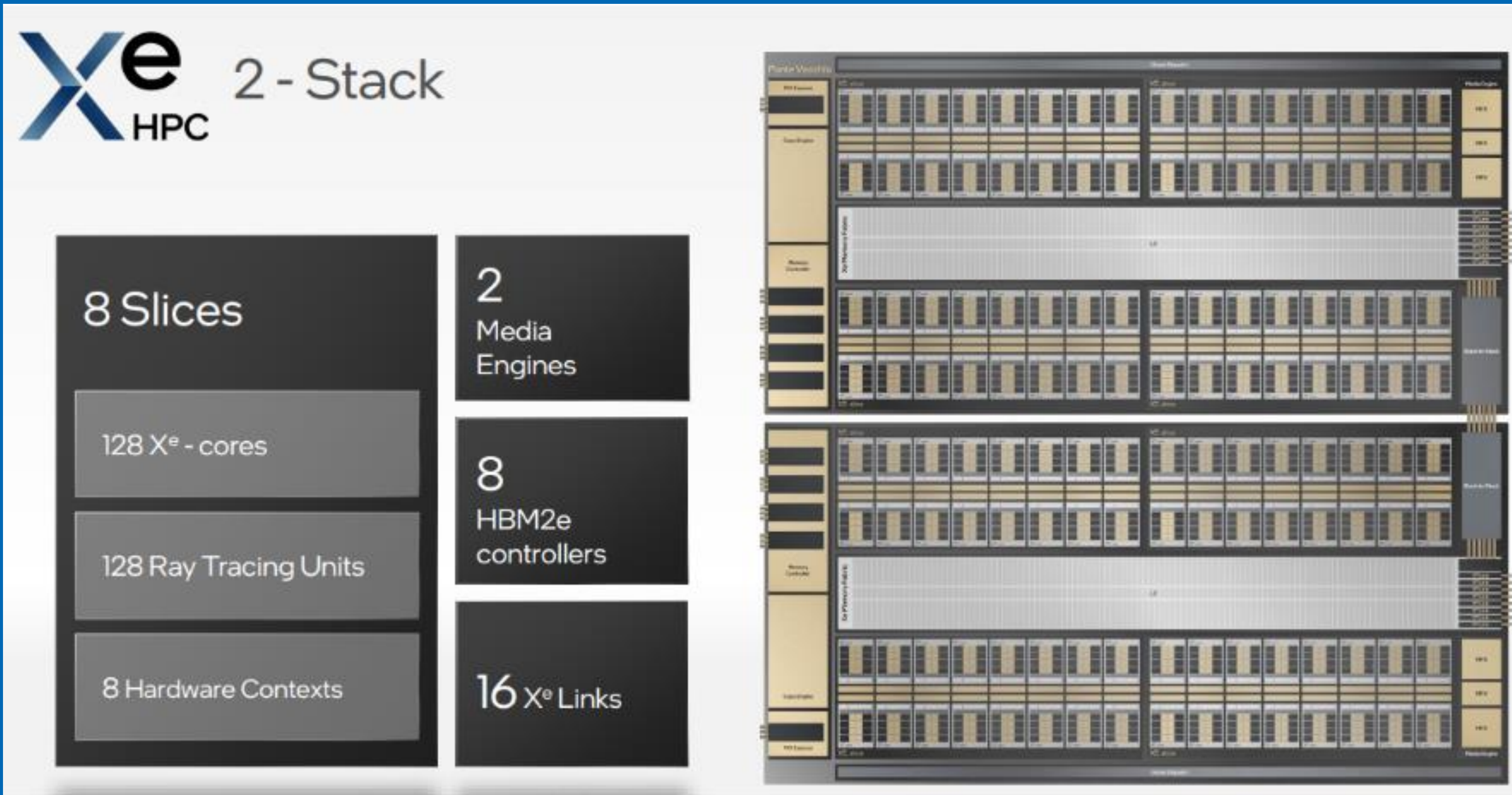
Xe Stack

Up to 4 Xe-Slices, Media Engine, L2 Cache, Memory Controllers, Xe-Links



2 Xe Stack

GPU with multiple Xe-Stack



Xe-HPC Architecture

- The Compute building block of the Xe HPC-based GPU is the Xe-Core consisting of 8 vector engines.
  - (Vector Engine formerly referred to as Execution-Units/EU, Xe-Core formerly referred to as Sub-Slice in Gen9/Gen11 Graphics HW)
- 16 Xe-Cores with a hardware context make up a Xe-Slice
- Up to 4 Xe-Slice makes Xe-Stack (with up to 64 Xe-Cores)
- 1 or more Xe-Stacks can be present in GPU

Intel Data Center GPU MAX Series

[Intel® Data Center GPU Max Series Overview]

Available today:

- Intel® Data Center GPU Max 1100 (56 Xe Cores)
- Intel® Data Center GPU Max 1550 (128 Xe Cores)

Intel Developer Cloud

Intel® Developer Cloud is a service platform for developing and running workloads in Intel®-optimized deployment environments with the latest Intel® processors, Intel® GPUs and performance-optimized software stacks.

- Sign-up for free
- cloud.intel.com

Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

intel®