

CPP-Summit

谢丙堃

《现代C++语言核心特性解析》作者

SYCL 编码实践 与编译设计浅析

议程

1

SYCL & DPC++ 基础介绍

异构计算的背景, DPC++ 的优势

2

SYCL & DPC++ 编程基础

理解DPC++中设备、队列、统一共享内存

3

SYCL & DPC++ 编译设计简介

理解DPC++编译设备代码的方法

4

DevCloud中学习SYCL & DPC++

介绍使用DevCloud的方法

01

SYCL & DPC++ 基础介绍

计算时代的变化

CPU 单核
单线程



CPU 多核
多线程



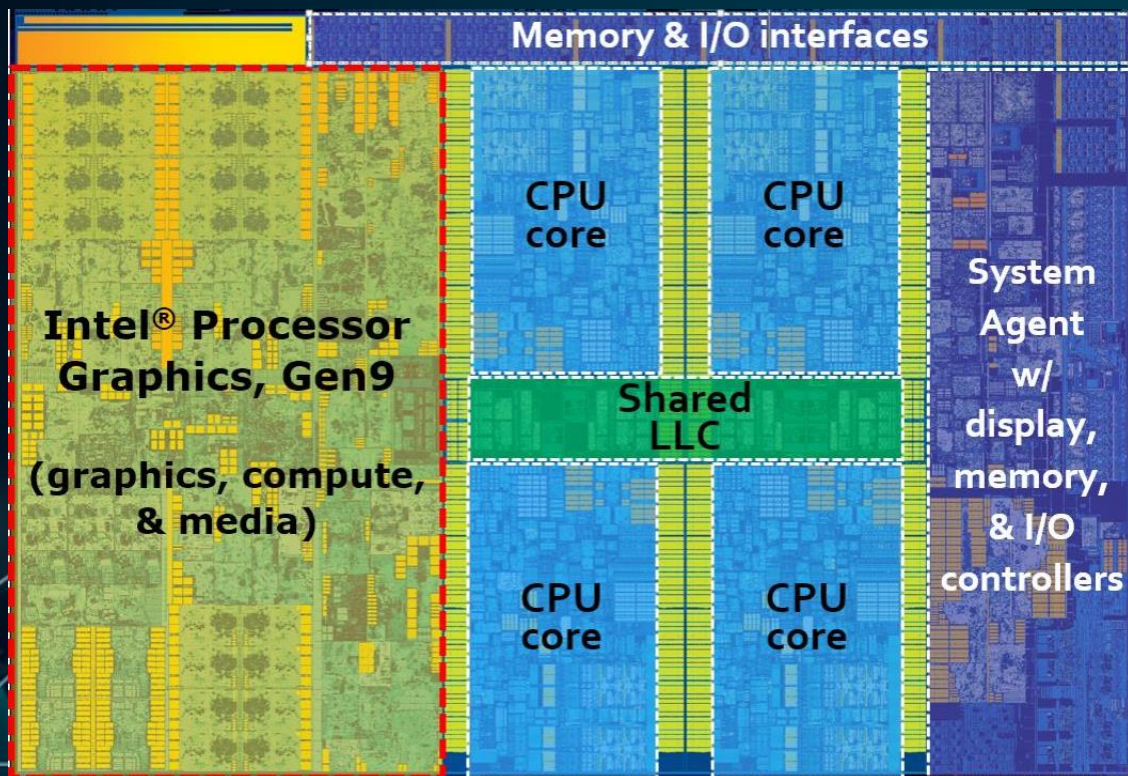
多种计算芯片
异构计算

什么是异构计算

- 异构计算指使用一种以上处理器或内核的系统。
 - 通过添加不同的协处理器；
 - 集成专门的处理功能来处理特定任务。
- 这些系统的性能或能效可显著提升。

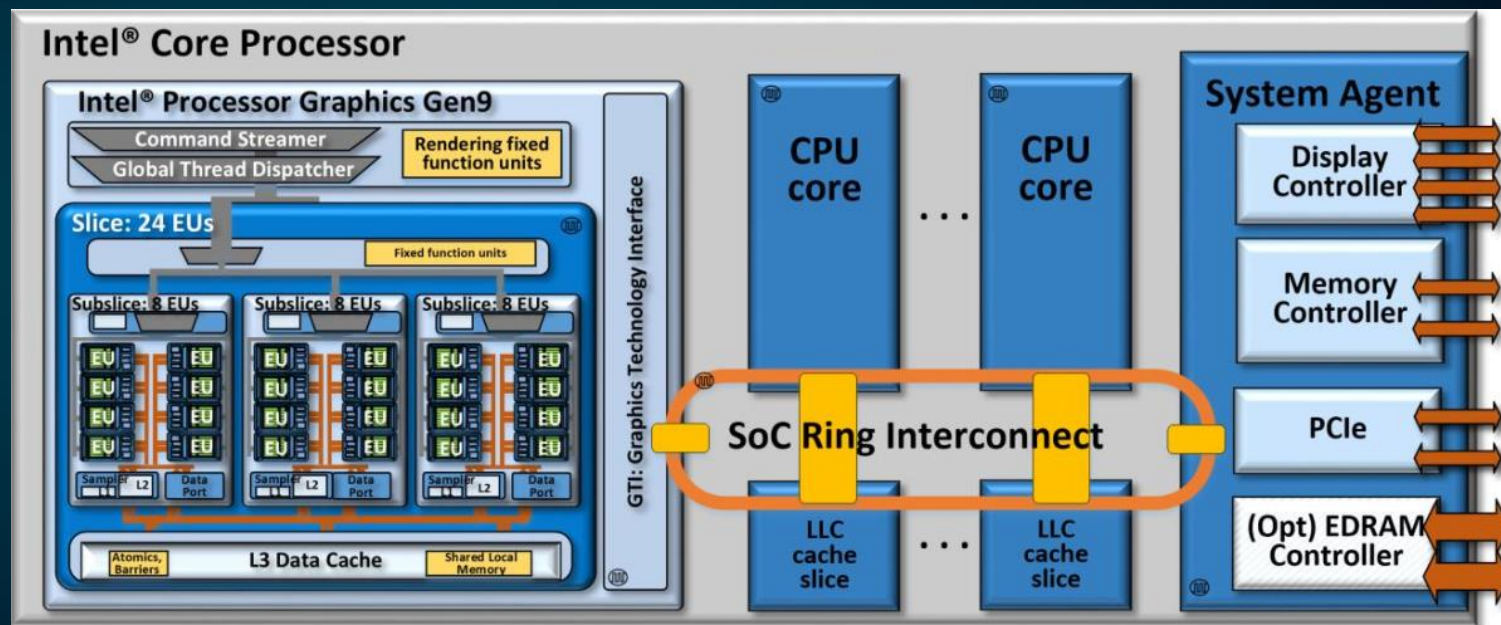
现代编程任务中的挑战

- 英特尔 酷睿 i7 处理器 6700K



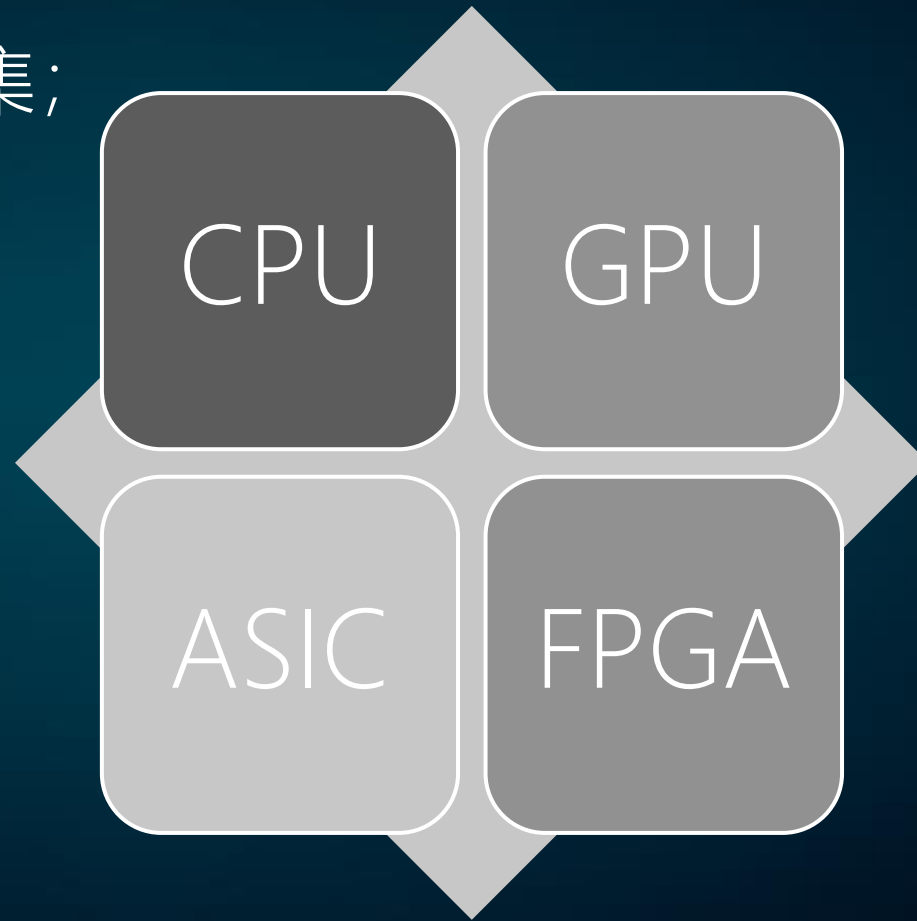
现代编程任务中的挑战

- Intel HD Graphics 530
 - Shading Units 192
 - TMUs 24
 - ROPs 3
 - Execution Units 24



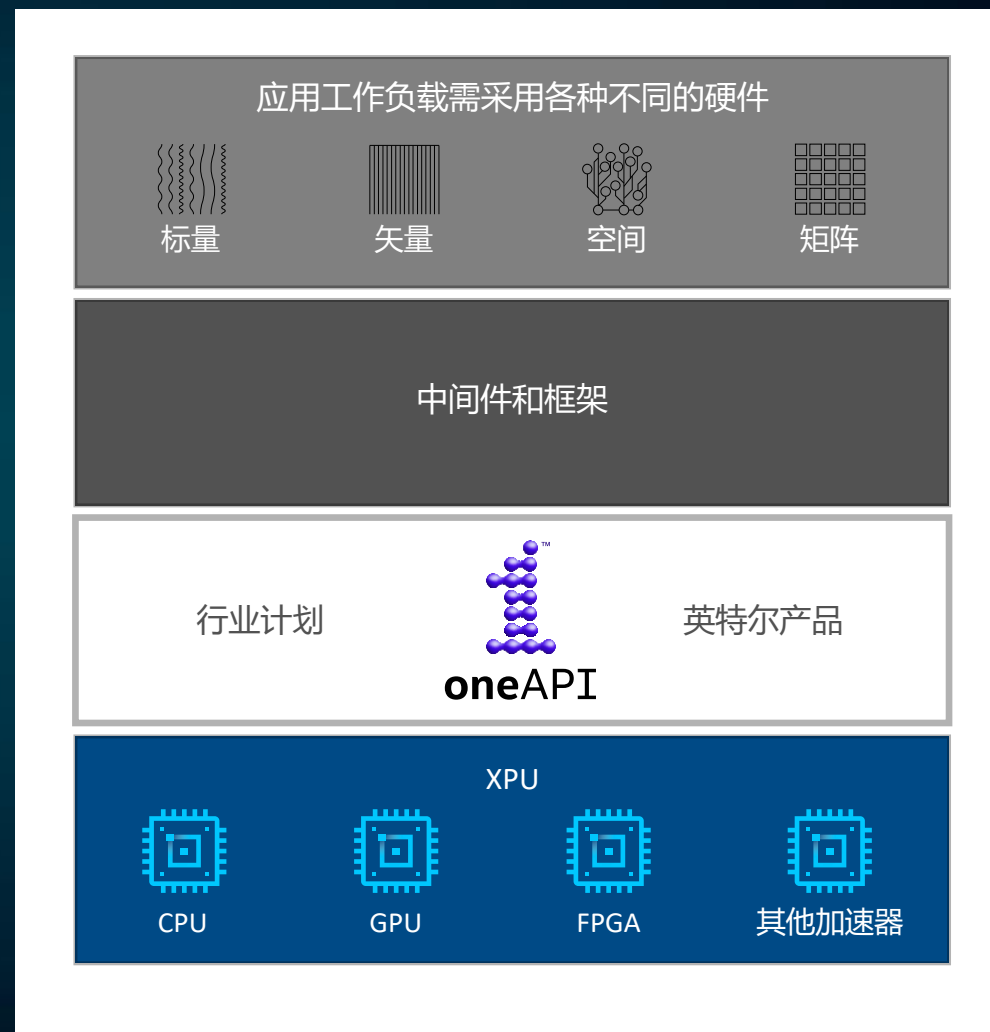
现代编程任务中的挑战

- 每种处理器拥有不同的指令集；
- 使用不同的编程方法和规范；
- 需要不同的工具链。



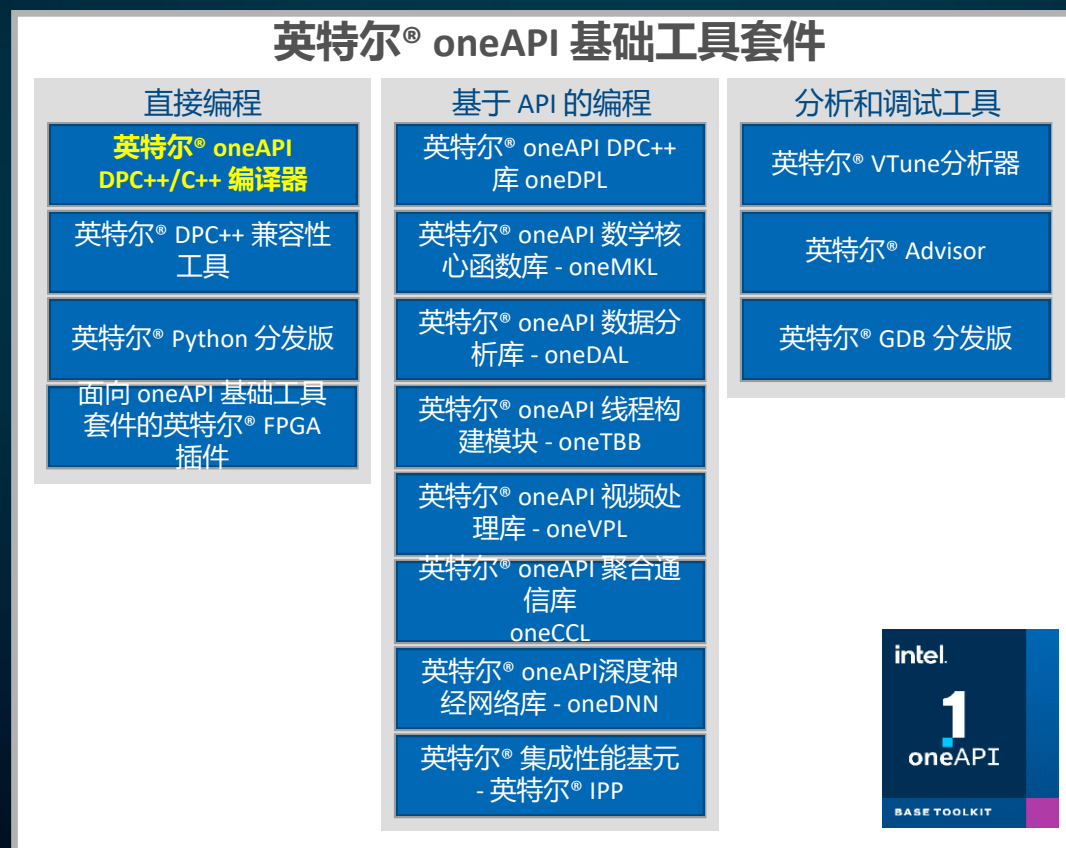
oneAPI - 用于多种架构编程模型

- oneAPI 是 Intel 为了推动加速计算的新时代，摆脱专有编程模型的经济和技术负担而推出的一套开发框架。
- 硬件和框架直接的抽象层
 - 让应用开发者不必担心硬件架构
 - 专注于任务本身
 - 兼容性、移植性强
- 开发套件
 - 软件工具众多
 - 适用范围广
 - HPC IOT AI ...



oneAPI – SYCL & DPC++

- DPC++是Intel oneAPI开发套件中的一个组件。



DPC++ (Data Parallel C++)

- DPC++是SYCL标准的一种实现：
 - 使用标准现代C++ (C++17标准)；
 - 集成了SYCL标准，支持数据并行性和异构编程；
 - 扩展SYCL标准：增强效率，提升性能；
 - 基于LLVM的开源编译器。

02

SYCL & DPC++ 编程基础

一个最简单的例子

单一源代码：host和device代码混合编程。

使用标准C++代码。

parallel_for: 使用加速设备并行执行lambda表达式计算

queue: 使用DPC++队列来调度和执行设备上的命令队列

malloc_shared: 使用Unified Shared Memory (USM) 来进行数据管理

```

#include <CL/sycl.hpp>
constexpr int N=16;
using namespace sycl;
int main() {
    queue q;
    int *data = malloc_shared<int>(N, q);
    q.parallel_for(N, [=](auto i) {
        data[i] = i;
    }).wait();
    for (int i=0; i<N; i++) std::cout << data[i] << "\n";
    free(data, q);
    return 0;
}
    
```

free: 使用sycl::free释放分配的USM

Host code

Device code

Host code

队列(queue)

- 队列用于提交命令组(command group)到SYCL 运行时执行;
 - 成员函数submit: 将命令组功能对象提交到队列, 以便安排在设备上执行;
 - 成语函数parallel_for是submit的一种简化写法。
- 队列是一种将工作(work)提交到设备的机制;
- 一个队列映射(map)到一个设备, 多个队列(multiple queue)可以映射到同一设备。

使用submit提交命令

```
#include <CL/sycl.hpp>
constexpr int N=16;
using namespace sycl;
int main() {
    queue q;
    int *data = malloc_shared<int>(N, q);
    q.submit([&] (handler &h){
        h.parallel_for(N, [=](auto i) {
            data[i] = i;
        });
    }).wait();

    for (int i=0; i<N; i++) std::cout << data[i] << "\n";
    free(data, q);
    return 0;
}
```

handler: 命令组句柄对象,
提供一系列的调度函数

设备选择

- 队列可以通过设备选择器(device selector)选择指定设备:

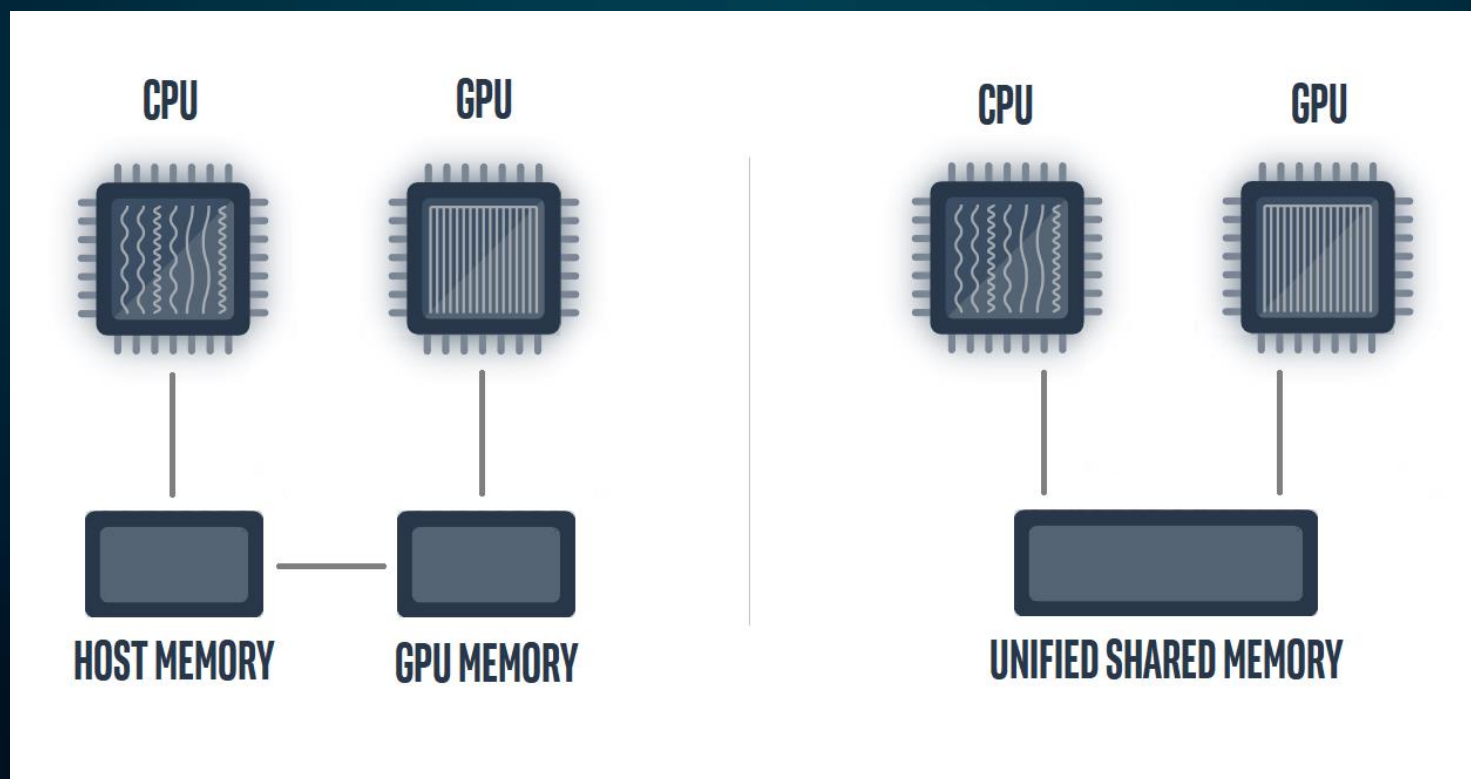
```
default_selector selector;  
// host_selector selector;  
// cpu_selector selector;  
// gpu_selector selector;  
queue q(selector);  
std::cout << "Device: " << q.get_device().get_info<info::device::name>() << std::endl;
```

- 自定义设备选择:

```
class IntelGPUSelector : public device_selector {  
public:  
    int operator()(const device& Device) const override {  
        const std::string DeviceName = Device.get_info<info::device::name>();  
  
        return Device.is_gpu() && (DeviceName.find("Intel") != std::string::npos) ? 100 : 0;  
    }  
};
```

统一共享内存 (USM)

- 统一共享内存(Unified Shared Memory)是一种基于指针的内存模型方法，适用于异构编程。



SYCL缓冲器(buffer) 方法

- 缓冲器(buffer)
- 访问器(accessor)
- 优点:
 - 非常简洁地表达了数据依赖关系
- 缺点:
 - 使用buffer不如直接使用指针和数组方便

```
#include <CL/sycl.hpp>
#include <vector>
constexpr int N = 16;
using namespace std;
using namespace sycl;
int main() {
    queue q;
    vector<int> v(N);
    {
        buffer buf(v);
        q.submit([&](handler &h) {
            accessor a(buf, h, write_only);
            h.parallel_for(N, [=](auto i) { a[i] = i; });
        }).wait();
    }

    for (int i = 0; i < N; i++)
        std::cout << v[i] << "\n";
    return 0;
}
```


DPC++ 并行内核

- 并行内核允许一个操作的多个实例并行执行
- 并行内核中没有循环和迭代器，每个操作都是完全独立的，并且不分顺序
- 并行内核使用 `parallel_for` 函数表示

CPU 应用中的 `for` 循环（能否并行
要看编译器）

```
for(int i=0; i < 1024; i++){  
    a[i] = b[i] + c[i];  
});
```

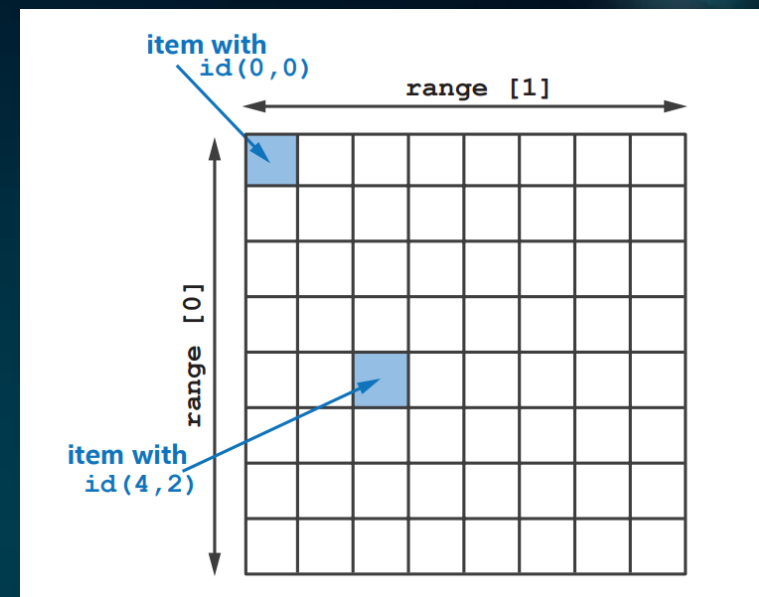


使用 `parallel_for` 卸载到加速器

```
h.parallel_for(range<1>(1024), [=](id<1> i){  
    A[i] = B[i] + C[i];  
});
```

DPC++ 基础数据并行内核

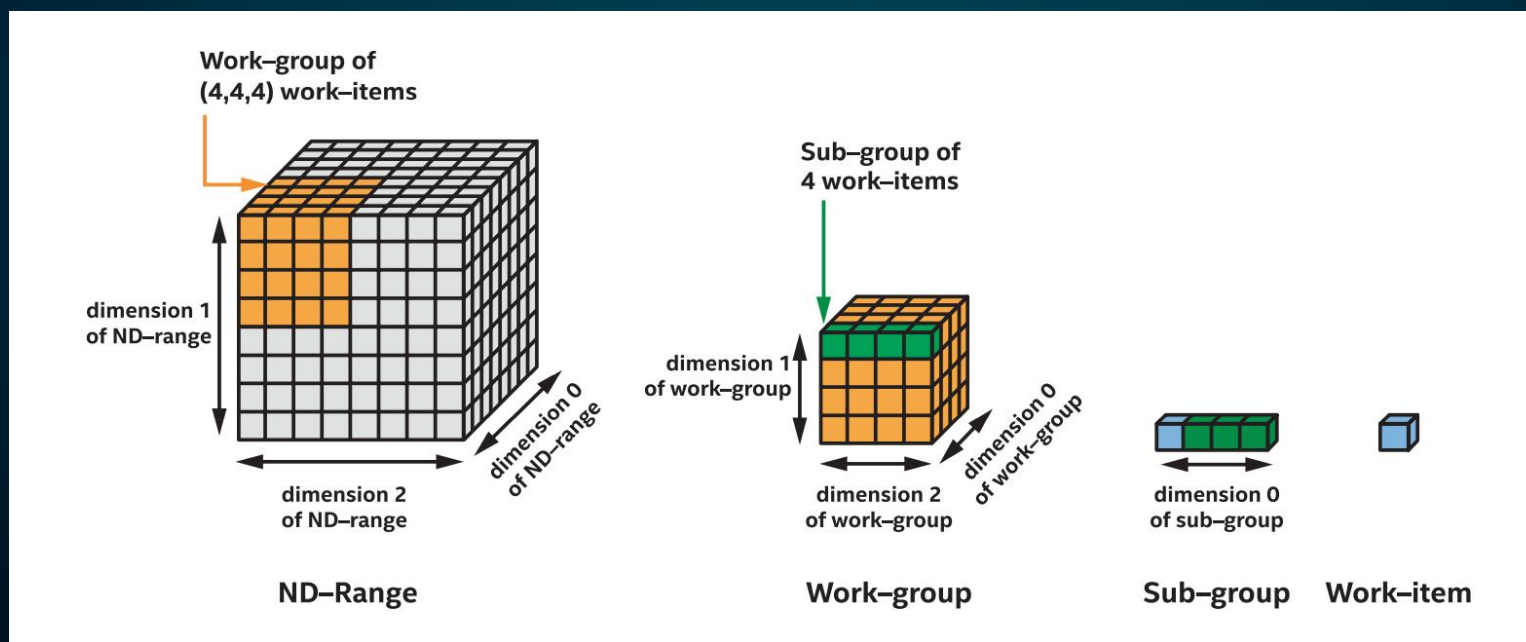
- 基本并行内核的功能通过 range、id 和 item 类提供。
- range 类用于描述并行执行维度和大小
 - 可以表示1、2、3维
 - 维度需要在编译时确定
 - 每个维度的大小可以是运行时指定
- id 类用于表示range空间中的索引
 - 同样可以表示1、2、3维
 - 维度需要在编译时确定
 - 索引一个并行运行的实例
 - buffer的偏移
- item 类代表内核函数的单个实例
 - 封装内核的执行范围和该范围内的实例索引（分别使用 get_id 和 get_range）
 - 与 range 和 id 一样，它的维度必须在编译时确定



```
h.parallel_for(range<1>(1024), [=](item<1> item){
    auto idx = item.get_id();
    auto R = item.get_range();
    // CODE THAT RUNS ON DEVICE
});
```

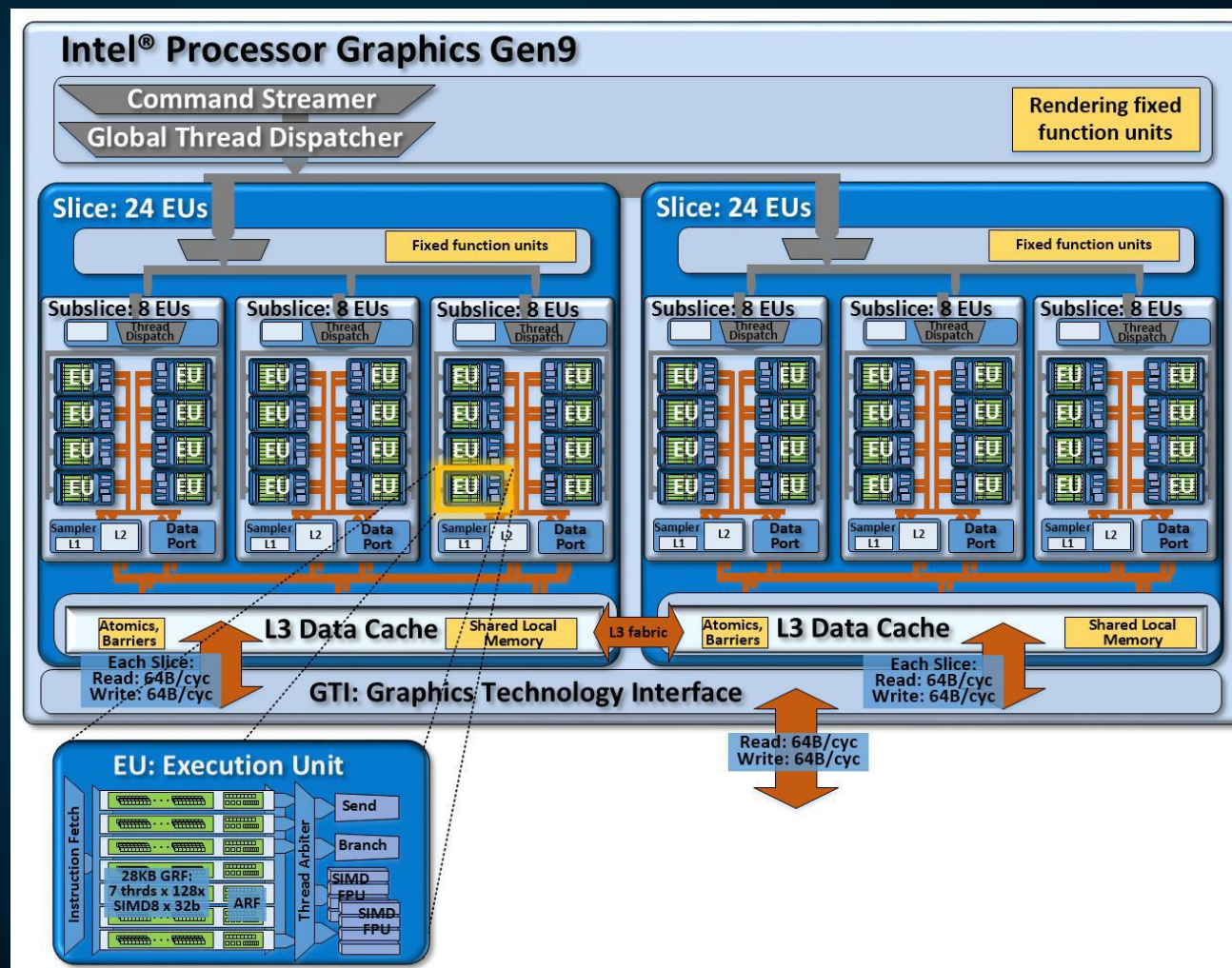
DPC++ 显式 ND-Range 内核

- 基础并行内核虽然使用方便，但是无法根据硬件架构进行优化
- ND-Range 内核可以将实例分为不同类型的分组，并且将它们精确的映射到硬件平台上
- 正确的使用 ND-Range 内核可以充分的发挥出硬件性能潜力，包括内存访问和计算单元分配等



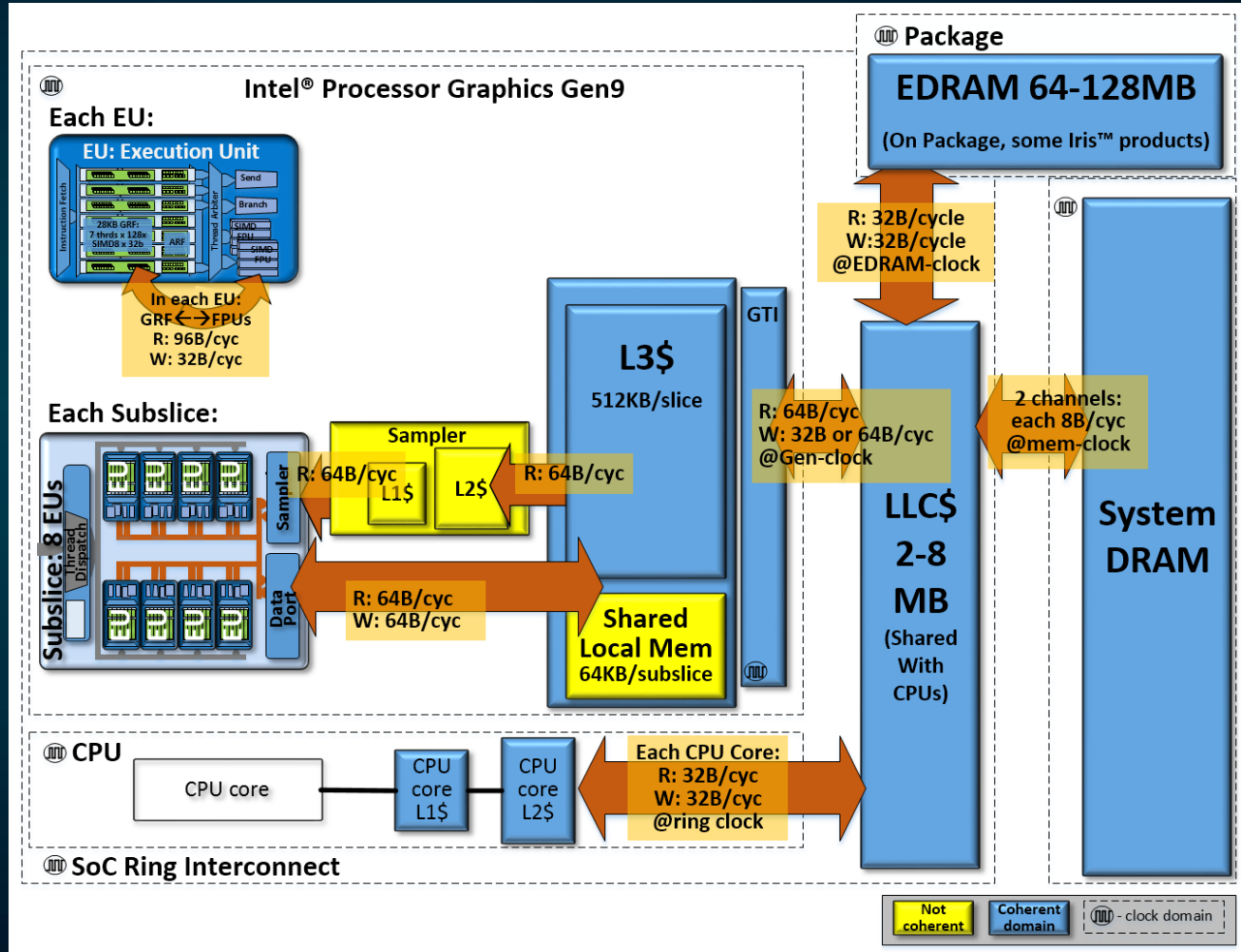
DPC++ ND-Range 发挥硬件性能原理

- Intel Skylake GT3/e Graphics
- With 48 EUs and eDRAM



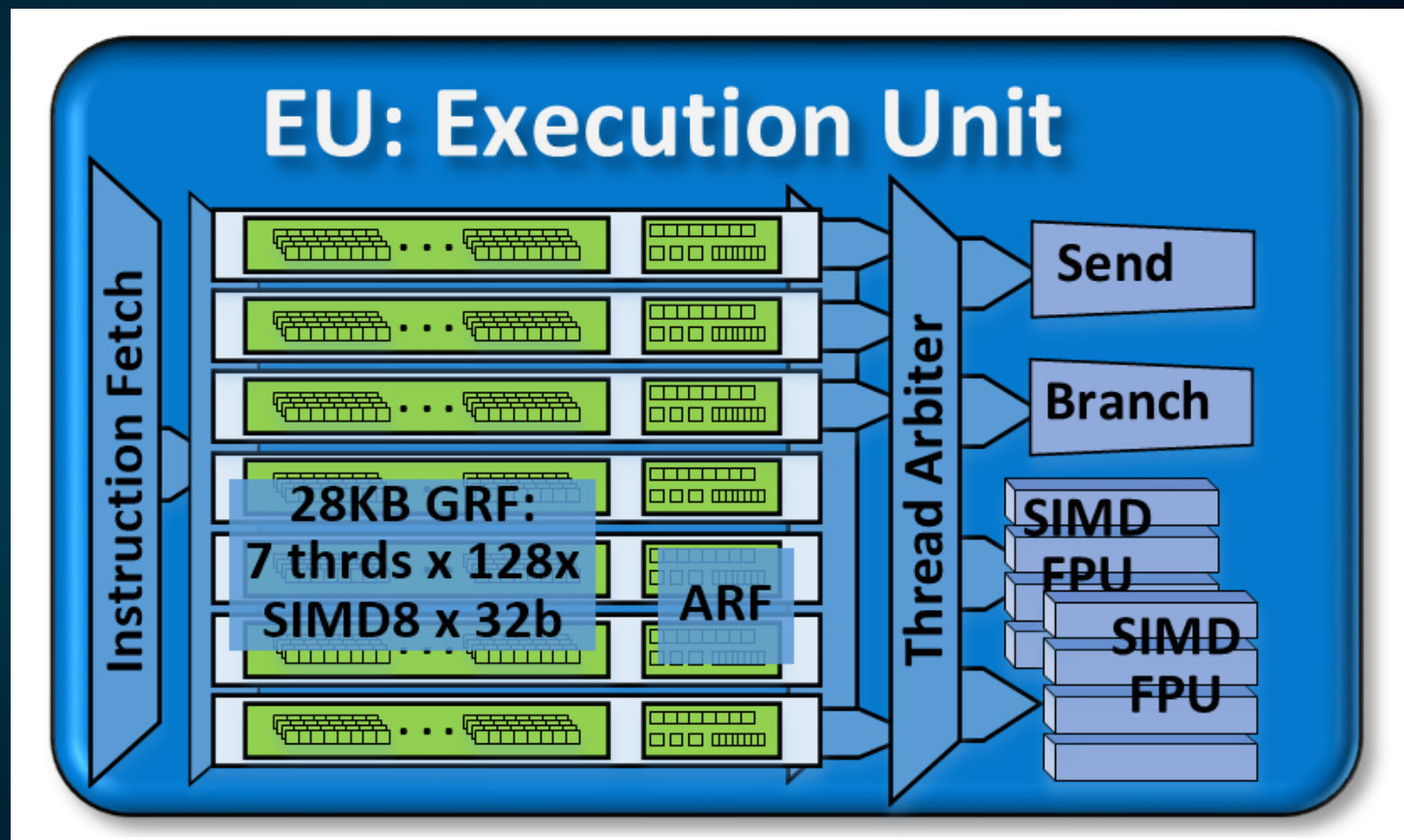
DPC++ ND-Range 发挥硬件性能原理

- Intel Skylake GT3/e Graphics
- With 48 EUs and eDRAM



DPC++ ND-Range 发挥硬件性能原理

- Intel Skylake GT3/e Graphics
- With 48 EUs and eDRAM



DPC++ 中的范围

- DPC++ 语言和运行时包含一系列 C++ 类、模板和库，但并不是所有 C++ 特性都能在任意范围使用
- 应用范围和命令组范围：
 - 在主机上执行的代码
 - 应用和命令组范围内均支持 C++ 的所有功能
- 内核范围：
 - 在设备上执行的代码
 - 内核范围支持的 C++ 有部分限制
 - 更广泛的设备支持和大规模并行性
 - 不支持 C++ 特性包括：动态多态性、动态内存分配（因此不使用 new 或 delete 运算符进行对象管理）、静态变量、函数指针、运行时类型信息 (RTTI) 和异常处理。
 - 不允许从内核代码调用任何虚拟成员函数和可变参数函数。
 - 内核代码中不允许递归。

03

SYCL & DPC++ 编译器设计简介

Intel DPC++ Compiler

Compiler	Notes	Linux* Driver	Windows* Driver
Intel® DPC++ Compiler	A C++ and Khronos SYCL* compiler with a Clang front-end.	dpcpp	dpcpp (clang compatible) dpcpp-cl (clang-cl compatible)
Intel® C++ Compiler	A C++ compiler with a Clang front-end, supporting OpenMP* offload.	icx for C icpx for C++	icx

应用程序构建流程

• 前端

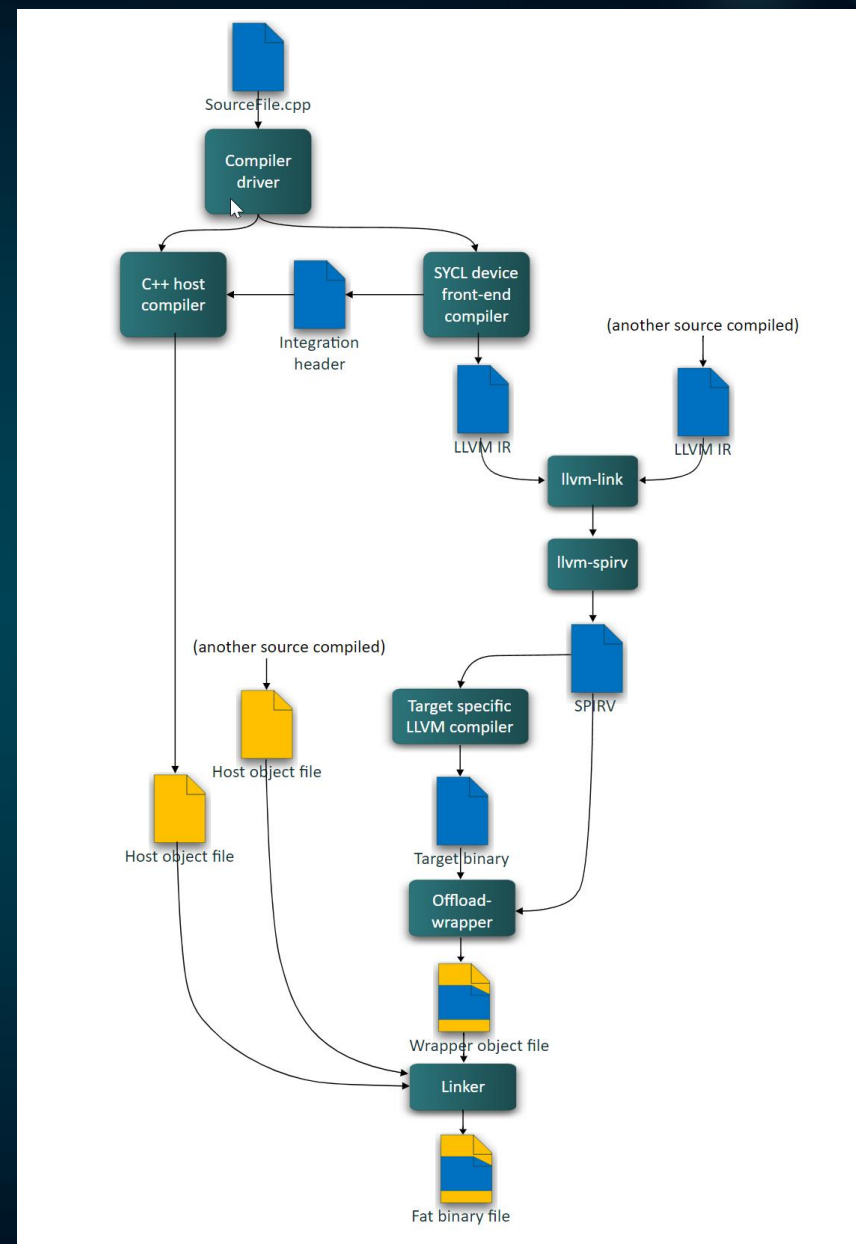
- 解析输入源, “概述”代码的设备部分, 对设备代码应用额外的限制;
- 为设备代码生成 LLVM IR 和提供运行时库的内核名称、参数顺序和数据类型的“集成标头”。

• 中端

- 转换初始 LLVM IR 以供后端使用。
- 例如: LLVM IR → SPIR-V 转换器

• 后端

- 生成本机“设备”代码。
- 编译时 (在提前编译场景中) 或在运行时 (在即时编译场景中) 被调用。



转存SPIR-V

3.1. Magic Number

Magic number for a SPIR-V module.

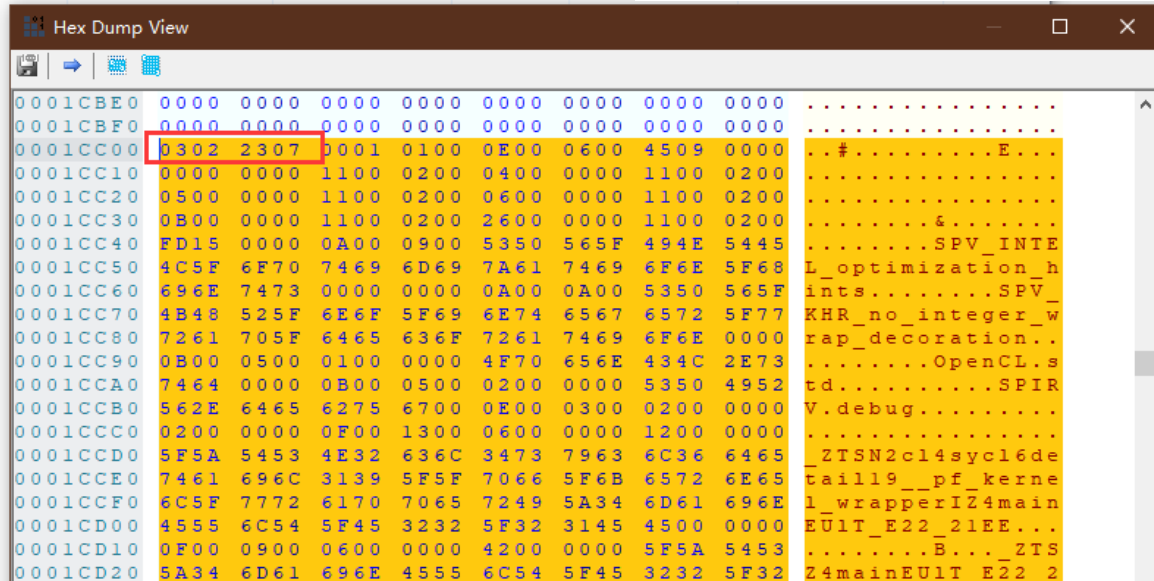
Tip

Endianness: A module is defined as a stream of words, not a stream of bytes. However, if stored as a stream of bytes (e.g., in a file), the magic number can be used to deduce what endianness to apply to convert the byte stream back to a word stream.

Magic Number

0x07230203

Offset	Name	Virtual Size	Virtual Address	RVA	Pointer 1
00000208	.text	0x00011961 (70.3 KB)	0x0000000040001000	0x00001000	0x00000000
00000230	.text.st	0x00001043 (4.6 KB)	0x0000000040013000	0x00013000	0x00001000
00000280	.rdata	0x0000571C (21.7 KB)	0x0000000040015000	0x00015000	0x00001000
000002F8	.data	0x00002971 (10.3 KB)	0x000000004001B000	0x0001B000	0x00001000
00000398	.pdata	0x00000B04 (2.7 KB)	0x000000004001E000	0x0001E000	0x00001000
00000460	.idata	0x00002274 (8.6 KB)	0x000000004001F000	0x0001F000	0x00001000
00000550	.tgtsym	0x000001C2 (450 bytes)	0x0000000040022000	0x00022000	0x00001000
00000668	__CLANG__	0x0001B8F3 (110.2 KB)	0x0000000040023000	0x00023000	0x00001000
000007A8	.00cfg	0x00000151 (337 bytes)	0x000000004003F000	0x0003F000	0x00003000
00000910	__RDATA	0x00000953 (2.3 KB)	0x0000000040040000	0x00040000	0x00003000
00000AA0	.reloc	0x00000681 (1.6 KB)	0x0000000040041000	0x00041000	0x00003000



Clang 前端中的 SYCL 支持

- 设备代码概述
 - 负责识别和概述单一来源中的“设备代码”。
- SYCL 内核函数对象（仿函数或 lambda）底层化
 - 为 SYCL 内核创建了一个 OpenCL 内核函数接口。
- 设备代码诊断
 - 对设备代码实施语言限制。
- 集成标头生成
 - 发出的信息是使用 OpenCL API 绑定 SYCL 代码的主机和设备部分所需要的。

Clang 前端中的 SYCL 支持

```
int foo(int x) { return ++x; }
int bar(int x) { throw std::exception{"CPU code only!"}; }
...
using namespace cl::sycl;
queue Q;
buffer<int, 1> a{range<1>{1024}};
Q.submit([&](handler& cgh) {
    auto A = a.get_access<access::mode::write>(cgh);
    cgh.parallel_for<init_a>(range<1>{1024}, [=](id<1> index) {
        A[index] = index[0] * 2 + foo(42);
    });
})
...
}
```

Clang 前端中的 SYCL 支持

```
// SYCL kernel is defined in SYCL headers:
template <typename KernelName, typename KernelType/*, ...*/>
__attribute__((sycl_kernel)) void sycl_kernel_function(KernelType KernelFuncObj) {
    // ...
    KernelFuncObj();
}

// Generated OpenCL kernel function
__kernel KernelName(global int* a) {
    KernelType KernelFuncObj; // Actually kernel function object declaration
    // doesn't have a name in AST.
    // Let the kernel function object have one captured field - accessor A.
    // We need to init it with global pointer from arguments:
    KernelFuncObj.A.__init(a);
    // Body of the SYCL kernel from SYCL headers:
    {
        KernelFuncObj();
    }
}
```

设备代码单独链接

- 用户将源代码分成四个文件：dev_a.cpp、dev_b.cpp、host_a.cpp和host_b.cpp，其中只有dev_a.cpp和dev_b.cpp包含设备代码。

设备链接： dev_a.cpp
dev_b.cpp -> dev_image.o

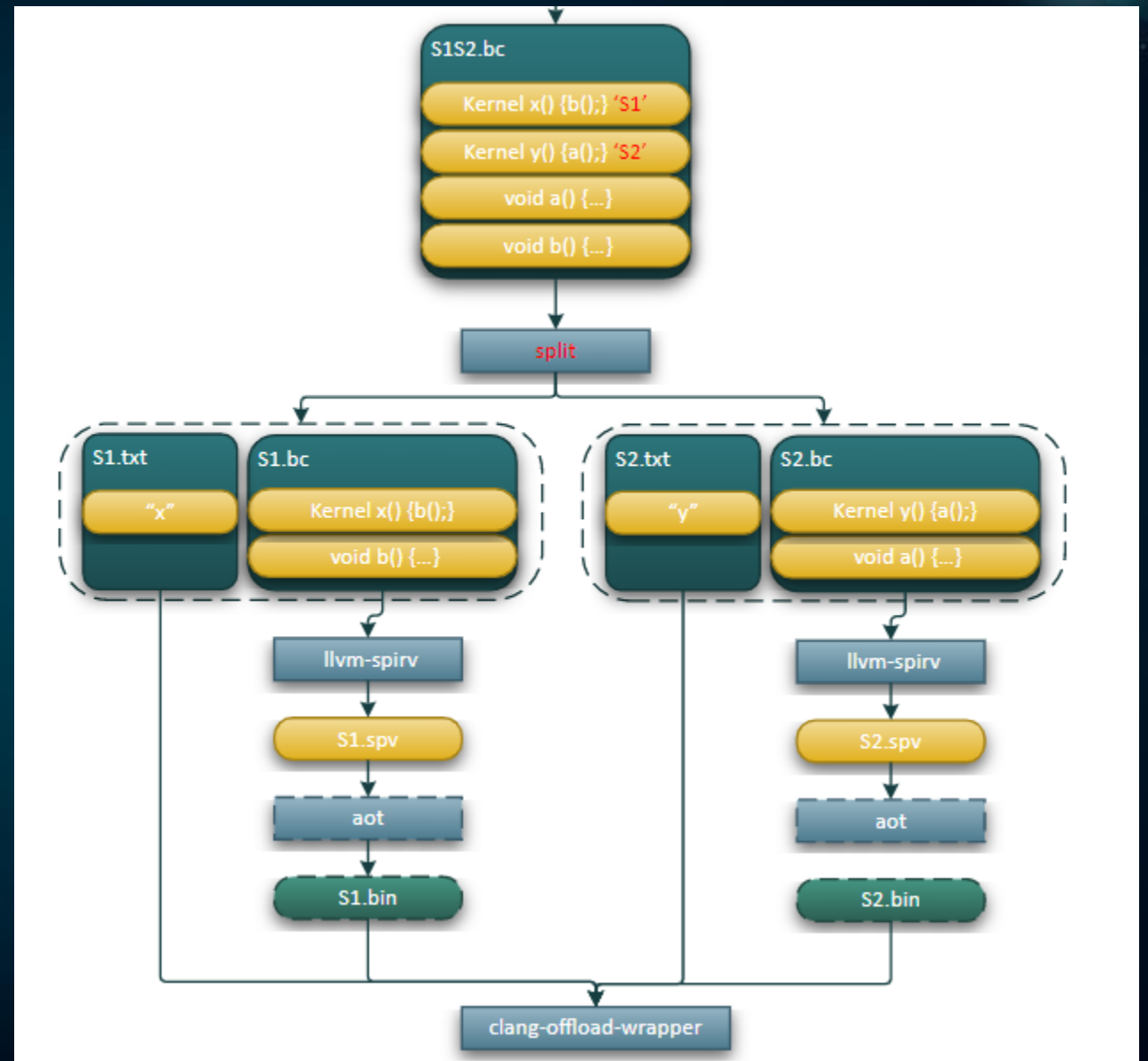
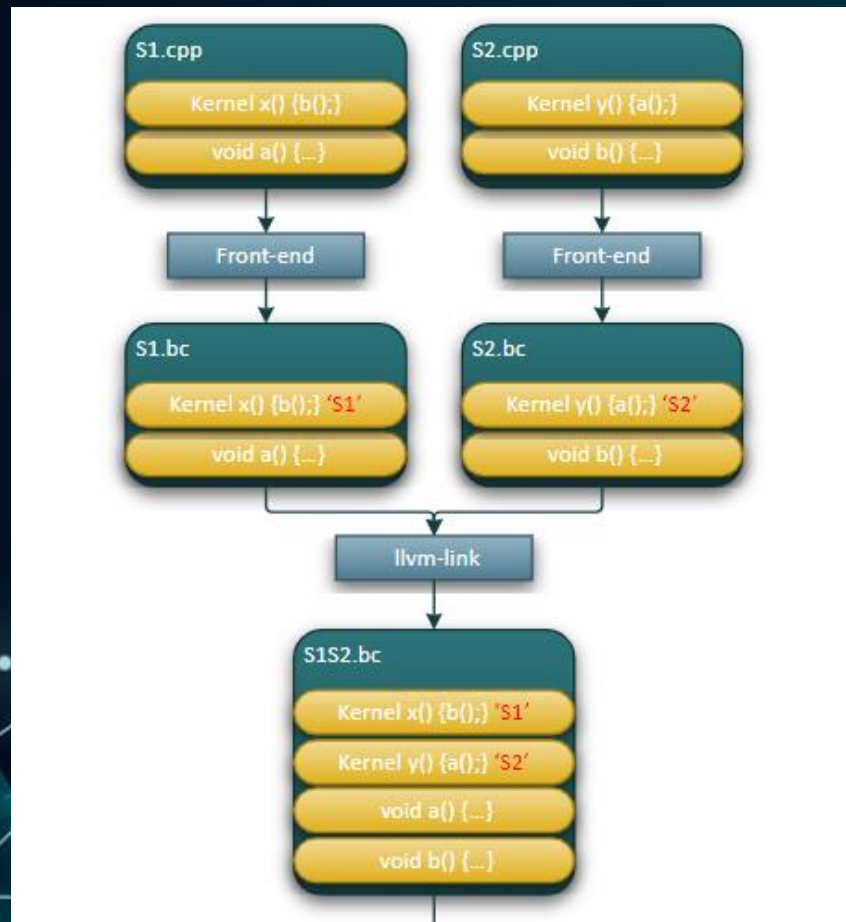
链接： dev_image.o
host_a.o host_b.o -> 可执
行文件

主机编译： host_a.cpp ->
host_a.o; host_b.cpp ->
host_b.o

设备代码拆分

- 将所有设备代码放入单个 SPIR-V 模块在以下情况下效果不佳：
 - 定义了数千个内核，其中只有一小部分在运行时使用。将它们全部放在一个 SPIR-V 模块中会显著增加 JIT 时间。
 - 设备代码可以专门用于不同的设备。例如，只有在 FPGA 上才能执行的内核应该仅能够适用于 FPGA 的扩展。即使从未在其他设备上调用此特定内核，这也会导致其他设备上的 JIT 编译失败。
- 为了解决这些问题，编译器可以将单个模块拆分为更小的模块：
 - 根据源代码生成单独的模块（翻译单元）
 - 根据单个内核生成单独的模块

设备代码拆分

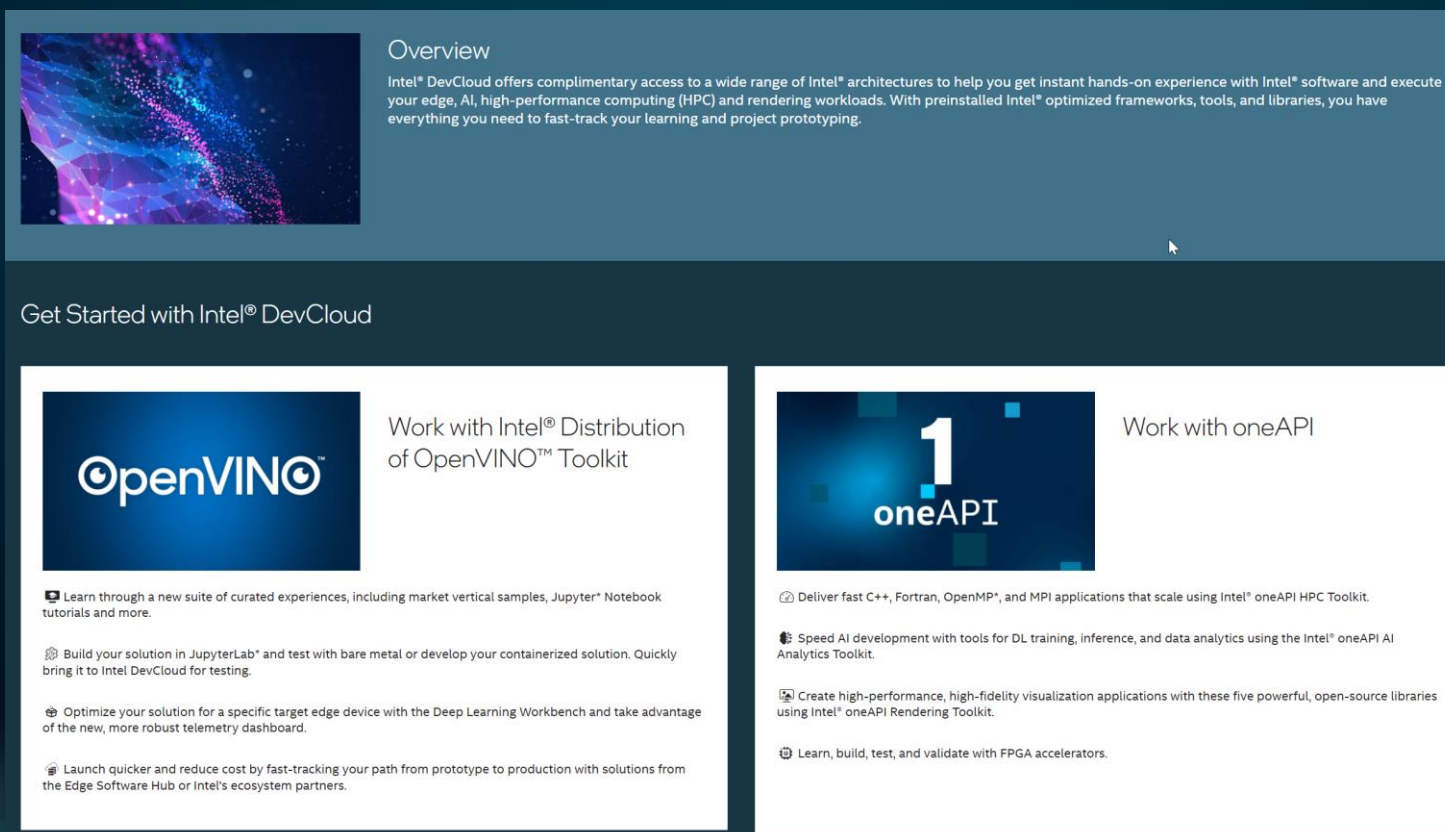


04

DevCloud中
学习SYCL & DPC++

DevCloud简介

- DevCloud是Intel提供的免费访问各种Intel架构的云平台，它可以让使用者快速动手体验和学习各种优化框架、工具和库。
 - 访问链接：<https://www.intel.com/content/www/us/en/developer/tools/devcloud/overview.html>



The screenshot shows the Intel DevCloud overview page. At the top, there is a section titled "Overview" with a blue and purple abstract background image. Below this, the text describes the platform's purpose. The main content area is titled "Get Started with Intel® DevCloud" and features two columns of information. The left column is for "OpenVINO™" and the right column is for "oneAPI".

Overview

Intel® DevCloud offers complimentary access to a wide range of Intel® architectures to help you get instant hands-on experience with Intel® software and execute your edge, AI, high-performance computing (HPC) and rendering workloads. With preinstalled Intel® optimized frameworks, tools, and libraries, you have everything you need to fast-track your learning and project prototyping.

Get Started with Intel® DevCloud

Work with Intel® Distribution of OpenVINO™ Toolkit

- Learn through a new suite of curated experiences, including market vertical samples, Jupyter® Notebook tutorials and more.
- Build your solution in JupyterLab® and test with bare metal or develop your containerized solution. Quickly bring it to Intel DevCloud for testing.
- Optimize your solution for a specific target edge device with the Deep Learning Workbench and take advantage of the new, more robust telemetry dashboard.
- Launch quicker and reduce cost by fast-tracking your path from prototype to production with solutions from the Edge Software Hub or Intel's ecosystem partners.

Work with oneAPI

- Deliver fast C++, Fortran, OpenMP®, and MPI applications that scale using Intel® oneAPI HPC Toolkit.
- Speed AI development with tools for DL training, inference, and data analytics using the Intel® oneAPI AI Analytics Toolkit.
- Create high-performance, high-fidelity visualization applications with these five powerful, open-source libraries using Intel® oneAPI Rendering Toolkit.
- Learn, build, test, and validate with FPGA accelerators.

通过ssh连接DevCloud

- 连接教程详细;
- 操作过程操作简单;
- 编程自由度高;
- 推荐经过学习后实验使用。

1

Connect to DevCloud ^

Connect to the DevCloud using SSH Clients.

Select Preferred OS/Client Interface:

- Cygwin on Windows*
- Linux* or macOS*
- Visual Studio Code

2

Hello World! Get Started by running a simple sample on DevCloud. v

Use this simple sample to confirm that you are connected to oneAPI DevCloud

3

Run Base Toolkit Samples on DevCloud

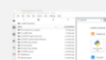
Explore the [samples](#) already installed in Step 2.

[Browse Available Samples](#)

通过jupyter notebook使用DevCloud

- 非常详尽的学习教程：
 - 教学循序渐进；
 - 文字和图片描述DPC++方方面面；
 - 附带可修改和运行代码示例。
- 强烈推荐初学者使用。

Learn the Essentials of Data Parallel C++



Module 0
Introduction to JupyterLab* and Notebooks.
Learn to use Jupyter notebooks to modify and run code as part of learning exercises.

[Try it in Jupyter](#)



Module 1
Introduction to DPC++

- Articulate how oneAPI can help to solve the challenges of programming in a heterogeneous world.
- Use oneAPI solutions to enable your workflows.
- Understand the DPC++ language and programming model.
- Become familiar with using Jupyter notebooks for training throughout the course.

[Try it in Jupyter](#)



Module 2
DPC++ Program Structure

- Articulate the SYCL* fundamental classes.
- Use device selection to offload kernel workloads.
- Decide when to use basic parallel kernels and ND Range Kernels.
- Create a host accessor.
- Build a sample DPC++ application through hands-on lab exercises.

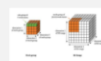
[Try it in Jupyter](#)



Module 3
DPC++ Unified Shared Memory

- Use new DPC++ features like Unified Shared Memory (USM) to simplify programming.
- Understand implicit and explicit ways of moving memory using USM.
- Solve data dependency between kernel tasks in an optimal way.

[Try it in Jupyter](#)



Module 4
DPC++ Sub-Groups

- Understand advantages of using Sub-groups in DPC++.
- Take advantage of Sub-group collectives in ND-Range kernel implementation.
- Use Sub-group Shuffle operations to avoid explicit memory operations.

[Try it in Jupyter](#)



Module 5
Demonstration of Intel® Advisor

- See how Offload Advisor¹ identifies and ranks parallelization opportunities for offload.
- Run Offload Advisor using command line syntax.
- Use performance models and analyze generated reports.

Offload Advisor is a feature of Intel Advisor installed as part of the Intel(R) oneAPI Base Toolkit.

[Try it in Jupyter](#)



Module 6
Intel® VTune™ Profiler on Intel® DevCloud

- Profile a DPC++ application using Intel® VTune™ Profiler on Intel® DevCloud.
- Understand the basics of command line options in VTune Profiler to collect data and generate reports.

[Try it in Jupyter](#)





Module 7
DPC++ Library Utilization
Maximize productivity with this companion to Intel® oneAPI DPC++ Compiler providing an alternative for C++ developers.

[Try it in Jupyter](#)

通过jupyter notebook使用DevCloud

Launcher Unified_Shared_Memory.ipynb

1. Inspect the code cell below and click run  to save the code to file.

2. Next run  the cell in the **Build and Run** section below the code to compile and execute the code.

```
[1]: %%writefile lab/usm.cpp
//=====
// Copyright © 2020 Intel Corporation
//
// SPDX-License-Identifier: MIT
// =====
#include <CL/sycl.hpp>
using namespace sycl;

static const int N = 16;

int main() {
    queue q;
    std::cout << "Device : " << q.get_device().get_info<info::device::name>()
              << std::endl;

    ///< USM allocation using malloc_shared
    int *data = malloc_shared<int>(N, q);


    ///< Initialize data array
    for (int i = 0; i < N; i++) data[i] = i;

    ///< Modify data array on device
    q.parallel_for(range<1>(N), [=](id<1> i) { data[i] *= 2; }).wait();

    ///< print output
    for (int i = 0; i < N; i++) std::cout << data[i] << std::endl;
    free(data, q);
    return 0;
}
```

Overwriting lab/usm.cpp

Build and Run


Select the cell below and click run  to compile and execute the code:

```
[1]: ! chmod 755 q; chmod 755 run_usm.sh; if [ -x "$(command -v qsub)" ]; then ./q run_usm.sh; else ./run_usm.sh; fi
```

Job has been submitted to Intel(R) DevCloud and will execute soon.

If you do not see result in 60 seconds, please restart the Jupyter kernel:
Kernel -> 'Restart Kernel and Clear All Outputs...' and then try again

Job ID	Name	User	Time Use	S Queue
1849943.v-qsvr-1	...ub-singleuser	u123998	00:00:12	R jupyterhub
1849944.v-qsvr-1	run_usm.sh	u123998	0	Q batch

Waiting for Output 

TimeOut 60 seconds: Job is still queued for execution, check for output file later (run_usm.sh.o1849944)

Done!

```
#####
# Date: Sat 12 Feb 2022 04:24:19 AM PST
# Job ID: 1849944.v-qsvr-1.aidevcloud
# User: u123998
# Resources: neednodes=1:gpu:ppn=2,nodes=1:gpu:ppn=2,walltime=06:00:00
#####
## u123998 is compiling DPCPP_Essentials Module3 -- DPCPP Unified Shared Memory - 1 of 4 usm.cpp
Device : Intel(R) UHD Graphics P630 [0x3e96]
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
#####
# End of output for job 1849944.v-qsvr-1.aidevcloud
# Date: Sat 12 Feb 2022 04:24:40 AM PST
#####
Job Completed in 60 seconds.
```


谢谢聆听