

# SYCL编码实践与编译设计浅析



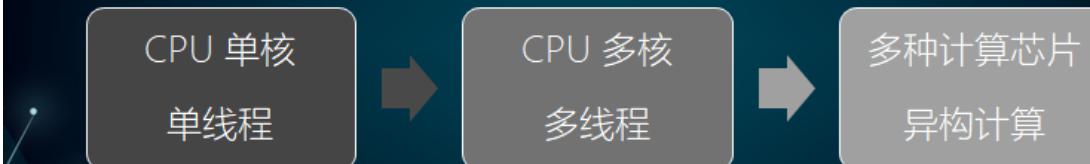
## 背景介绍

今天的计算机世界和20年前相比发生了翻天覆地的变化。2000年的时候，处理器的速度还在遵循摩尔定律，也就是说处理器的性能大约每两年翻一倍。在那个时候，编写单线程程序在不经过任何优化的情况下，更换主频更高的CPU就可以“无缝”的提高软件性能。

当然，我们都知道，摩尔定律并没有延续太长时间。在2006年左右的时候，CPU主频已经很难提高了，所以CPU厂商调整了思路，推出了更多的多核处理器。这对于程序员来说也提出了一个挑战，我们需要编写多线程程序以提高CPU的利用率，提高程序效率。幸运的是，写多线程程序并不会带来太多问题，因为硬件架构没有变化，指令集没有变化，无非是编写程序时需要注意资源的竞争问题。

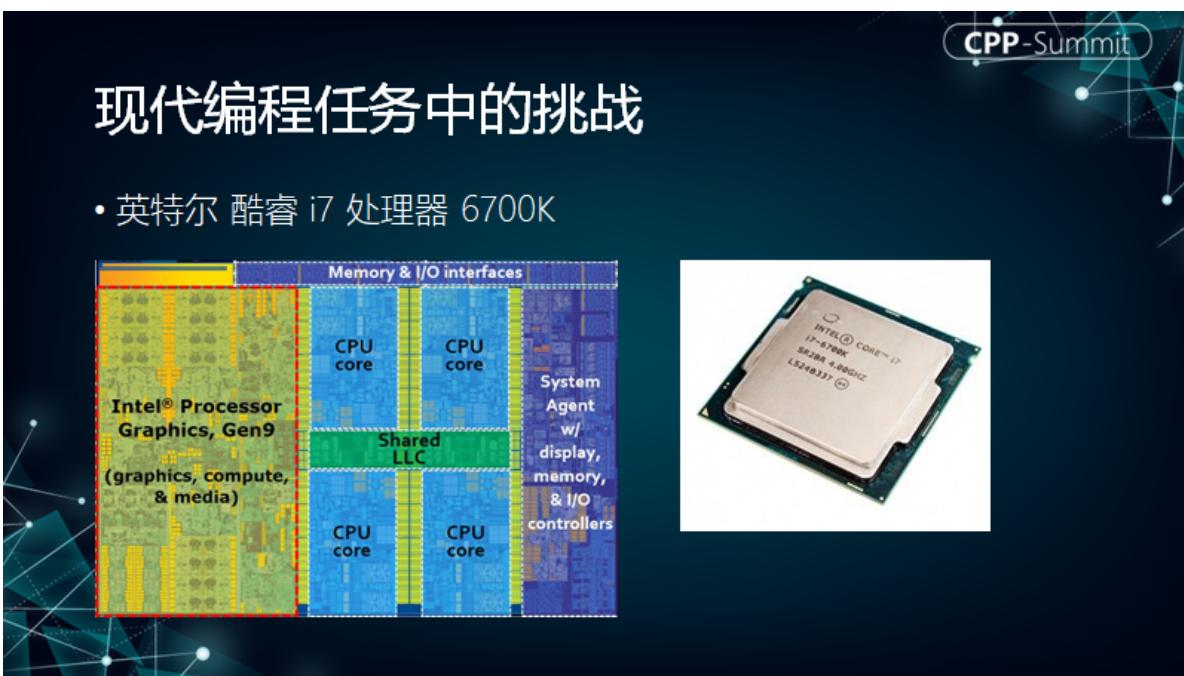
随着时间的推移，我们又经过了10年，在这10年中我们见证了AI和区块链技术的崛起，我们发现CPU的计算能力已经不能满足对计算的需求了。我们需要更加专注计算的设备帮助我们提高程序的效率，例如GPU、TPU等等。

# 计算时代的变化



我们称这种混合多种设备做计算的形态为异构计算。让我们用更加学术一些语言来描述异构计算——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的系统。

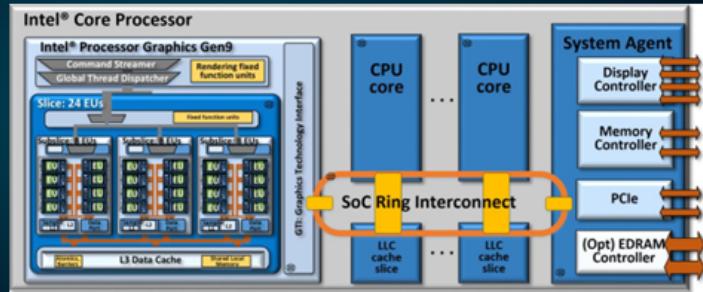
可以看出，相对于编写多线程程序，编写不同架构和不同指令集的程序的难度是非常高的，但是也是非常有意义的。让我们来看看一颗带有显示芯片处理器的架构：



这是一颗比较老的处理器：Intel i7 6700K，但是我们必须意识到其GPU的运算能力并不弱。从面积上来看，GPU芯片的“占地面积”几乎和CPU平分秋色。也就是说，如果我们编写的程序只用到了CPU部分，那么是相当不划算的，因为大量“面积”的计算能力是没有用到的。

# 现代编程任务中的挑战

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

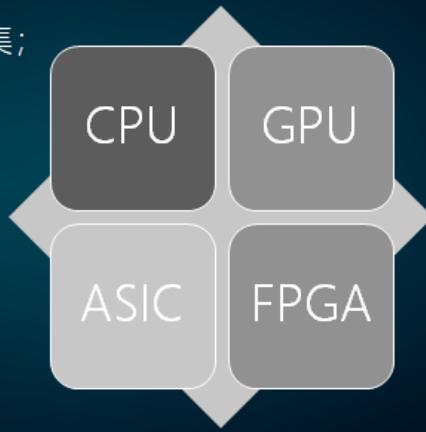


在这个“面积中”的芯片中，拥有192个着色器，24个纹理处理单元，3个光栅化处理单元和24个执行单元，对于需要做并行计算的程序来说，这个资源也是非常可贵的。

当然，这个时候问题也就来了。相同架构、指令集和语言的程序还比较容易写出来。那么不同架构，不同指令集，不同工具栏，不同语言的程序写起来就相当费劲了，这也是我们写异构程序时面临的重大挑战。

# 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。



为了应对异构计算中的挑战，Intel推出了他们的新开发框架oneAPI。具体来说，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。



简单来说，oneAPI就像操作系统的硬件抽象层，他让程序员不需要了解底层硬件有哪些，分别是什么。作为程序员，我们只需要知道我们的业务逻辑是什么，任务是什么，然后调用框架或者oneAPI接口即可。至于怎么和硬件交互，怎么把计算任务给到特定的加速器就不需要程序员关心了。当然了，既然oneAPI已经把硬件接管了，那么关于兼容性和移植性这些问题，也都是由oneAPI来处理的。

### 英特尔® oneAPI 基础工具套件

直接编程	基于 API 的编程	分析和调试工具
<a href="#">英特尔® oneAPI DPC++/C++ 编译器</a>	<a href="#">英特尔® oneAPI DPC++ 库 oneDPL</a>	<a href="#">英特尔® VTune 分析器</a>
<a href="#">英特尔® DPC++ 兼容性工具</a>	<a href="#">英特尔® oneAPI 数学核心函数库 - oneMKL</a>	<a href="#">英特尔® Advisor</a>
<a href="#">英特尔® Python 分发版</a>	<a href="#">英特尔® oneAPI 数据分析库 - oneDAL</a>	<a href="#">英特尔® GDB 分发版</a>
<a href="#">面向 oneAPI 基础工具套件的英特尔® FPGA 插件</a>	<a href="#">英特尔® oneAPI 线程构建模块 - oneTBB</a>	
	<a href="#">英特尔® oneAPI 视频处理库 - oneVPL</a>	
	<a href="#">英特尔® oneAPI 聚合通信库 - oneCCL</a>	
	<a href="#">英特尔® oneAPI 深度神经网络库 - oneDNN</a>	
	<a href="#">英特尔® 集成性能基元 - 英特尔® IPP</a>	

**intel oneAPI BASE TOOLKIT**

需要额外介绍的是，oneAPI的主要任务虽然是满足异构计算的需求，但是他能做的事情远远不止这些。我们可以看到oneAPI的基础套件，除了这篇文章主要介绍的SYC&DPC++以外，还有很多的实用工具和库，比如Vtune和TBB。有兴趣的朋友可以在官方网站上看到更加详尽的介绍，而我们现在要直入主题，介绍SYCL的基础编程技术。

## 基础编程

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

Host code

Device code

Host code

```
#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;
}
```

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

free: 使用sycl::free释放分配的USM

先看到一个简单的示例代码，虽然代码只有十多行，但是也是一个典型的SYCL代码了。

首先要说明的是，这份代码是一个单一源代码，也就是说主机代码host code和设备代码device code是混合在一起的。其次这份代码完全是标准C++代码，没有任何扩展关键字或者语法。

这份代码分为了三个部分，第一部分是host code，第二部分是device code，第三部分回到host code。其中用到了SYCL最典型的三个组件：队列、统一共享内存和并行内核函数。

## 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

这里队列queue用于提交命令组(command group)到SYCL运行时执行，它是一种将工作(work)提交到设备的机制。一个队列可以映射(map)到一个设备，多个队列(multiple queue)可以映射到同一设备。queue的成员函数parallel\_for可以将命令提交到队列，不过这只是一种简化写法，更加完整的写法是调用submit函数，submit函数将提交一个命令组lambda表达式，在表达式中命令组对象的成员函数再调

用parallel\_for函数，将内核函数提交给队列。

有一点必须注意的是，队列可以将命令提交给设备，但是具体提交给什么设备应该是可以指定的。所以SYCL提供了几种方法指定设备，一种是直接指定设备类别，比如default\_selector、cpu\_selector、gpu\_selector等，这种选择器都是SYCL标准中预设好的。不过这种选择设备的方式也有局限性，因为如果同一类型存在多种不同的设备，上述方式就没有选择。

- 队列可以通过设备选择器(device selector)选择指定设备：
- 自定义设备选择：

```
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;
```

SYCL当然也考虑到这一点，所以标准提供了一种继承device\_selector的方式来选择设备，device\_selector是一个抽象类，我们需要继承并且实现函数调用运算符，也就是operator()。在这个函数中，可以通过参数device来获取设备信息，比如类型和名称，然后通过这些信息选择目标设备。比如这里的代码示例，就是选择一个Intel的GPU设备并返回100，数值越大优先级越高。

- 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。

另外一个值得注意的是统一共享内存，我们知道CPU使用的内存和GPU使用的内存是不同的，通常是需要数据转换的，但是通过DPC++的机制，可以让程序员编写的程序似乎在使用“同一片”内存，非常的简单方便。

# 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;
}
```

如果不使用统一共享内存，可以使用SYCL的缓冲器方法。这种方法相对于统一共享内存要复杂一点，使用到了缓冲器和访问器。我们可以观察代码，首先需要将数据对象，这里是vector，作为参数传给buffer的构造函数，用于构造buffer对象，然后在命令组范围内使用buffer构造accessor，最后才能在内核函数中使用accessor去访问vector的数据。值得注意的是，buffer必须使用一个作用域包括起来，因为buffer只有在析构的时候才会将加速设备的数据转移到主机内存中。很显然，使用这种方式不如直接操作指针简单方便。

## 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

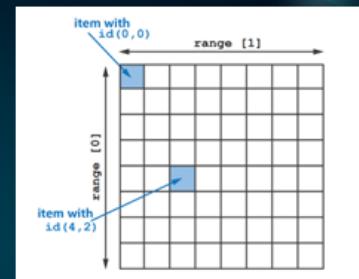
```
h.parallel_for(range<1>(1024), [=](id<1> i){
    A[i] = B[i] + C[i];
});
```

接下来通过介绍并行内核的方式，来大概的解释一下并行计算的原理。我们先来看上图左边的代码，这是一个非常简单的for循环代码，那么CPU在运行这部分代码的时候一般情况是每次循环进行一次赋值，并且是按照i从小到大的顺序执行。当然如果编译器足够聪明，并且这里对a数组的赋值不会影响到b和c，那么编译器可能会进行一些编译优化，让循环一次能并发执行多次赋值，总之这得看编译器的能力。

而右边的代码就不同了，很明显，它是使用了并行计算，那么在这个并行操作中，并不存在循环和迭代操作，每个操作都是完全独立的发射到硬件设备中执行，并且没有顺序之分。所以在这种情况下，必然能保证代码能高效的并发执行。

# 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
});
```

让我们再深入一步，来看一下基础数据并行内核的编程方法和原理。基础并行内核功能需要三个类来完成，分别是range、id和item。在介绍这三个类之前，我们先看右上角的这幅图，这幅图可以看成要执行得实例得合集，每一个小方块都是一个实例。这个合集是一个二维的合集，合集的范围用range类来表示，其实range可以是一维、二维或者三维的，这张图只表示了二维。

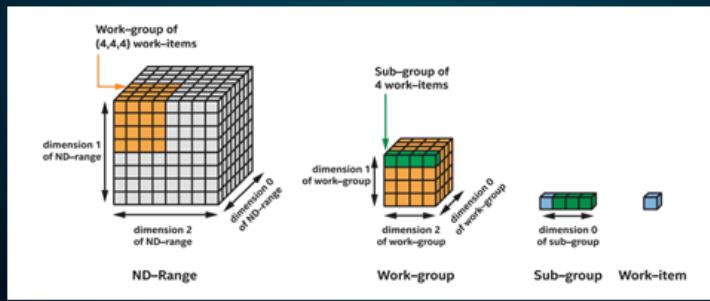
回到range类，它实际上是个类模板，它的维数是模板参数，所以需要在编译时确定，而每个维度的大小可以在运行时指定。

id类也是一个类模板，模板参数也是维度，同样的需要在编译时确定，它可以表示一个实例，也就是一个小方块在运行时的索引，或则buffer的偏移。

item类相对复杂一点，模板参数也是维度，它代表内核函数的单个实例，也就是说它封装了一些内核函数，比如可以获取内核执行范围，也就是range，或者执行索引，也就是id，具体的函数为get\_range和get\_id。

## DPC++ 显式 ND-Range 内核

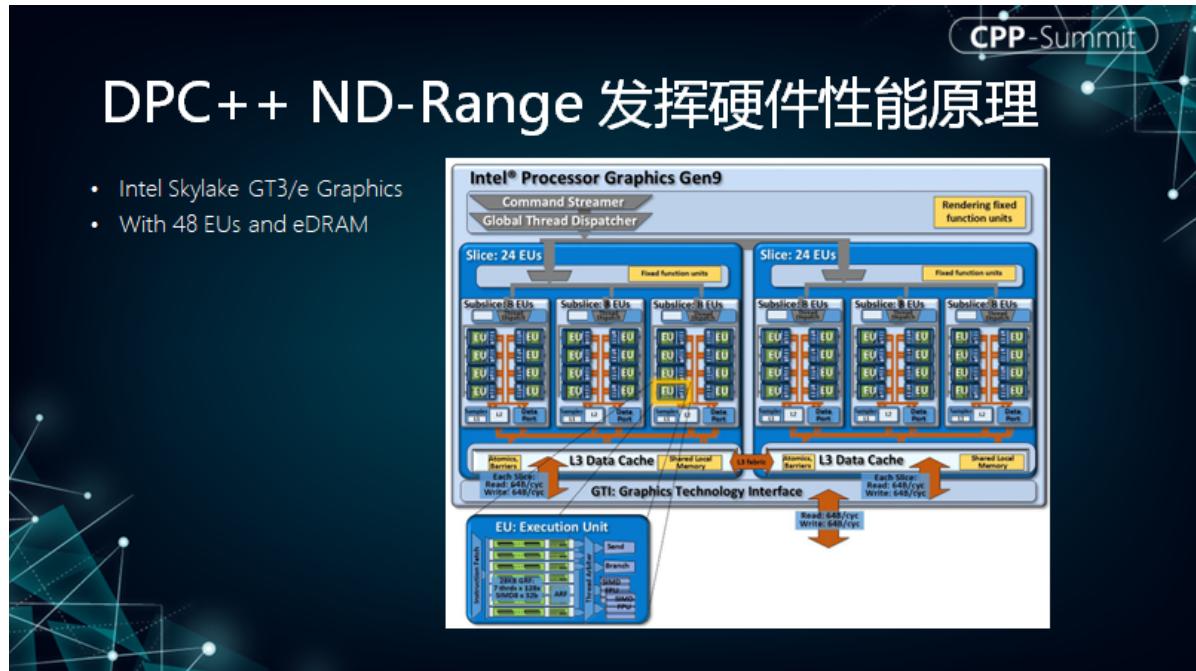
- 基础并行内核虽然使用方便，但是无法根据硬件架构进行优化
- ND-Range 内核可以将实例分为不同类型的分组，并且将它们精确的映射到硬件平台上
- 正确的使用 ND-Range 内核可以充分的发挥出硬件性能潜力，包括内存访问和计算单元分配等



介绍完了基础数据并行内核，现在来简单介绍一下显式 ND-Range 内核。通过前面介绍的基础数据并行内核，我们可以发现，使用它编写代码非常简单方便，但是问题是它无法根据硬件架构来做优化。而 ND-Range 就不同了，它可以将实例拆分为不同的类型的分组，再根据分组类型的特点映射到相应的硬件平台上。

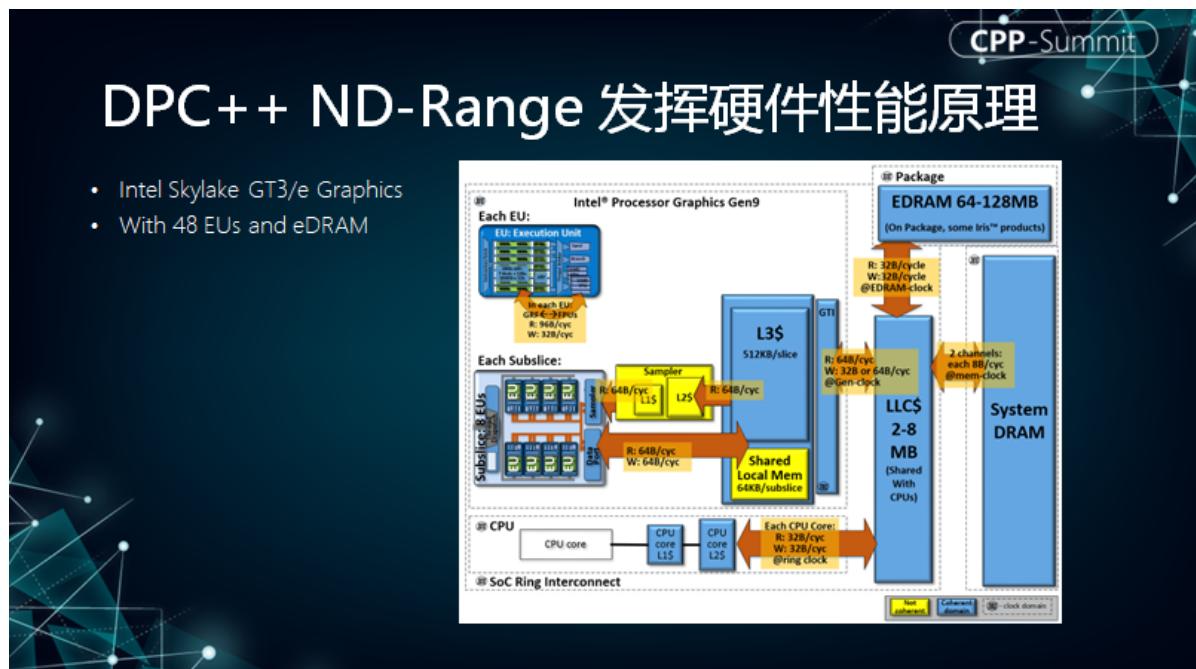
我们来看上面的这张图，最左边的是一个三维的ND-Range，它可以被拆分为一个Work-group工作组，这是第一个类型的组。接着，通过工作组，我们还能拆分出Sub-group子分组。注意，子分组他是一个一维的分组，非常有利于SIMD指令的执行，最后还能拆分成Work-item，也就是一个工作实例。

通过这种拆分，让合适硬件运算合适大小的分组，可以充分发挥硬件的性能潜力，包括内存方面和计算单元的分配方面。



好了，现在让我们看看ND-Range为何能发挥出硬件的性能优势。首先我选取了一块核显芯片的架构图，这个芯片比之前我们看到的GT2多了一块分片。但是请注意，如果没有充分发挥硬件的潜力是很难达到 $1+1=2$ 的效果的。我们可以看到3级数据缓存的下面是一个GTI，我们可以把它当成是GPU和SoC其余部分之间的通道，这个其余部分就包括LLC（last level cache）和增强或者普通的动态随机存取存储器。

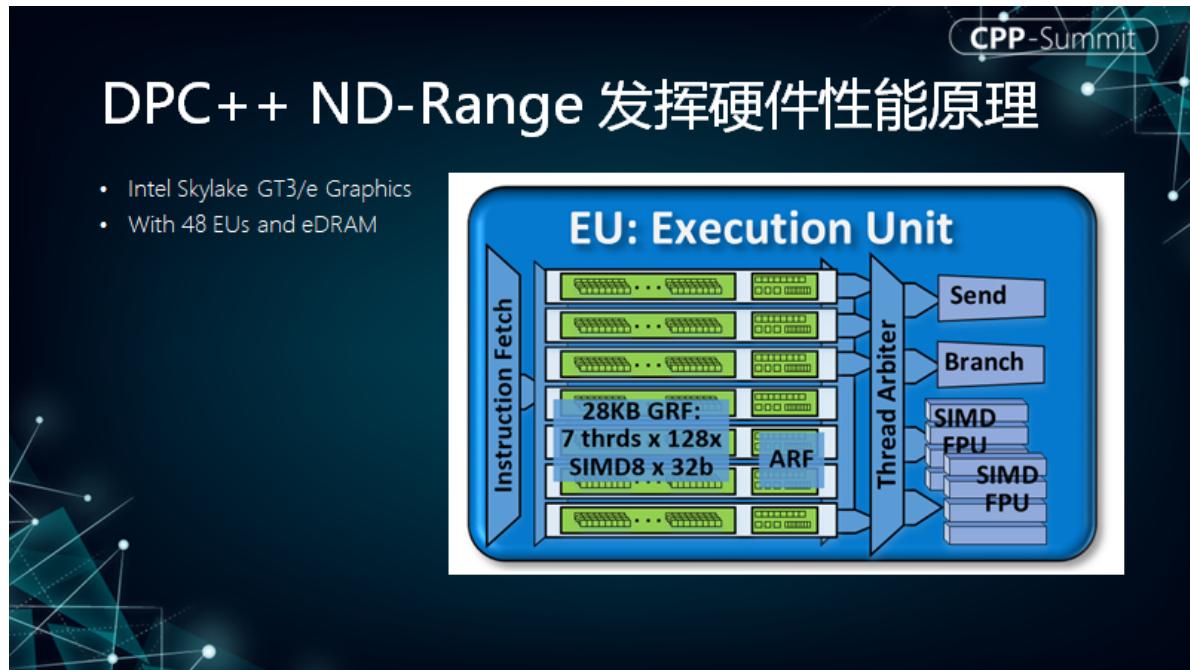
这张图上可以看到，将数据从GTI读取或者写入到3级缓存，一个时钟周期是64字节的数据，从GTI到其他部分存取数据也是类似的，需要一些时间。现在假设我们的发射给硬件设备的计算任务，需要跨越不同的分片，也就是说，数据需要通过GTI进行数据传输，那再这样的情况下，计算效率必然会降低不少。



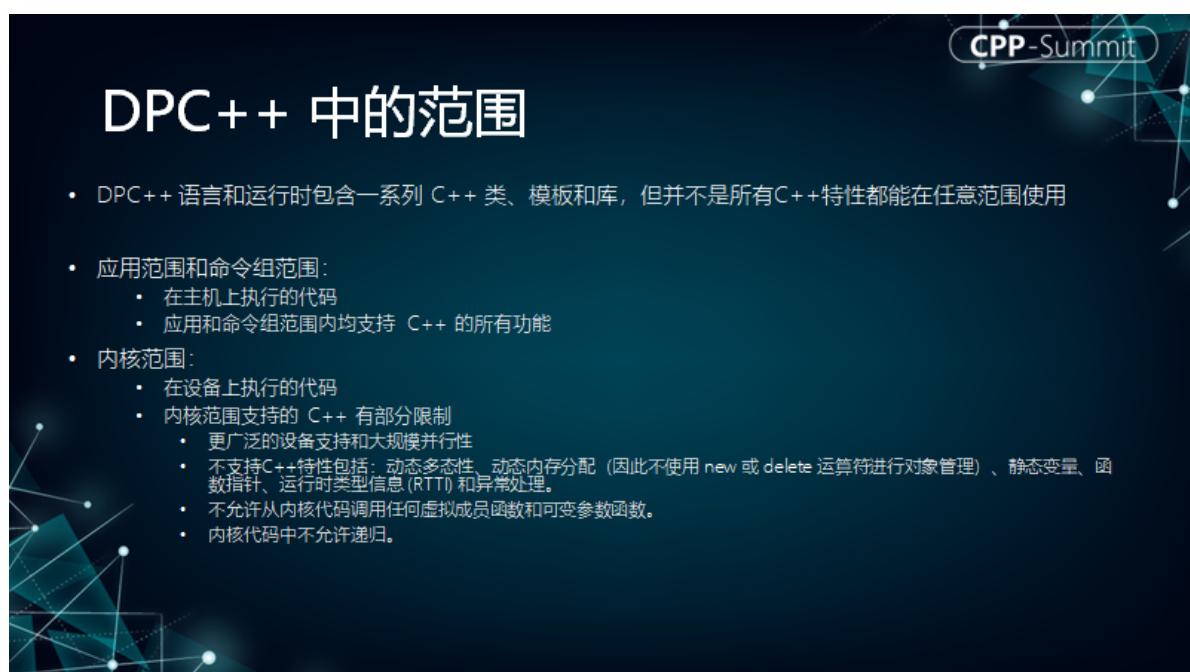
继续看到设备数据传输的大图，这张图对数据交换的流程进行了放大，而且描述也更准确。我们可以看到，三级缓存到LLC的读写速度是不一致，而LLC和不同的DRAM的数据交换速度也不同，特别是对于普通的DRAM，数据交互会慢很多。如果所有的计算动作都在一个分片里完成，那么就省去了很多对外交换数据的时间。

另一方面，每个分片还有子分片，子分配里一级和二级数据缓存，如果所有的计算动作都在子分片中完成，那么数据就不需要到三级缓存做交换，可想而知，速度肯定更加快。

这里其实还能再进一步，就是每个子分片里还有EU执行单元，执行单元的数据可以共享，也就是说不需要去1级缓存交换数据，如果一次计算能在一个EU中完成，那么效率比如非常高。



刚刚我们介绍了内存和数据传输对于效率的影响，接下来我们看到执行单元内部，来了解一下怎么分配计算任务来提高效率。图中我们可以看到，一个EU有28k的GRF通用寄存器文件，这个是怎么算的呢？实际上每个EU有7个线程，每个线程有128个32位寄存器，可以来执行simd8指令，并且同时发射4条指令。那么如果我们用ND-Range的分组，将计算实例合理的分配到每个执行单元的线程，那么理想情况下就能发挥加速设备的最大能力，当然这是理想情况。



在编码过程中有一点是必须特别注意的，虽然SYCL是基于C++，并且在运行时代码里包含了很多C++类、模板，但必须知道的是并不是所有的C++特性都能在任意范围使用的，这里的范围指的是内核范围。在SYCL中一般来说可以分为三个范围，分别是应用范围、命令组范围和内核范围。

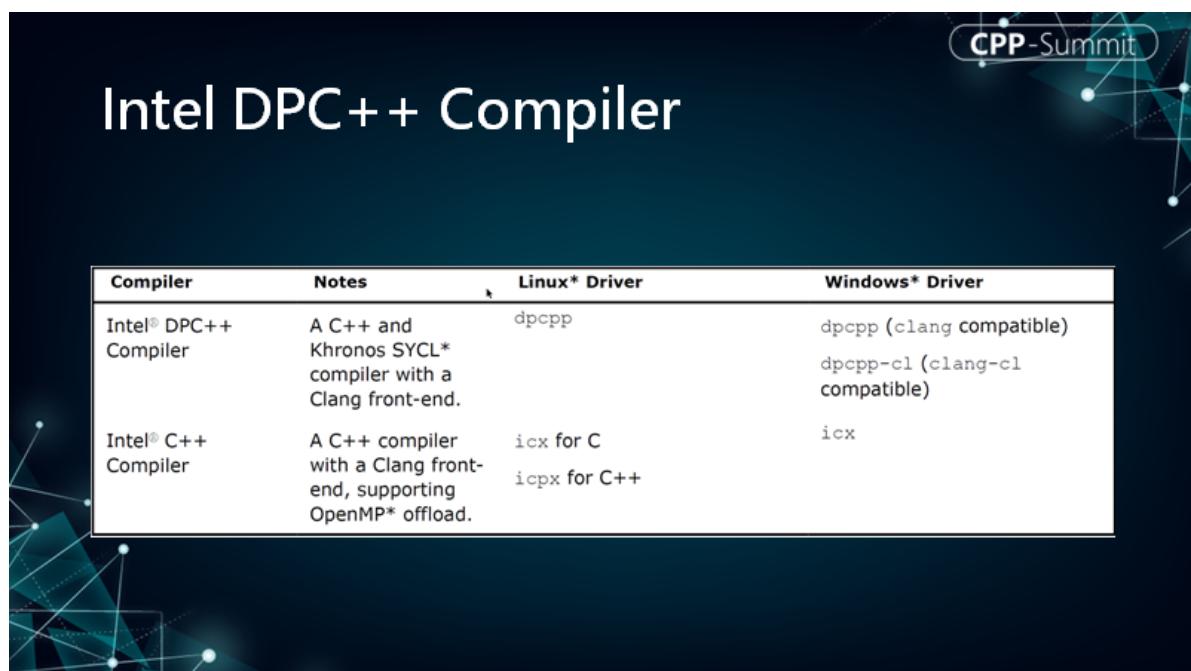
其中，应用范围是queue调用submit以外的代码，而命令组范围是submit中的lambda表达式的代码，内核范围当然是运行在加速设备里的代码。

因为应用范围和命令组范围都是运行在主机上，所以可以支持所有的C++功能。而内核代码就不一样了，因为会被转换为特殊的设备代码，所以有些C++功能是不能使用的，包括动态多态性、动态内存分配（显然我们不能使用 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，这些都是不允许的。另外还不允许在内核中调用虚拟成员函数和可变参数的函数，也不允许递归。

虽然看似有很多限制，但是影响并不大，毕竟在内核中大部分写的是算法本身，不会包括业务逻辑和架构设计，也用不到什么高级的特性，能完成基本算法就可以了。

## 编译设计

oneAPI中的SYCL编译器叫做DPC++，它是一个以LLVM和Clang为基础的开源编译器。



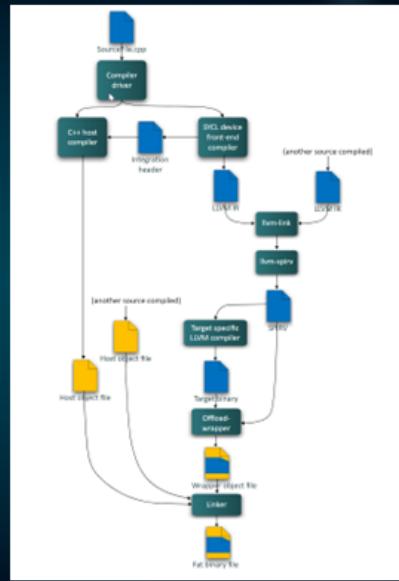
The image shows a slide from the CPP-Summit featuring the Intel DPC++ Compiler. The title is "Intel DPC++ Compiler". A table provides information about the compiler's support for Linux and Windows drivers.

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

需要注意的是，oneAPI中的编译器很多，DPC++编译器的驱动名为dpcpp，不要和其他编译器弄混淆。

# 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。



接下来我们来看看DPC++编译单一源代码的过程，直接看到右边的这张图。首先从源代码出发，经过编译器的驱动和前端，将单源代码拆分位C++的主机代码和设备代码两个部分，设备代码的编译前端会产生一个集成头文件，目的是让主机代码能顺利的编译通过，并且产生主机的object file，主机这部分的编译过程就不详细讲了，我们主要来描述设备代码的编译部分。

设备编译器前端编译了一个LLVM IR，并且通过LLVM Link程序，将多个IR文件链接在一起，通过LLVM Spirv编译成一个spirv文件，spirv是一个用于并行计算和图形计算的开放标准中间语言，这里就不做深入介绍。有了这个spirv文件，LLVM可以生成一个AOT的设备代码，当然也可以将spirv包装起来作为一个JIT的代码，最后通过链接器将主机的object file和设备的object file链接起来，生产所谓的胖二进制文件，也就是可执行文件。

总结一下三端做的事情，前端概述设备代码和主机代码，生成LLVM IR和集成头文件。中端，转换LLVM到spirv文件。后端，生产本机和设备代码，包括AOT和JIT的。

在编译过程中有很多有趣的细节，我这里挑出了两个来描述。

## 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

第一个有趣的细节是设备代码的单独编译链接，主要动机是让用户能够在更改仅影响主机的代码时节省重新编译时间。在生成设备映像需要很长时间（例如 FPGA）的情况下，这种节省可能非常显着。

例如，如果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，那么编译过程将可以分为三个步骤：

1. 设备链接：dev\_a.cpp dev\_b.cpp -> dev\_image.o（包含设备镜像）
2. 主机编译（c）：host\_a.cpp -> host\_a.o；host\_b.cpp -> host\_b.o
3. 链接：dev\_image.o host\_a.o host\_b.o -> 可执行文件

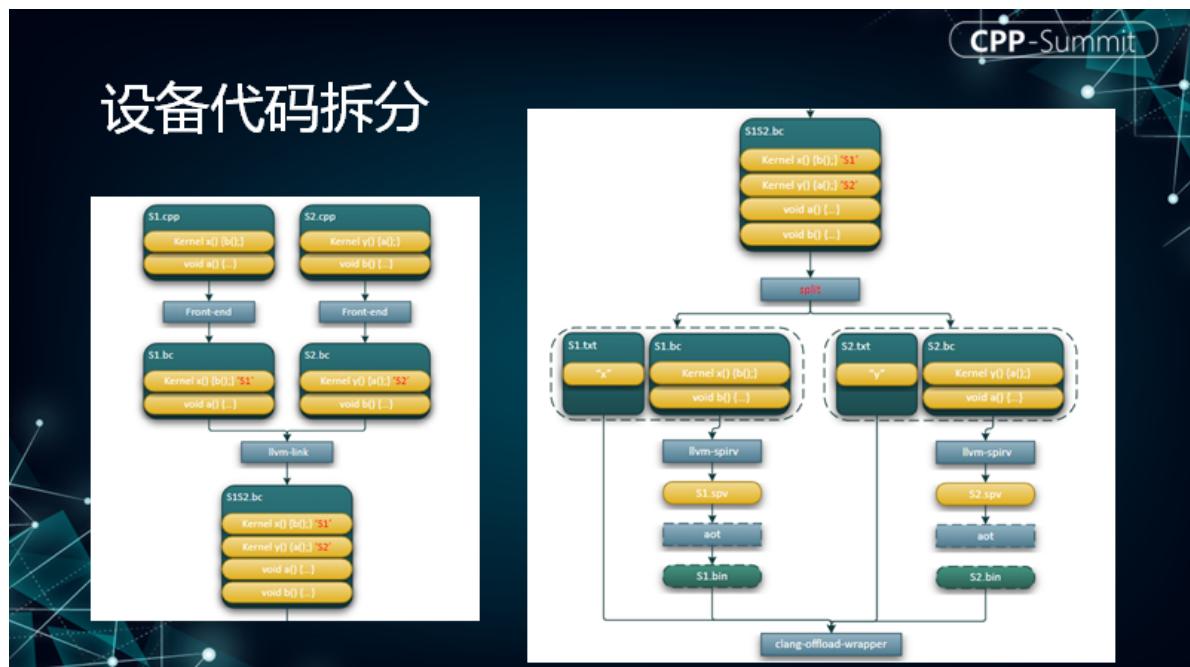
对于某些目标，步骤 1 可能需要数小时。但如果用户希望在仅修改 host\_a.cpp 和 host\_b.cpp 后重新编译，他们可以简单地运行步骤 2 和 3，而无需重新运行昂贵的步骤 1。



第二个细节是设备代码的拆分，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：

1. 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。
2. 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。

所以在这种情况下是很有必要对设备代码进行拆分的。拆分的方式有两种，第一种是简单粗暴的根据源代码生成单独的模块（翻译单元），第二种方法更加合理，根据单个内核生成单独的模块。



针对第二种方法，目前的做法是：

- 为 SYCL 前端的每个内核生成带有翻译单元 ID 的特殊元数据。此 ID 将用于按翻译单元对内核进行分组
- 使用 llvm-link 链接所有设备 LLVM 模块
- 在完全链接的模块上执行拆分
- 为每个生成的设备模块生成一个符号表（内核列表），以便在运行时正确选择模块
- 分别对每个生成的模块执行 SPIR-V 翻译和 AOT 编译
- 将有关内核的信息添加到每个设备镜像的包装对象

## DevCloud学习SYCL和DPC++

介绍完了SYCL的编程方法和编译流程，接下来将介绍通过使用DevCloud来学习SYCL编程。

The screenshot shows the Intel DevCloud landing page. At the top, there's a banner for the 'CPP-Summit'. Below it, a large section titled 'DevCloud简介' (DevCloud Introduction) is visible. Underneath, there are two main sections: 'Get Started with Intel® DevCloud' for 'OpenVINO' and 'Work with Intel® Distribution of OpenVINO™ Toolkit', and another for 'oneAPI'.

- 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>

在使用DevCloud做实验之前，还是得先介绍一下这个实验平台。DevCloud是Intel免费提供的一个云平台，通过这个云平台我们可以体验和学习Intel提供的各种框架，软件，工具。使用是完全免费的，申请也很简单很快，唯一不足的是国内访问速度有点慢。

The screenshot shows a guide titled '通过ssh连接DevCloud' (Connecting via ssh to DevCloud). It consists of three numbered steps:

- 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.** ▾  
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

DevCloud有两种使用方式，第一种用ssh直接连上去用，整个连接过程都有教程，操作很简单。这种方法就如同操作VPS，好处是编程的自由度很高，比较推荐已经学会一些SYCL编程知识的朋友来使用。

The screenshot shows a grid of six Jupyter Notebook modules under the heading "Learn the Essentials of Data Parallel C++". Each module has a "Try it in Jupyter" button.

- Module 0: Introduction to JupyterLab® and Notebooks**
  - Learn to use Jupyter notebooks to write, run code and edit code as part of learning exercises.
- Module 1: Introduction to DPC++**
  - Articulate how oneAPI can help to overcome the challenges of programming in a heterogeneous world.
  - Use oneAPI solutions to enable your workflow.
  - Understand the DPC++ language and programming model.
  - Become familiar with using Jupyter notebooks for training throughout the course.
- 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.
- Module 3: DPC++ Unified Shared Memory**
  - Use DPC++ Unified Shared Memory (USM) to simplify programming.
  - Understand implicit and explicit ways of moving memory using USM.
  - Explain data dependency between kernel tasks in an optimal way.
- Module 4: DPC++ Sub-Groups**
  - Understand the advantages of using Sub-groups in DPC++.
  - Take advantage of Sub-group collectives in ND-Range kernel implementation.
  - Use Sub-group Shuffle operations to achieve more memory operations.
- Module 5: Demonstration of Intel® Advisor**
  - Identify and rectify parallelization opportunities for offload.
  - Run Offload Advisor using command-line syntax.
  - Use performance models and analysis generated reports.
- Module 6: 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.
- Module 7: SYCL Library Utilization**
  - Increase productivity with data comparison to Intel® oneAPI DPC++ Library Utilization as an alternative for C++ descriptors.

第二种方式是使用jupyter notebook来使用DevCloud，我们可以看到右边这张图，通过这种方式可以非常系统的学习oneAPI和SYCL，这里的每篇教程都是图文介绍，还有示例代码可以直接运行，对于初次接触oneAPI的朋友来说相当友好。

## 总结

随着科学技术的发展，我们发现如今的应用对算力的要求越来越高，而且未来还会越来越高。单一的处理器显然已经远远不能满足巨量的计算需求。而我们现在似乎已经找到了一条不错的应对大量算力需求的方案，它就是异构计算。但是使用异构计算最大的困难在于针对不同的硬件需要不同的编程方法，而oneAPI和SYCL的出现有望解决这类问题，通过编写标准的C++代码让我们程序能够高效的运行在各种处理器设备上。