# 第3章 并行计算与GPU架构

深度学习模型的训练和推理需要大量的算力支持，而以GPU为代表的并行计算则提供了强大的计算能力。深度学习和并行计算相辅相成，相互促进，最终达到了深度学习当前的繁华局面。虽然现在开始出现了很多用于深度学习的专用硬件，但是，基于GPU的Vulkan和OpenCL等业界标准在深度学习中还是扮演着极其重要的角色，为深度学习提供着强大的算力支持。

所以，本章首先将简要介绍并行计算的发展和基本概念，然后介绍Intel GPU的计算架构，以及在深度学习的加速实现中比较常用的cl\_intel\_subgroups在Intel GPU上的参考实现。接下来的第4章和第5章将分别介绍OpenCV深度学习模块中基于Vulkan和OpenCL的加速实现。

## 3.1 并行计算浅谈

在计算机早期时候，任务只能被依次串行执行。随着计算机技术的发展，操作系统开始引入进程和线程的概念，此时，虽然在CPU上还是按时间顺序串行执行，从用户角度来说，任务已经可以被并发的处理。所以，提高CPU性能非常重要的一个方法就是提高处理器的时钟频率，只要频率能大幅增加，处理器在单位时间内就可以执行更多的指令，从而处理器性能就自然而然的大幅度提高了，但是，随着物理极限的临近，这种方法开始受到限制。另外一个方法是指令的多段流水线执行，通过将一条指令的执行过程拆分成多个阶段，比如取指、译码、执行、访存和写回阶段，从而实现同时执行多个指令的不同阶段的并行效果，但是，由于指令间的依赖关系，以及条件分支等指令的存在，也不可能通过无限制的拆分流水线来提高性能。于是，为了持续地提高计算能力，诸如超线程、多核处理器和向量化指令等支持并行计算的更多技术就开始发展起来了。

大致来说，所谓并行计算，就是将一个任务拆成多个子任务，然后这些子任务可以在同一时刻被计算执行。一个典型的例子，就是GPU中3D graphics的顶点着色（vertex shader）和片段着色（fragment shader）。在顶点着色中，每个顶点被独立地计算处理；而片段着色中的每个像数也是相互独立的，可以同时计算处理。所以，自从GPU引入了可编程模块后，就自然的支持了并行计算。也因此，GPU在一开始就成为了深度学习模型训练的首选硬件平台。

并行计算大致可以分成以下几类：

1） 分布式并行

任务分解后，通过网络将子任务分配给多台计算机同时执行。比如，寻找梅森素数（MersennePrimes）[[1]](#footnote-2)就是一个这样的例子。如果p是一个素数，记Mp = 2^p – 1，如果Mp也是一个素数的话，则称其为梅森素数。有很多数学家都投身于梅森素数的寻找中，为了在互联网时代更好地寻找更多的梅森素数，提出了GIMPS计划，即Great Internet Mersenne Prime Search（梅森素数互联网大搜索）。只要进入GIMPS网站，下载Prime95软件包，解压后在自己机器上运行，就可以贡献一份个人的力量。每台机器都会被分配不同的Mp数值，全球参与计划的机器以数据并行的方式进行分布式合作，以完成搜索目标。

2） 多处理器并行

在一台机器中存在多个处理器，比如，一台服务器中可能有多个CPU socket可以插入多个CPU处理器。也可能是一台机器中存在CPU、GPU、FPGA和DSP等多种处理器，任务被分配到这些不同类型的处理器上同步进行，这种基于不同类型处理器的并行计算被称为异构并行计算。

OpenCL是一个针对异构计算而提出的通用计算API，支持不同类型的计算设备，向上层软件开发者提供可移植的高效接口。OpenCL用platform的概念来表示这样的一台机器系统，包括host设备和若干计算设备，host设备可以理解为正运行着操作系统的CPU，可用于分配任务；而计算设备则包括CPU、GPU和DSP等计算资源。OpenCL用DeviceID来抽象表示这些计算设备，而用device\_type来表示设备的类型。为了从host端将任务传送到计算设备中，OpenCL为DeviceID抽象了CommandQueue的概念，这样，任务命令可以被host发到这个queue中，而计算设备则从queue中获取命令再执行。为了多个命令在API层的同步，OpenCL还提出了Event的概念。针对最重要的计算命令，OpenCL用kernel和program进行抽象，kernel会被发送到command queue中，再被相应的计算设备执行。最后，计算设备要访问的存储空间被抽象为Buffer和Image，并用Sampler这个概念对Image进行采样。

目前，一个OpenCL实现（即硬件设备加上配套的OpenCL软件驱动程序，包括OpenCL kernel编译器等）往往只支持一种计算设备，即使系统中存在多个不同类型的计算设备。一个可能的解决方法，是在platform层加载各种计算设备的OpenCL接口，而且，每个计算设备都汇报自己的计算能力，使得用户可以将任务分解分配到最合适的计算设备上，从而最有效地用好整个系统的计算能力。

3） 多核并行

多核一般是针对CPU而言，在服务器上比较常见，比如可能有百多个逻辑核心。只有在多CPU多核的前提下，进程（线程）才能被真正的并行执行，因为不同的线程可以被分配在不同的核上，每个线程可以独占一个CPU核。一般来说，每个线程应该尽量被分配在一个CPU核上不再切换，如果发生切换的话，会影响性能。在Linux中，可以用taskset命令将某个进程绑定到某个CPU核上，也可以在代码中用setaffinity系列函数将进程或线程绑定到某个CPU核上。在第6章会有更多关于多线程和CPU指令的介绍。

在多核环境下，如果性能不佳，首先要检查的是所有CPU核的使用情况，可以用第7章介绍的top和VTune等工具查看是否所有CPU核都已被有效地利用上了。

4） 指令级并行

指令级并行发生在处理器核的内部。指令有多种类型，比如访存指令、计算指令和分支指令，在Intel GPU中，这些指令是由物理上不同的指令部件实现，这样，多个不同类型的指令就可以并行执行。同一类型的指令，还可以用流水线技术，将指令的完成分解为多个阶段，以实现同时执行多个指令的效果。另外，还有一个重要做法，是指令的向量化，即用一条指令来处理多个数据，也就是用单指令多数据SIMD（Single Instruction Multiple Data）的方法，来达到并行的效果。

一个SIMD部件的简单示意如图3-1所示，这里展示的是SIMD-8，即可以同时处理8路数据。图中正在执行的是“add c，a，b”，这是一条指令（单指令），a、b和c是操作数（这里省略了数据类型），在实际指令中可能分别对应着一个长度为256比特的寄存器（256比特可以保存8个float数值），为了描述简单，不妨将它们分别看作一个长度为8的数组。SIMD部件在执行这一条指令的时候，会对数组中的所有元素（多数据）同时做加法操作。所以，长度为8的两个数组对应元素相加再存入第三个数组相应位置，用一条SIMD-8的加法指令，就可以并行的完成了。要用好SIMD指令，就需要准备好向量化的操作数，而如何构成这样的操作数，到底是用户在写代码的时候就要准备好，然后编译器进行协助；还是用户无需关注，完全由编译器来完成，这涉及到AOS（Array Of Struct）和SOA（Struct Of Array）这两种不同的思路，将在后面详细展开介绍。



1. SIMD-8示意图

## 3.2 Intel GPU架构及其在并行计算的应用

本节将介绍Intel GPU硬件的计算架构及其在并行计算中的应用，中间会穿插OpenCL的实现手段和优化建议。希望读者有一定的OpenCL基础，可以先看一下5.1节，了解work item和work group的概念，知道共享本地存储（shared local memory），也知道全局存储（global memory）和私有存储（private memory）等概念。

我们首先介绍Intel GPU计算架构，然后介绍AOS（array of struct结构体数组）和SOA（struct of array数组结构体）这两种不同内存布局下的并行计算思路，解释OpenCL是如何在Intel GPU上以SOA方式实现，并简要分析其特点；最后，在上述知识的基础上，介绍机器学习中很常用的OpenCL subgroup extension在Intel GPU上是如何实现的，以帮助读者更好地理解GPU的工作过程，有助于理解基于OpenCL的OpenCV深度学习模块的加速实现。

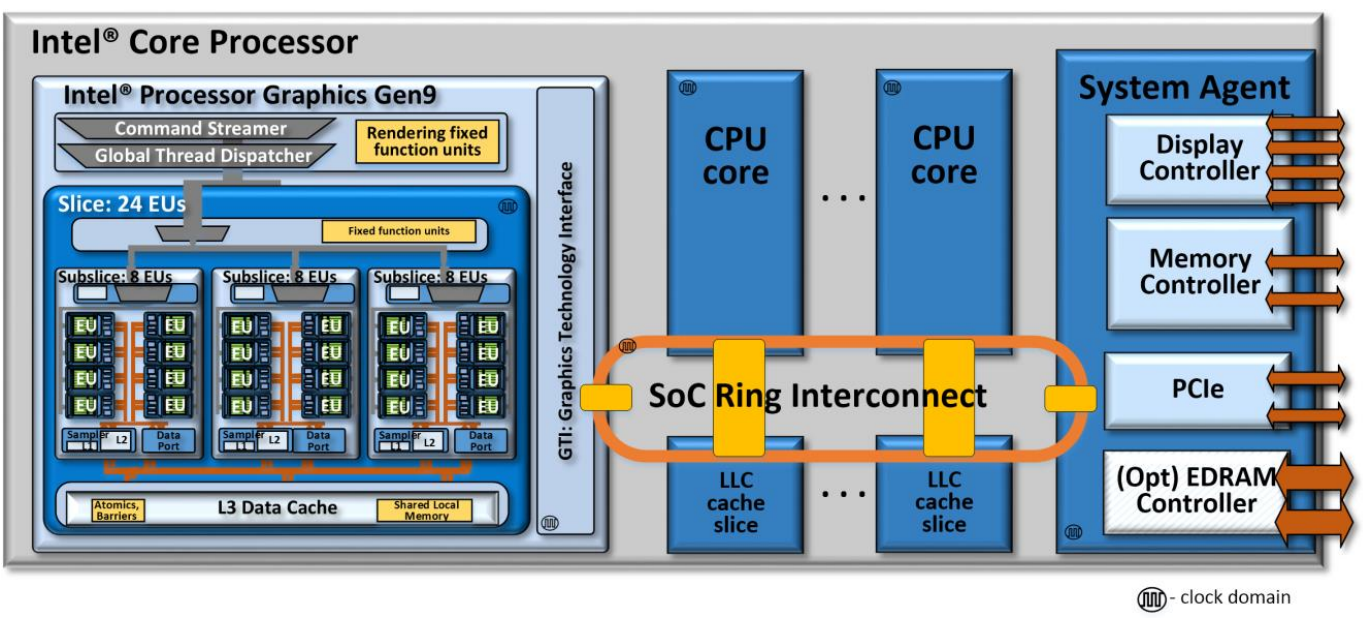
### 3.2.1 Intel GPU的计算架构

从20世纪的3D加速卡开始，显卡芯片经过了爆炸性的发展，NVIDIA公司将芯片命名为GPU（Graphics Processor Unit，图形处理单元），而ATI公司（后被AMD收购，拉开了将CPU和GPU集成到一个chip中的序幕）则命名为视觉处理单元（Visual Processor Unit，VPU），最终GPU这个名称胜出。传统上，一个GPU芯片包括显示引擎（display engine）、三维引擎（3D engine）、二维引擎（2D engine）和视频编解码（video codec）。其中，显示部分用于连接显示器，将帧缓存（frame buffer）中的数据传给显示器；而3D部分则用于三维图形的渲染，并逐渐发展出可编程模块，比如OpenGL中的GLSL（OpenGL Shading Language）和Direct3D中的HLSL（High Level Shading Language）等；2D部分则用于二维图像诸如菜单等的绘制，随着Windows Vista（内部代号Longhorn）的新设计出现，二维菜单可以直接用3D部分来完成，因此2D部分已被逐渐舍弃；视频编解码部分是视频处理模块，包括固定管线的硬件设计，也包括使用可编程单元来进行编解码工作。

更进一步的，可编程模块被用于更多的计算领域，也引出了GPGPU（General Purpose Computing GPU，通用计算GPU）的概念，比如NVIDIA公司推出的CUDA，Khronos组织在维护的业界标准OpenCL等。而最近人工智能浪潮的发展表明，GPU的并行计算特性，很好地契合了深度学习的训练和推理过程的数学计算，已使GPU成为机器学习加速的最重要的硬件平台。

#### CPU和集成显卡之间的联系：SoC

Intel GPU和CPU被集成在一个片上系统（system-on-a-chip，SoC）中，以经典的第六代酷睿Skylake为例（公开的GPU资料最为详细），如图3-2所示。整个SoC叫做Intel Core Processor，包括系统代理（System Agent）、CPU、GPU（Intel Processor Graphics Gen9），最末级缓存（Last Level Cache, LLC）和片上系统环连接（SoC Ring Interconnect）。其中，环连接（Ring Interconnect）是个双向环形总线，连接其他各组成部分；系统代理中包括display模块、内存控制单元、PCIe总线控制器等I/O接口；CPU部分可以扩展多个CPU core形成多核CPU；所占面积比例最大的是GPU，即图中左侧的Intel Processor Graphics Gen9。Gen9是内部开发代号，表示第9代（generation）的意思，Intel用简单的Gen8、Gen9、Gen9.5、Gen9 LP（low power）和Gen11等内部代号指代各种GPU型号，最后对应着市场上诸如HD graphics 530、Intel Iris Graphics等各种GPU产品。说明一下，这里的第几代是GPU内部代号，和酷睿第几代没有直接关系，第X代酷睿中可能集成的是第Y代GPU（即GenY）。



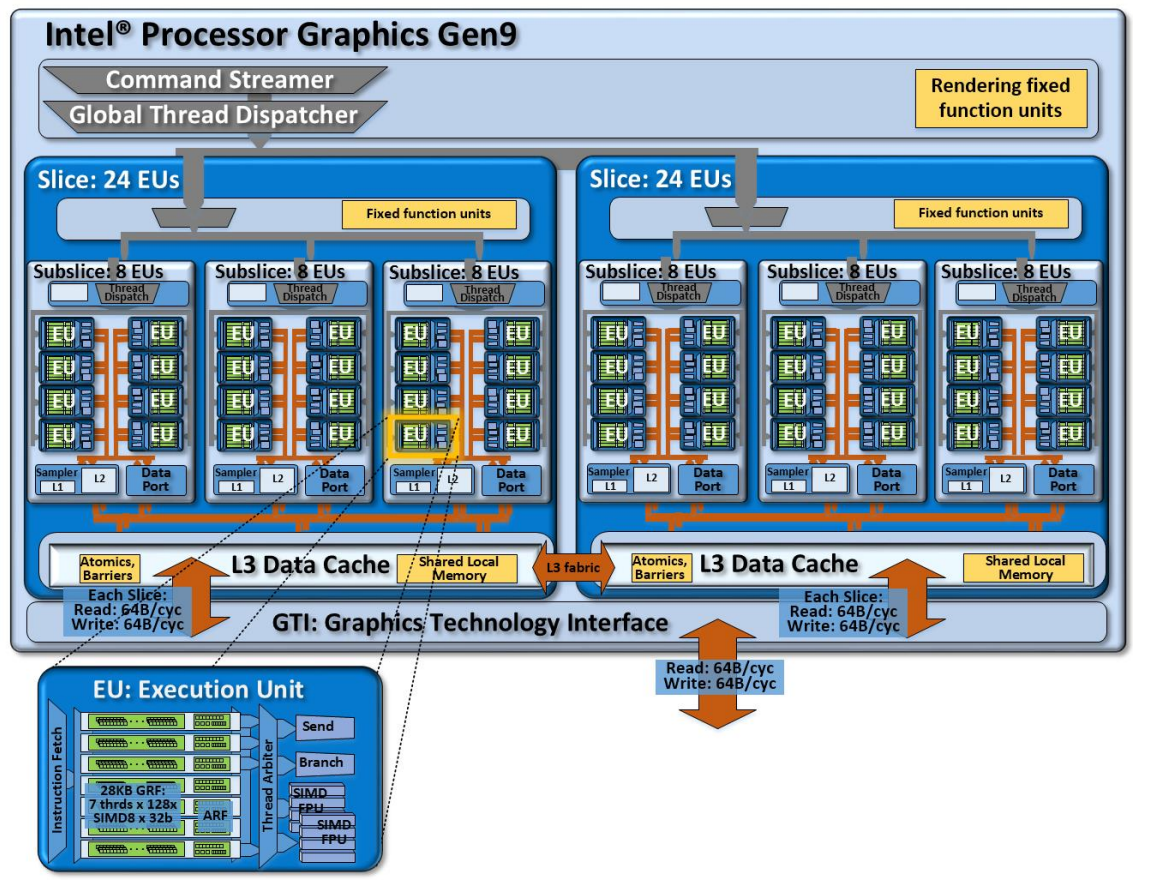
1. Intel SoC各组成部分示意图
2. [[2]](#footnote-3)

从图3-2可以看出，Intel GPU中并没有片内存储（on chip memory，或者local video memory，也就是俗称的显存），GPU所有的访存操作都依次经过图形技术接口（Graphics Technology Interface，GTI）和环连接（SoC Ring Interconnect）再经过系统代理（System Agent）中的存储管理器（Memory Controller），最终访问内存（DRAM，即俗称的插在主板上的内存条）；而CPU 核心的访存操作，也经过了环连接，再经过系统代理中的存储管理器，最终访问内存，这使得Intel GPU可以很方便地和CPU共享物理内存，即统一内存架构（Unified Memory Architecture）。对于给定的物理内存，通过CPU 存储管理器（Memory Management Unit，MMU）可以映射为CPU地址，通过GPU类似部件可以映射为GPU地址，不同的CPU地址和GPU地址因此被映射到同一块物理内存，这样无须拷贝，即可实现CPU和GPU之间的内存共享，也称作零拷贝（Zero-Copy）技术，在OpenCL中，可通过传入标志CL\_MEM\_USE\_HOST\_PTR或CL\_MEM\_ALLOC\_HOST\_PTR来创建支持零拷贝的buffer。

这种技术进一步发展，可以使得CPU地址和GPU地址也相同，从而使得CPU和GPU之间可以共享具有更加复杂数据结构（比如链表）的数据，此即OpenCL Spec 2.0中提出的共享虚拟内存（shared virtual memory，SVM），一种可能实现手段，就是在内核态为CPU和GPU的MMU配置相同的虚拟地址到物理地址的映射关系，从而实现相同的CPU地址和GPU地址。CPU/GPU访问内存时，还可以缓存到最末级缓存（last level cache，LLC）中，LLC不只是在存储器结构层次中增加了一层，也为CPU和GPU之间共享内存发挥作用，LLC在大部分产品中都存在，但有些硬件平台比如Baytrail中不存在LLC，存储读写性能因此有所影响。这种没有显存、使用系统内存的区别于独立显卡的技术，当GPU被集成在主板芯片组中的时候，被称为集显（集成显卡），当GPU和CPU一起集成在SoC中的时候，被称为核显（核心显卡）。

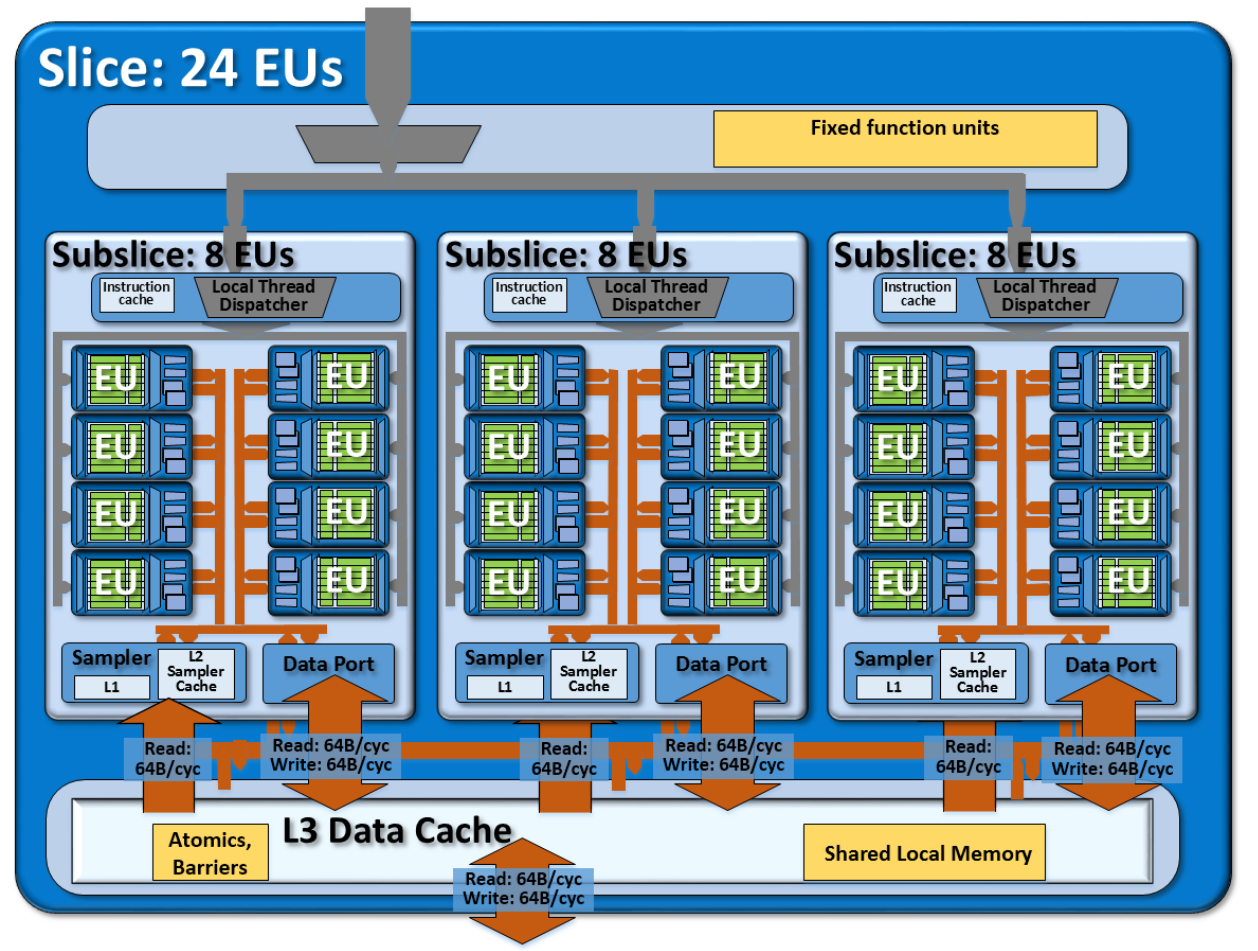
#### GPU的分片结构：Slice和Subslice

接下来，从计算架构的角度，看一下GPU的内部组成，如图3-3所示，这是基于Gen9的一个衍生产品规格，其中，Command Streamer读取并分析来自驱动软件的command buffer，然后将command传送给GPU内部相应的硬件模块，对计算相关的work load，则通过全局线程分发器（ Global Thread Dispatcher）被传送至Slice中，再通过Subslice中的Local Thread Dispatcher被传送到最终的执行单位（Execution Unit，EU）中，成为EU中的硬件线程（hardware thread，后面偶用线程来简述）。图3-3中包括2个Slice，每个Slice由3个Subslice组成，而每个Subslice包括8个EU，每个EU中可以支持7个硬件线程，而且EU中算术逻辑单元（ALU）由单指令多数据（SIMD）构成。通过配置不同数量的Slice，调整Slice中Subslice的数量，调整Subslice中EU个数，以及EU中的线程数量，可以衍生出覆盖从低端到主流再到高端的全系列GPU配置，比如GT2，GT3，GT4e（e表示配置了EDRAM，后文将有介绍）等，数字越大表示性能越强。



1. GPU内部组成图2

继续深入Slice和Subslice的内部，如图3-4所示，图中的L3 Data Cache除了作为正常的缓存使用外，还包括Atomics、Barriers和共享本地存储（Shared Local Memory，SLM）等功能，对应着OpenCL中工作组（work group）相关的概念。Atomics可以支持OpenCL内核中atomic\_add/sub等函数的实现，如果参数是global memory的话，还需要GTI的支持。Barrier和SLM被分配到每个Subslice中，不同Subslice中的barrier和SLM是相互独立的。所以，如果在OpenCL内核中用到了barrier或者local memory的话，一个work group中的所有work item必须被分配（dispatch）到同一个Subslice中，不可以跨Subslice存在。



1. GPU Slice和Subslice示意图2

在Gen9中，每个Subslice有16个barrier，因此，如果OpenCL内核中用到了barrier的话，每个Subslice中最多可以被dispatch进来16个work group，因为barrier是从属于work group的同步方法。当然，因为每个Subslice中有多个EU，有较大的计算能力，所有，每个work group中的work item不应该太少，否则，会浪费Subslice中的算力。在Gen9中，每个Subslice有64K字节的专用SLM，OpenCL中local memory的大小不可以超过每个Subslice能拥有的SLM的容量。在关于local memory的读写设计上，由于其在L3 cache中以bank的形式组织，所以还要考虑bank冲突问题。SLM以四个字节（DWORD）的粒度为单位进行存储（banked），一共有16个bank（没有统一的中文翻译，因此后续继续使用bank这个英文单词）。例如，第1个DWORD被存在bank0，那么，第2个DWORD被存储在bank1，以此类推，第16个DWORD被存储在bank15，然后，第17个DWORD又被存在bank0，第18个DWORD被存储在bank1，依此类推。

如果我们要访问的local memory出现在同一个bank中的话，那就存在冲突，会导致读写性能变差，一些常见例子如代码清单3-1所示。可以在看完下一节了解work item是如何被映射到SIMD中后（即：一个work item被映射到SIMD的一个lane上），再回头来看这些例子，就能更好的理解。如果我们可以用好local memory，在多个work-item间共享数据 ，则可以节约内存总线的带宽，这对于I/O敏感的任务特别有效。

1. 使用local memory的一些常见例子

\_\_local uint \*buffer;

// 没有bank冲突

uint x = buffer[get\_local\_id(0)];

// 没有bank冲突

uint x = buffer[get\_local\_id(0) + 1];

// work item (id) 和 (id + 8)冲突，浪费一半的带宽

uint x = buffer[get\_local\_id(0) \* 2];

// 最差情况，所有被访问数据都在一个bank里，浪费了15/16的带宽

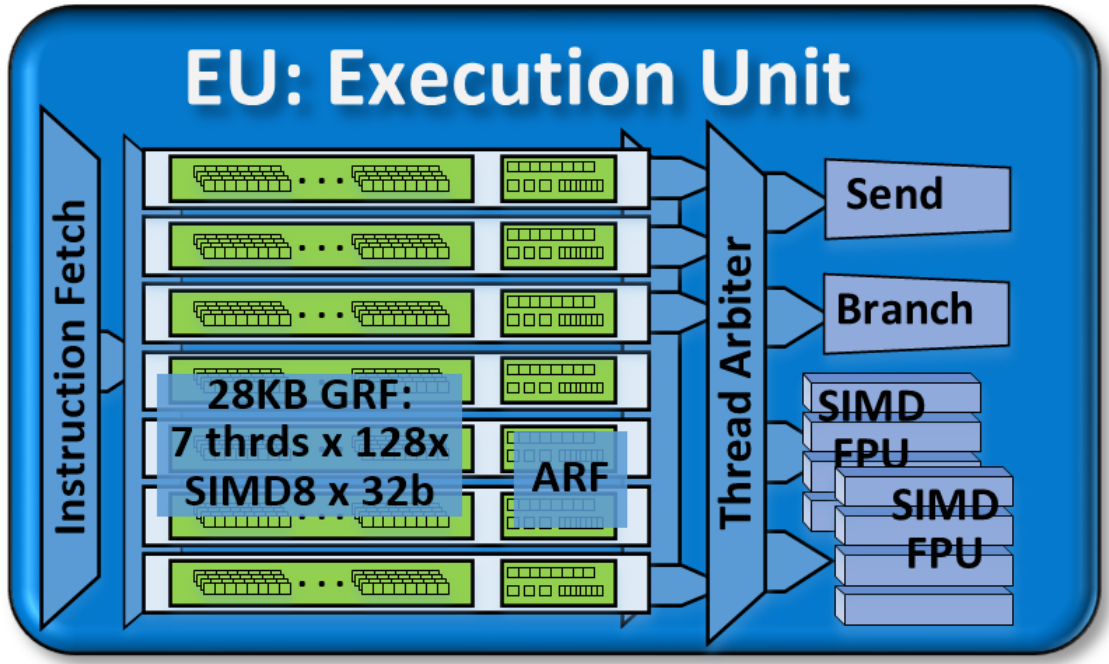
uint x = buffer[get\_local\_id(0) \* 16];

在Subslice中，还有Sampler和Data Port。Sampler用来存取OpenCL中的image（即由clCreateImage函数创建），支持各种filter方法；Data Port用来存取OpenCL中的buffer（即由clCreateBuffer函数创建）。

#### 执行单元

继续深入，到了探究EU（Execution Unit，执行单元）的时候了。为了更好地提供并行能力，EU实现了线程级并行和指令级并行的方法，如图3-5所示。线程依次经过Global Thread Dispatcher和Local Thread Dispatcher后进入EU，Gen9的EU可以容纳7个硬件线程，EU中的Thread Arbiter（线程仲裁器）选择就绪的线程，进入右侧的4个指令部件（发送Send、分支Branch和两个独立的SIMD 浮点计算单元FPU）中执行，这4个指令部件可同时执行来自4个线程的指令。当指令部件上的某线程处于等待状态，就会被Thread Arbiter换成另一个就绪线程，这个思路和操作系统中的进程/线程管理是类似的。

Send部件用来发送消息（Send Message）到GPU的其他模块，主要完成诸如访存、Sampler等耗时的操作。Branch部件用来处理跟踪指令中的分支和汇合（divergence和convergence）。SIMD FPU的宽度为4（即SIMD有4个lane，可记为SIMD-4，为避免混淆，后文我们将直接使用lane这个英文单词），每个lane每个cycle（机器周期）可以完成一次float32的MAD（乘法和加法）操作。所以，理论上每个EU每个cycle可以完成 2（mad包括乘法和加法两个操作）\* 4 个lane \* 2 FPU = 16次浮点操作（即16 FLOP/cycle）。两个SIMD FPU还支持整数操作，其中一个SIMD FPU还支持诸如sin、cos等复杂数学运算和double操作。



1. EU结构示意图2

EU中每个硬件线程都有两类寄存器，ARF（Architecture Register File架构寄存器）和GRF（General Register File通用寄存器），不同线程之间的寄存器相互独立，各管各的。ARF保存线程的运行状态，比如当前执行到了哪条指令（instruction pointer）；GRF是通用寄存器，在OpenCL内核编译器生成的指令中，就会用到这些寄存器。在Gen9中，每个线程有4K字节的GRF（也就是说，每个EU一共有4KB\*7个线程=28K字节的通用寄存器），每个通用寄存器是256比特，所以一共有4K \* 8比特/ 256比特 = 128个通用寄存器，记为g0、g1、g2，直到g127。寄存器支持多种数据类型，比如unsigned byte、int32、half float、float和double等，通过增加后缀表示寄存器中数据的数据类型，比如g1:D表示r1寄存器中的内容被解析为8个DWORD，用g5:DF表示r5寄存器被解析为4个double型的数据，而g100:F则表示g100中有8个float32的数据。通用寄存器的访问非常灵活，通过更多的标记手段，可以支持多个寄存器组成位数更宽的寄存器，甚至支持多个寄存器组成块，并按块和外部存储器交换数据，这方面展开的细节比较繁琐，而且对理解Intel GPU计算架构的帮助并不是很大，所以，略之。只需记住通用寄存器的访问方式非常灵活即可。

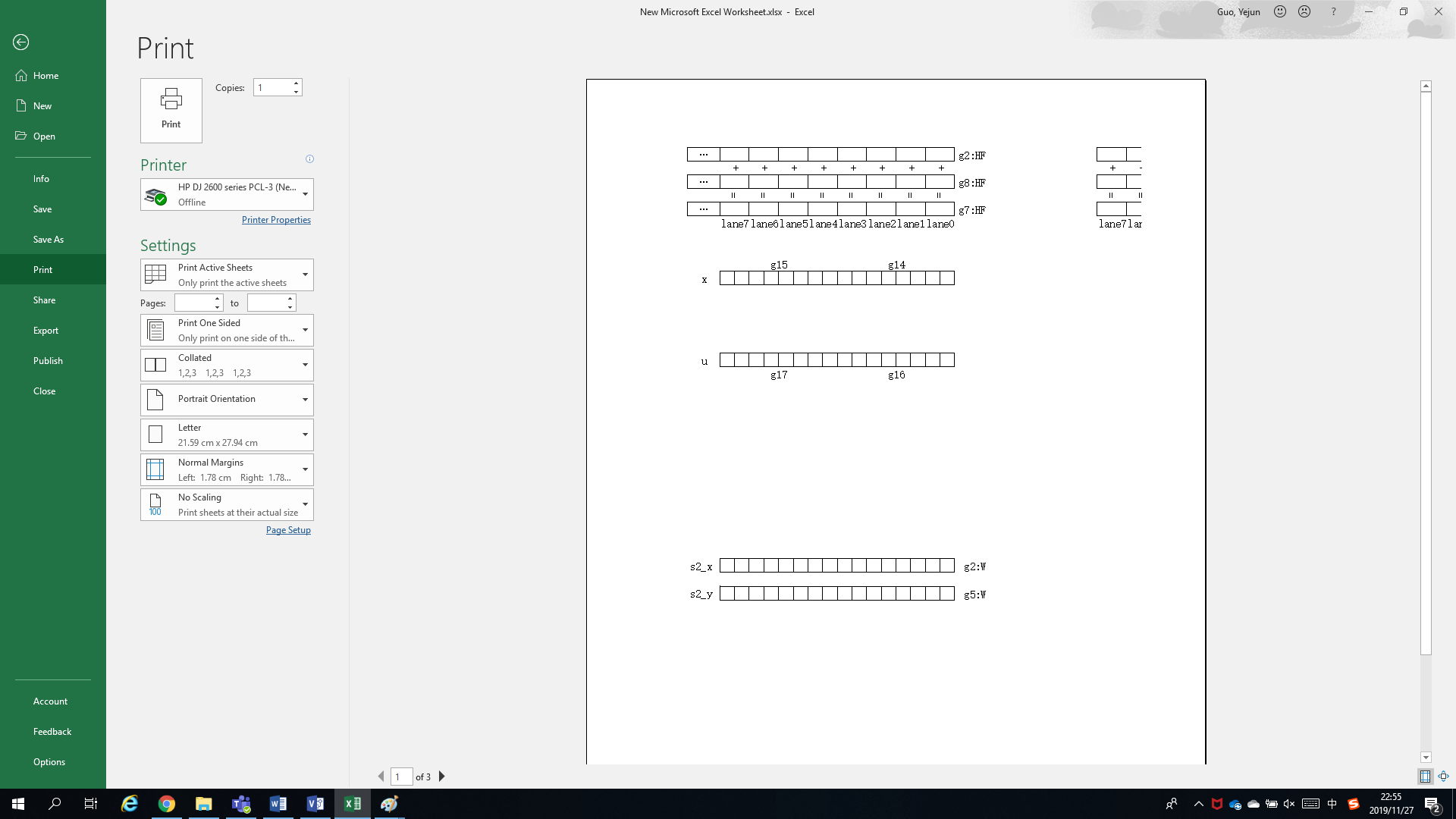
EU的硬件指令部件FPU的SIMD宽带为4，但是对外（对驱动软件、编译器等）能提供多种SIMD宽度，比如SIMD-1、SIMD-2、SIMD-4、SIMD-8、SIMD-16和SIMD-32等，EU内部硬件会自动转换到硬件SIMD-4中执行，当然实际需要的cycle数量会相应的变化。对编译器来说，就好像EU直接提供了SIMD-n（宽度为n，一共有n个lane）的指令部件，而且EU也在逻辑上直接提供了SIMD-n的线程支持。所以，当我们继续后续讨论的时候，除非特意说明这是基于物理上的硬件指令部件，将假设认为硬件中存在SIMD-n指令部件，以方便简化讨论，而且事实上也确实可以基于这个抽象层次来理解。从裸机到操作系统再到应用程序，就是一层层的抽象，抽象是非常强大的工具，可以帮助我们屏蔽暂时无关的细节，只需关注相应抽象层的接口，以更好的理解世界。在这里，SIMD-n就是一个恰当的抽象级别。

一个SIMD-8的半精度（half float）加法的汇编指令（GPU的汇编指令，不是CPU的汇编指令），可以如下表示。

add(8) g7:HF, g2:HF, g8:HF

注意：前面提到过通用寄存器的表示方法比较复杂，这里的表示方法省略了很多必备元素，只是做个简单示意，后文我们也将采用这种简单示意的方法。

其中，add表示加法指令，后面的括号表示SIMD的宽度为8（对应着lane0到lane7），:HF表示寄存器的数据类型都是16位的half float（半精度浮点数，或者称为half precision），所以，这条指令是将寄存器g2中的低128比特中的8个float16数值，加上g8中的低128比特中的8个float16数值，放到g7的低128比特中，如图3-6所示。

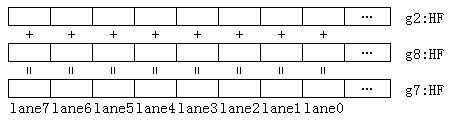


1. SIMD-8半精度加法指令示意图（低128比特）

做个简单扩展，用下面汇编指令实现寄存器高128比特的加法。

add(8) g7.8:HF, g2.8:HF, g8.8:HF

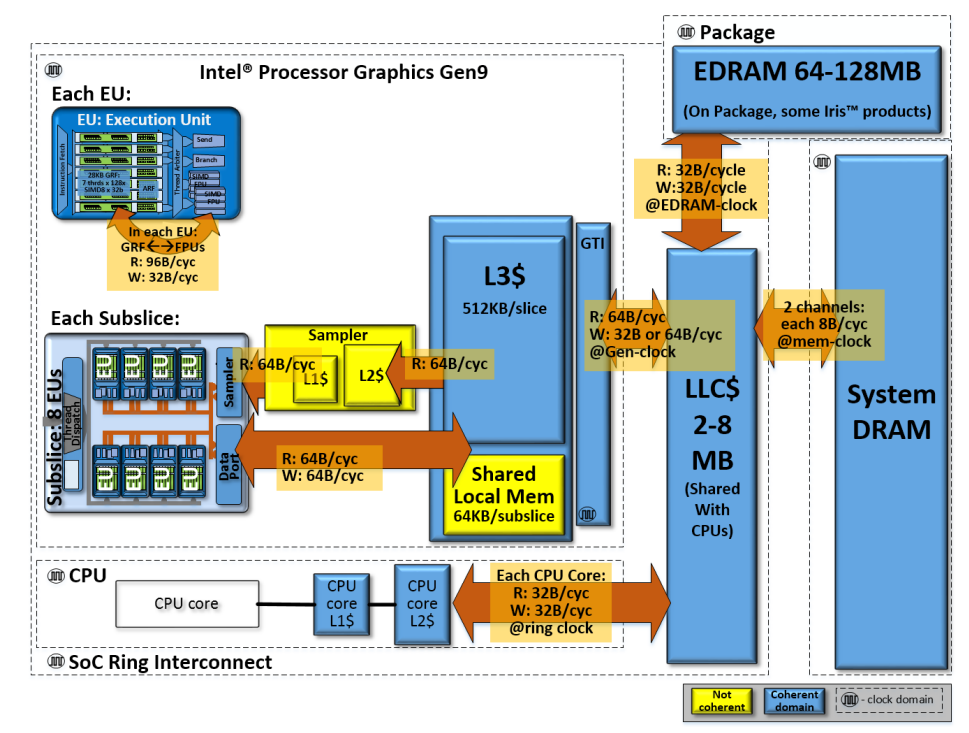
其中的.8表示起始的offset，从此寄存器的第8个HF开始，如图3-7所示。在上个例子中的g7:HF则可以理解为g7.0:HF，由此也可以看出寄存器的访问是非常灵活的。当然，这里的寄存器表示方法也只是一个简单示意，并非正式写法。



1. SIMD-8半精度加法指令示意图（高128比特）

#### GPU存储层次

最后，来看一下Gen9中的存储层次结构，如图3-8所示，在EU内部，FPU和通用寄存器之间的传输速度最快，每个cycle可以读96字节、写32字节，之前提过，EU中有两个SIMD FPU指令部件，每个FPU的SIMD宽度为4，每个cycle可以执行一次mad操作，所以，EU在每个cycle可以执行 2个FPU部件 \* 4个lane =8次mad操作，而mad操作有3个源操作数，1个目标操作数，以典型的浮点操作为例子，每个操作数是float型占4个字节，所以，每个cycle需要读取 8次mad操作 \* 3个源操作数 \* 4字节 = 96字节，每个cycle需要写 8次mad操作 \*1个目标操作数 \* 4字节 = 32字节，这和FPU与寄存器的读写性能刚好是一致的。再往外是L1 cache、L2 cache和L3 cache，每个cycle都可以读64字节、写64字节；然后是LLC（Last Level Cache）和EDRAM（embedded DRAM）。EDRAM是可选项，仅在某些高端型号才有，比如Intel Iris Pro 6200中就配置了128M的EDRAM，在计算架构下，EDRAM可以被用作cache缓存。再来看GPU和System DRAM之间的速度，支持双通道，每通道每个cycle只有8字节的读写速度，所以，对IO密集型的任务，要仔细调整算法来利用好cache，如果能让SIMD-n中所有lane访问的数据在同一个cache line上，这对提高性能具有重要的作用。



1. Intel GPU存储层次结构图2

除了要用好Cache外，OpenCL内核中对global memory的数据类型的选择也非常关键。一般来说，几乎所有硬件为了更快的访问内存，都会对要访问的内存首地址有对齐的建议。Intel GPU也不例外，其对系统内存的访问根据首地址是否DWORD对齐分成两种方法，不对齐情况的访问速度非常差；对齐情况下所使用的读写方法速度较快，每次最多可以读写四个DWORD。对OpenCL编译器来说，就是根据“\_\_global gentype \*p”中的数据类型gentype来判断是否DWORD对齐，比如，char、 char2、short等都无法保证是DWORD对齐，所以，就按照不对齐的假设来读写，这个时候的读写效率很受影响。碰到首地址不对齐而又读取连续一段数据的情况，如果数据足够多，那么推荐使用OpenCL的vload16/vload8等函数，可以一次性读取多个数据，编译器内部可能会做优化。如果gentype是uchar4、int、float2和short8等类型，那么隐含保证了p是DWORD对齐的，此时，尽量考虑使用uchar16/ushort8/uint4等数据类型以每次都可以读满四个DWORD。

### 3.2.2 AOS和SOA，两种不同的SIMD使用思路

在3.1中，我们介绍了并行计算以及SIMD。SIMD有两种使用思路，分别为AOS和SOA。AOS即Array of Struct，首先是个数组，然后每个数组元素（array element）是个struct；而SOA则是Struct of Array，首先是个struct，然后在struct中的每个数据成员是一个数组。我们将首先用C语言来介绍两者的具体区别，然后，将其应用到硬件寄存器级别，并从中衍生出两种不同的并行计算思路，最后，更加深入的分析基于SOA的OpenCL实现有什么特点。

假设要表示某个高维空间中的一些点（Point）的信息，每个点的属性信息由a、b、c和d这4个变量来表示。在如代码清单3-2所示的AOS内存布局中，某一个点的全部属性被放在一起，然后是下一个点的全部属性；

1. C语言的AOS形式

// AOS方式

struct Point {

float a;

float b;

float c;

float d;

};

Point points[512];

// 访问数据的典型应用场景是获取某个点的全部属性值。

for (int i = 0; i < 512; ++i) {

printf(“%f, %f, %f, %f\n”, points[i].a, points[i].b, points[i].c,

points[i].d);

}

在如代码清单3-3所示的SOA的内存布局中，所有点的某个属性被放在一起，然后是所有点的下一个属性值。

1. C语言的SOA形式

// SOA方式

struct Points {

float a[512];

float b[512];

float c[512];

float d[512];

};

Points points;

// 访问数据的典型应用场景是获取全部点的单一属性值。

for (int i = 0; i < 512; ++i) {

printf(“%f\n”, points.a[i]);

}

两者内存布局，并不存在哪个更优哪个更差，只是使用场景不同。如果一旦要访问一个点的某个属性，就会同时访问该点的全部属性，那应该采用AOS内存布局，可能读取一次内存到cache line中就足够了。同样道理，如果一旦要访问一个点的某个属性，就会访问所有点的该属性，那么应该采用SOA内存布局。

从C语言的变量的内存布局，到GPU硬件中的寄存器的数据布局，其实并没有什么不同，只要将GPU中的128个寄存器，看作一维的连续存储单元的组合就可以了，就好像内存的组成一样。为描述简单，我们假设有4个点（记为p0、p1、p2和p3）的信息要放在四个寄存器中（g0、g1、g2和g3），而且每个寄存器可以容纳4个浮点数，那么，寄存器的数据布局如图3-9所示，其中左侧是AOS布局，每个点的所有属性依次填入寄存器中，右侧则是SOA布局，先把所有点的a属性填入寄存器g0中，然后把所有点的b属性填入寄存器g1中，以此类推。



1. 寄存器数据布局

从C代码到硬件寄存器，虽然数据布局的概念没有什么变化，但是，由于应用场合的不同，如何利用这样的数据布局，并行计算和上述的C语言代码还是有区别的。对于AOS结构的并行计算思路，逻辑非常直截了当，我们在内核中写的伪代码如代码清单3-4所示。

1. AOS下内核伪代码和生成的汇编指令

//内核伪代码

vector<float, 4> p0, p1, p2;

...

p2 = p0 + p1;

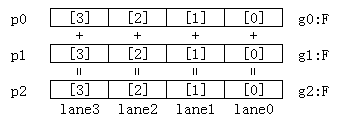
...

--------------------------------------------

// 编译器生成的SIMD-4指令

add(4) g2:F, g0:F, g1:F

然后生成SIMD-4的指令，向量p0的四个元素被填入寄存器g0，向量p1的四个元素被填入寄存器g1，向量p2的四个元素对应着寄存器g2，指令执行效果如图3-10所示。



1. AOS结构下的并行计算。代码清单3-4的指令执行结果（假设每个寄存器可以容纳4个浮点数），图中的 [i] 表示相应向量的第i个元素

我们可以看出来，这个思路最关键的是在写代码的时候，要尽量多利用向量vector（或者矩阵matrix等类似的概念），然后编译器也会尽量的将数据组合成vector形式，再将vector形式的代码翻译为SIMD指令，这样，一条代码就意味着一个指令多个数据，从而从SIMD指令中获益。这种思路的例子有CPU上的SSE/AVX指令，有Intel的CMRT[[3]](#footnote-4)（C for Media RunTime，将Intel GPU计算架构直接暴露给用户使用的库）等，难点在于对写内核代码的程序员要求比较高，需要用好vector，对编译器的要求也比较高，需要将代码中的标量数据尽量的组合为vector。假如代码写的不好，或者编译器优化不佳，最后得到的硬件指令只是一些标量运算，那就极大的浪费了SIMD中的并行运算性能了。基于SOA结构的OpenCL，可以较完美的解决这些问题。

我们知道，OpenCL内核代码是基于单个work item而写，而且里面可以任意的使用标量计算，在写内核的时候，无须有向量化的思维。这是因为，每个work item被映射到SIMD-n的一个lane上，每个lane上执行的本质上就是一个标量运算。图3-11展示了n是4的情况（即包括lane0、lane1、lane2和lane3），图中，wi是work item的缩写，为了表述方便，不管实际上是几维的work item（是指由clEnqueueNDRangeKernel函数参数work\_dim决定的维数），都从逻辑上将所有work item按一维从0开始重新排序，为每个work item赋予一个唯一的标识符，即图中的wi\_x表示一个标识序号是x的work item。根据work item的总量和work group的大小（图中假设work group的大小是包括m个work item），GPU会创建出足够数量的SIMD-4硬件线程，然后每个lane执行一个work item，最终完成所有的任务。



1. work item到SIMD-4的映射关系

由于每个SIMD模块对应着Intel GPU中的EU中的一个硬件线程，所以在Gen9中，如果采用宽度为n的SIMD指令，每个EU可以同时容纳的work item的最大数目是 7n个，因为每个EU中最多有7个硬件线程。如果work item之间都是独立的，那么理论上OpenCL可以支持任何数量的work item，只要将线程持续的往EU中送即可，EU做完一个线程就可以做下一个新的线程。在Gen9中，每个Subslice中有8个EU，所以每个Subslice可同时容纳的work item的最大数量是 56n个，如果work group中存在barrier或者local memory的话，如前所述，work group必须要被限定在单个Subslice范围内，那么一个work group中可以包含的work item的最大数量就是 56n个。

回过头来，OpenCL是如何应用SOA的呢？从前面C语言的例子出发，需要有个概念转换，即，需要将前面讨论的高维空间中的点换成work item。所以，在图3-9中，p0要转换成work item0，p1要转换成work item1，p2要转换成work item2，p3要转换成work item3。假如，OpenCL内核代码中有四个浮点变量a、b、c和d，那么，四个work item中的a变量被放入寄存器g0，b变量被放入寄存器g1，c变量被放入寄存器g2。相应的OpenCL内核伪代码和编译出来的指令如代码清单3-5所示。

1. SOA下内核伪代码和生成的汇编指令

// 内核伪代码

float a, b, c;

...

c = a + b;

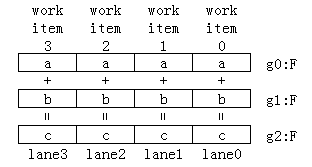
...

------------------------------------------

// 编译器生成的SIMD-4指令

add(4) g2:F, g0:F, g1:F

指令的执行结果见图3-12所示，四个work item中的变量a，和另一个变量b相加，结果存到变量c中，这样一条SIMD-4的add指令，就完成了4个work item的加法操作。



1. SOA结构下的并行计算。代码清单3-5的执行结果（假设每个寄存器可以容纳4个浮点数）

指令和指令执行结果，从硬件角度来看，和AOS的并无不同。区别在于开发者写的代码，也因此造成了指令执行结果所代表的物理意义的不同，这里的物理意义，就是4个work item通过SIMD-4指令，并行的完成了任务。

不管是基于AOS还是SOA衍生的并行计算方法，它们在将任务分解为多个线程进行并行处理的思路是一样的，区别在于如何用好SIMD-n的并行指令。在AOS中，需要开发者明确写出可以被并行的代码（部分借助编译器的优化），才能用好SIMD-n指令；而在SOA中，无须开发者介入，编译器只需按照直截了当的逻辑来组合work item，就可以生成SIMD-n指令。

OpenCL中，一个work item被映射到SIMD的一个lane上，多个work item被组合到一条SIMD-n指令中，形成一个SIMD硬件线程，而此线程只有一个指令指针（保存在ARF中的 instruction pointer）指出当前指令运行到了哪里，这意味着所有的lane都共用一个指令指针，所以任何一个lane要跑的指令，其他lane都必须也要一起执行。一旦碰到条件分支，内核伪代码如代码清单3-6所示。

1. 条件分支的内核代码

if (cond)

do\_something\_a();

else

do\_something\_b();

在一个SIMD线程中，如果有些work item的cond变量的值是True，而其他work item的cond变量的值是False，也就意味着，有些lane只需执行do\_something\_a()，有些lane只需执行do\_something\_b()。但是在实际执行时，如图3-13所示。



1. 条件分支的执行

所有的lane都会进入do\_something\_a()，只是逻辑上cond的值为0的lane并不会执行；而且，所有的lane也都会进入do\_something\_b()，cond的值为1的lane也不会真正执行。碰到这种情况，因为SIMD-n中存在无须执行指令的空闲lane，所以，未能发挥SIMD并行的最佳性能。假如一个SIMD线程中所有work item的cond变量的值都是1的话，那么，就没有任何一个lane需要去做do\_something\_b()，这样，线程将只做do\_something\_a()，而不会进入do\_something\_b()，从而可用提高并行性能；假如cond变量的值都是0的话，也是一样的，线程将只做do\_something\_b()，而不会进入do\_something\_a()。所以，在算法设计的时候，就应想好SIMD宽度，分析好每个SIMD硬件线程中所有work item的情况，使得每个线程中所有work item的cond变量的值或者全为1，或者全为0，从而用满SIMD中所有的lane；或者，在算法设计的最初始就明确要尽量少用条件分支、嵌套循环等。

OpenCL的一个work item被映射到一个lane上，单个lane其实就是个标量运算器，所以，基于单个work item所写的内核，其代码也应该以标量运算为主。假如代码中存在向量情况，编译器怎么处理呢？一般会进行标量化处理，比如short2 vload2(size\_t offset, const \_\_global short \*p)会拆成两个vload函数，如代码清单3-7所示。

1. vload2函数调用和编译器标量化代码

//内核伪代码

…

// const \_\_global short \*p

short2 s2 = vload2(offset, p);

...

---------------------------------------------

// 经过编译器标量化后的伪代码

...

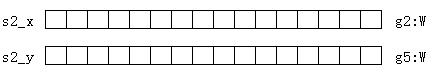
short s2\_x = vload(offset\*2, p);

short s2\_y = vload(offset\*2 + 1, p);

short2 s2 = (short2)(s2\_x, s2\_y);

...

其中变量的寄存器分配情况如图3-14所示（g2，g5只是例子，也可能分配到其他寄存器中），假如SIMD宽度为16，每个寄存器是256比特，那么16个work item中的s2\_x变量被分配到g2寄存器，而且16个work item中的s2\_y变量被分配到g5寄存器。 接下来基于g2或g5的全寄存器操作，就是针对这16个work item的并行操作。



1. vload2寄存器分配情况(解释代码清单3-7)

碰到特殊情况，比如可以直接用硬件特性进行优化解决，那就不进行标量化拆分处理，比如int4 vload4(size\_t offset, const \_\_global int \*p)。所以在写内核的时候，应该尽量的利用built-in 函数，而不要自己手写实现，因为built-in函数有更大的优化可能。

讲到这里，我们可以想到SOA也不是完美的，因为，假如一个work item内部需要对两个向量求內积，SOA就难以高效解决，这也是目前AOS和SOA两种思路并存的原因。

OpenCL内核代码不可避免的需要从DRAM中读写数据（比如读取global memory和constant memory），我们知道，从系统内存读取数据的速度，和EU SIMD指令的执行速度相比，非常的慢，我们应该尽量的利用cache来提高效率，在Intel GPU中，一条cache line的大小是64字节，所以，OpenCL内核读取global/constant memory，有如下的优化建议。

//假定数据是cache line对齐的global/constant memory，在SIMD16模式下

//最佳， 只读一行cache line， 没有带宽浪费

uint x = data[get\_global\_id(0)];

//不好， 跨2行cache lines， 浪费了一半的带宽

uint x = data[get\_global\_id(0) + 1];

//最差， 跨16行cache lines， 浪费了15/16 带宽

uint x = data[get\_global\_id(0) \* 16];

最后，简单提一下OpenCL中private memory的情况，private memory属于work item的私有存储，但是，EU中SIMD的lane并没有配备存储单元，所以，对于有限个数的private变量、不存在动态索引的private 数组，编译器会进行优化，将其编译为寄存器；但是，如果private变量太多，或者数组存在动态索引等情况，private memory只能用系统内存来提供，此时会非常影响性能。

### 3.2.3 cl\_intel\_subgroups在Intel GPU上的参考实现

在前面的两节中，我们了解了GPU的结构，以及不同的并行计算思想，本节介绍OpenCL扩展cl\_intel\_subgroups[[4]](#footnote-5)在Intel GPU上的参考实现。

在机器学习计算任务的优化中，OpenCL的半精度浮点数（half float）和subgroups（是一个OpenCL extension，名称为cl\_intel\_subgroups）被频繁用到，half float的实现逻辑比较简单，因为指令本身对各种数据类型是通用的，所有half float的实现主要在于分配好寄存器，然后在指令操作数中使用以HF为数据类型的寄存器，在我们之前的例子已有涉及。subgroups则和GPU硬件密切相关，我们将在本节详细讨论。

OpenCL的work item可以被分划为若干work group，而每个work group中的work item可以被进一步的分化为subgroup，实际上，subgroup就对应着一个SIMD硬件线程，被映射到同一个SIMD线程的work item就组成了一个subgroup。接下来介绍subgroups中定义的built in函数在Intel GPU上的参考实现[[5]](#footnote-6)，参考实现中的方法只是一种实现手段，不一定是最优的。

#### 1. 查询函数

查询subgroup中work item个数的函数原型如下所示：

uint get\_sub\_group\_size( void )

uint get\_max\_sub\_group\_size( void )

因为一个subgroup对应一个SIMD线程，所以，如果编译器选用SIMD指令的宽度是n，那么，get\_max\_sub\_group\_size的返回值也是n。考虑到work group中work item的数量可能不是n的整数倍，那么，几乎所有SIMD线程中的work item数量都是n，但是，最后一个SIMD线程的work item数量会小于n，get\_sub\_group\_size就返回每个线程上映射的work item的真实数量。

所有的subgroup从0开始编号，返回当前subgroup的编号的函数原型如下：

uint get\_sub\_group\_id( void )

在每个subgroup中，一个work item被映射到SIMD的一个lane上（lane编号从0到n-1），下面函数返回当前work item对应的lane的编号。

uint get\_sub\_group\_local\_id( void )

#### 2. 同步函数

在subgroup中进行同步的函数原型如下。

void sub\_group\_barrier( cl\_mem\_fence\_flags flags )

在Intel GPU实现中，因为subgroup对应的SIMD线程中每个lane都同步执行指令，所以本函数实现无需做任何事情。

#### 3. 条件组合函数

针对subgroup级别的条件组合函数的原型如下。

int sub\_group\_all( int predicate )

int sub\_group\_any( int predicate )

如果subgroup中所有work item的predicate变量都是True，则sub\_group\_all返回1，否则返回0。如果subgroup中存在任何一个work item的predicate变量是True，则sub\_group\_any返回1，否则返回0。以sub\_group\_all为例，伪代码、生成的硬件指令和分析见代码清单3-8所示。

1. sub\_group\_all函数调用和生成的汇编指令

// 内核伪代码

...

// int predicate;

int result = sub\_group\_all(predicate);

---------------------------------------------

// 生成的硬件指令（简化描述，有所调整）

// 假设：使用SIMD-16指令，而且predicate变量被分配为寄存器g16和g17中。

// 将寄存器g18和g19中的16个Unsigned DWORD（UD）值，置为1 (ox1UD)。

// 其中<1>表示从g18最低位开始，一个挨一个的依次写入UDORD值，因为SIMD的宽度为16，

// 所以，16个UDWORD就依次地也写到了g19中。（每个寄存器256比特，即8个DWORD值）

mov(16) g18<1>:UD 0x1UD

// 16个work item中的predicate变量被分配在寄存器g16和g17中，

// 其中 <8,8,1> 表示从g16最低位开始，数出16个DWORD值，限于篇幅，具体不展开解释。

// 这条命令，将g16开始的16个DWORD值，和0做不相等(ne)比较，结果写入flag寄存器f0.1中。

cmp.ne.f0.1(16) null:F g16<8,8,1>:D 0x0UW

// 首先将flag寄存器f0.1中的16个值做all16h，如果全1，则返回1，否则返回0。

// 如果返回1，则将g18/g19中的16个值（全是1）写入g20/g21中，

// 如果返回0，则将0写入g20/g21中。

// g20/g21中的16个DWORD值，即对应着16个work item中的变量result的值。

(+f0.1.all16h) sel(16) g20<1>:D g18<8,8,1>:UD 0x0UW

// 如果是sub\_group\_any的话，简单的将上述最后一条指令变为如下指令即可。

// (+f0.1.any16h) sel(16) g20<1>:D g18<8,8,1>:UD 0x0UW

#### 4. 广播函数

广播函数是将某一个work item中的变量的值，广播给subgroup中所有的work item知晓的函数，其函数原型如下所示。即，将由函数参数sub\_group\_local\_id决定的work item中的变量x的值，作为函数返回值赋给所有的work item。根据sub\_group\_local\_id是个编译期常量还是变量，分成两种实现方法。

gentype sub\_group\_broadcast( gentype x, uint sub\_group\_local\_id )

如果在编译时候发现sub\_group\_local\_id是个确定的值，比如是7，内核伪代码如下所示。

//内核伪代码

...

int y = sub\_group\_broadcast(x, 7);

还是以SIMD-16为例，假设16个work item的变量x的值被保存在寄存器g14/g15中，函数返回值y被保存在g16/g17中，那么，上述代码的最终执行效果如图3-15所示，SIMD-16中的第7个lane（lane7）的x值（即对应着寄存器g14.7），被赋给了所有lane中的y值（对应着寄存器g16/g17）。



1. lane7的x值，赋给了所有lane中的y值

对应生成的硬件指令非常简单，就一条mov指令，如下所示。

// 其中，g14.7<0,1,0>表示持续读取g14.7处的一个DWORD值作为源操作数。

mov(16) g16<1>:D g14.7<0,1,0>:D

如果sub\_group\_local\_id不是编译期常量，内核代码如代码清单3-9所示，变量from\_id是内核函数的一个参数，虽然其值在各个work item中都是相同的，但是，要等到运行时候才能确定。

1. sub\_group\_broadcast(变量)函数调用例子

//内核代码

kernel void compiler\_subgroup\_broadcast\_int(global int \*src,

global int \*dst,

uint from\_id)

{

uint index = get\_global\_id(0);

int val = src[index];

int bval = sub\_group\_broadcast(val, from\_id);

dst[index] = bval;

}

还是以SIMD-16为例，假如变量val最终被保存在寄存器g16/g17中，而且from\_id的运行时的值是9的话，那么可以构成一个数548（通过16<<5 + 9<<2计算得到）被放入a0寄存器中，a0寄存器是EU中用于寻址的一个特殊寄存器，其格式就是基址左移5位再加上偏移量左移2位，所以，a0中的548将被解释为从g16:D开始的第9个元素，即g17.1:D，然后，g17.1:D将被广播到寄存器g20/g21（即变量bval）中，如图3-16所示。



1. 借用寄存器a0进行广播

如果对生成的具体硬件指令感兴趣的话，可以继续阅读代码清单3-10，也可以直接略过。

1. sub\_group\_broadcast(变量)函数调用生成的硬件指令

// 生成的部分硬件指令，为简化描述，略有调整

// from\_id被保存在g1.6<0,1,0>中，这是一个UDWORD值，因为其在subgroup的所有work item中的值是相同的。

// 左移2位，是为了符合后续a0的格式要求，a0寄存器由16个UW值组成。

// 结果是16个相同的值(UD类型)，放在寄存器g18/g19中。

shl(16) g18<1>:UD g1.6<0,1,0>:UD 0x2UD

// 这里省略了用send指令将数据从global memory读入val中，具体在寄存器g16/g17中，

// 根据a0的格式要求，要将g16/17的寄存器编号填入a0中，所以这里要加 0x200 （即16<<5），

// 之前知道，g18/g19中保存和from\_id相关的16个UD值，取其低字节部分，形成16个UW值，

// 最终add指令，将结果填入a0寄存器的16个UW中。

add(16) a0<1>:UW g18<16,8,2>:UW 0x200UW

// 假如from\_id运行时的值是9，那么a0中的16个数值都是 9 << 2 + 0x200 = 548，

// 将128个寄存器从r0开始依次按字节从0开始排序，共有128\*32个字节，

// 第548个字节就对应着g17.1:D，

// 所以，这条指令将g17.1:D重复16次，放入寄存器g20/g21中，对应着bval变量。

mov(16) g20<1>:D g[a0]<VxH,1,0>:D

#### 5. 洗牌函数

shuffle有洗牌的意思，我们可以将一张牌看作一个数据，当前所有牌的叠放次序，决定了每一张牌是属于哪个work item的数据，经过洗牌后，牌还是那些牌，但是叠放次序发生了变化，也就是说，牌和work item的对应关系发生了变化，这张牌被洗过后很可能就属于另外一个work item了。subgroup中的洗牌函数则是将牌限制在了一个subgroup中，函数原型如下所示。

gentype intel\_sub\_group\_shuffle(gentype data, uint c)

gentype intel\_sub\_group\_shuffle\_down(gentype current, gentype next, uint delta)

gentype intel\_sub\_group\_shuffle\_up(gentype previous, gentype current, uint delta)

gentype intel\_sub\_group\_shuffle\_xor(gentype data, uint value)

之前的sub\_group\_broadcast广播函数是将subgroup中某一个work item中的变量x，广播给了该subgroup中所有的work item；而intel\_sub\_group\_shuffle不仅包括了这样的广播功能，还允许不同work item之间相互赋值，实现了将数据在不同的work item中进行交换，从而可以减少IO读取次数。在实现上，和sub\_group\_broadcast非常类似，只是寄存器a0中的值不再是16个相同的值（继续以SIMD-16为例），而是16个各异的值，从而达到shuffle的目的。假如函数的输入变量val对应着寄存器g16/g17，函数返回值存放的变量bval对应寄存器g20/g21，如图3-17所示，借助特殊寄存器a0，实现g16/g17到g20/g21的shuffle功能。比如，a0.6中的数值是548，而548表示对应源操作数是g17.1:D，这个源操作数将被写入目标寄存器g20.6:D中，即，a0.**x**所对应的源操作数，将被放入目标寄存器的第**x**个分量中。加了down/up/xor的后缀后，从实现上来说，也是类似的，只是填入a0寄存器中的数值需要根据函数后缀增加一些计算指令和条件判断指令来进行相应的调整。



1. shuffle洗牌函数执行效果图

#### 6. reduce函数

一般来说，map-reduce中的map是指将任务分解执行，而reduce则是将分解的结果进行整合，形成一个最终的单一结果。这里的reduce函数也是类似的功能，原型如下所示，将一个subgroup中的各个work item中变量x的值，做add/min/max运算进行整合，得到的结果（一个值），作为所有work item中的函数返回值。

gentype sub\_group\_reduce\_add( gentype x )

gentype sub\_group\_reduce\_min( gentype x )

gentype sub\_group\_reduce\_max( gentype x )

接下来以float y = sub\_group\_reduce\_add(float x)为例说明实现方法。使用SIMD-16指令宽度生成的硬件指令如代码清单3-11所示，这只是一种参考实现，其中的硬件指令大量的使用了标量运算，虽然性能不高，还存在很多优化空间，但是逻辑简单清晰，作为参考讲解是最合适的选择。

1. sub\_group\_reduce\_add函数调用生成的硬件指令

//生成的指令如下所示（简化描述），这只是一种参考实现，还可以优化实现

//变量x被保存在寄存器g14/g15中，可以依次用g14<0,1,0>:F，g14.1<0,1,0>:F，……，//g15.6<0,1,0>:F，g15.7<0,1,0>:F来得到所有work item的x变量值。

// 将这些变量依次相加，每次只是一个标量处理，所以，指令的（）里面的数字是1，

// 当SIMD宽度为16时，我们可以使用宽度小于16的SIMD指令，当然这有些浪费并行能力了。

// 具体每条指令的解释，可参看相应的图示化过程。

mov(1) g20<1>:F g14<0,1,0>:F

add(1) g20<1>:F g20<0,1,0>:F g14.1<0,1,0>:F

add(1) g20<1>:F g20<0,1,0>:F g14.2<0,1,0>:F

add(1) g20<1>:F g20<0,1,0>:F g14.3<0,1,0>:F

add(1) g20<1>:F g20<0,1,0>:F g14.4<0,1,0>:F

add(1) g20<1>:F g20<0,1,0>:F g14.5<0,1,0>:F

add(1) g20<1>:F g20<0,1,0>:F g14.6<0,1,0>:F

add(1) g20<1>:F g20<0,1,0>:F g14.7<0,1,0>:F

add(1) g20<1>:F g20<0,1,0>:F g15<0,1,0>:F

add(1) g20<1>:F g20<0,1,0>:F g15.1<0,1,0>:F

add(1) g20<1>:F g20<0,1,0>:F g15.2<0,1,0>:F

add(1) g20<1>:F g20<0,1,0>:F g15.3<0,1,0>:F

add(1) g20<1>:F g20<0,1,0>:F g15.4<0,1,0>:F

add(1) g20<1>:F g20<0,1,0>:F g15.5<0,1,0>:F

add(1) g20<1>:F g20<0,1,0>:F g15.6<0,1,0>:F

add(1) g20<1>:F g20<0,1,0>:F g15.7<0,1,0>:F

// 将计算结果存入寄存器g16/g17中，作为subgroup中16个work item的y变量值。

mov(16) g16<1>:F g20<0,1,0>:F

其中，函数的输入变量x被保存在寄存器g14/g15中，一共16个浮点数，对应着subgroup中16个work item的变量x，不妨记为a、b、c、d、…、l、m、n、o和p，如图3-18所示，第一条指令将a移入寄存器g20的最低位分量中；第二条指令是加法指令，将g20中的最低分量（已经有值a）和b相加，并且结果继续放入g20中的同一位置；如此一直执行到倒数第二条指令，将a+b+c+…+p的结果存入寄存器g20的最低位分量中；最后一条指令则是将最终的累加结果移入寄存器g16/g17中，对应着16个work item的变量y的值。由此也说明了16个work item中的函数返回值都是相同的。



1. sub\_group\_reduce\_add函数执行的图示化过程

#### 7. scan函数

之前介绍的reduce函数，考虑了subgroup中所有work item的情况，最后返回一个值。还存在其他情况，比如，要统计2岁以内所有儿童的最高身高、统计3岁以内的所有儿童的最高身高、直到统计17岁以内所有儿童的最高身高，在16个任务有所重叠又有所区别，将这16个任务分配到一个subgroup中的16个work item中完成的话，就需要scan系列函数来完成，函数原型如下所示。作个粗略通俗的类比解释，如果统计x岁以内所有儿童的最高身高的时候，包括了x岁的儿童，那么函数名应该含有inclusive字样，如果不包括x岁的儿童，则应使用含有exclusive字样的函数名。

gentype sub\_group\_scan\_exclusive\_add( gentype x )

gentype sub\_group\_scan\_exclusive\_min( gentype x )

gentype sub\_group\_scan\_exclusive\_max( gentype x )

gentype sub\_group\_scan\_inclusive\_add( gentype x )

gentype sub\_group\_scan\_inclusive\_min( gentype x )

gentype sub\_group\_scan\_inclusive\_max( gentype x )

这批函数在subgroup中对有选择性的work item做相应的add/min/max操作，每个work item得到不同的函数返回值。接下来以float y = sub\_group\_scan\_inclusive\_add(float x)为例详细介绍函数的功能和实现手段。使用SIMD-16指令宽度生成的硬件指令如代码清单3-12所示，这也只是一种参考实现。

1. sub\_group\_scan\_inclusive\_add函数调用生成的汇编指令

// 生成的硬件指令，简化描述

// 变量x被保存在寄存器g14/g15中，最后结果在寄存器 g20/g21中。

// 相关指令前面都已经有过介绍，不再赘述。

mov(1) g20<1>:F g14<0,1,0>:F

add(1) g20.1<1>:F g20<0,1,0>:F g14.1<0,1,0>:F

add(1) g20.2<1>:F g20.1<0,1,0>:F g14.2<0,1,0>:F

add(1) g20.3<1>:F g20.2<0,1,0>:F g14.3<0,1,0>:F

add(1) g20.4<1>:F g20.3<0,1,0>:F g14.4<0,1,0>:F

add(1) g20.5<1>:F g20.4<0,1,0>:F g14.5<0,1,0>:F

add(1) g20.6<1>:F g20.5<0,1,0>:F g14.6<0,1,0>:F

add(1) g20.7<1>:F g20.6<0,1,0>:F g14.7<0,1,0>:F

add(1) g21<1>:F g20.7<0,1,0>:F g15<0,1,0>:F

add(1) g21.1<1>:F g21<0,1,0>:F g15.1<0,1,0>:F

add(1) g21.2<1>:F g21.1<0,1,0>:F g15.2<0,1,0>:F

add(1) g21.3<1>:F g21.2<0,1,0>:F g15.3<0,1,0>:F

add(1) g21.4<1>:F g21.3<0,1,0>:F g15.4<0,1,0>:F

add(1) g21.5<1>:F g21.4<0,1,0>:F g15.5<0,1,0>:F

add(1) g21.6<1>:F g21.5<0,1,0>:F g15.6<0,1,0>:F

add(1) g21.7<1>:F g21.6<0,1,0>:F g15.7<0,1,0>:F

其中，函数的输入变量x被保存在寄存器g14/g15中，一共16个浮点数，对应着subgroup中16个work item的变量x，不妨记为a、b、c、d、…、l、m、n、o和p，如图3-19所示，第一条指令将a移入寄存器g20的最低位分量中；第二条指令是加法指令，将g20中的最低分量（已经有值a）和b相加，并且结果继续放入g20中的次低位分量中；如此一直执行到最后一条指令，将a+b+c+…+p的结果存入寄存器g21的最高位分量中。寄存器g20/g21对应着16个work item的变量y的值，由此也说明了16个work item中的函数返回值各不相同。



1. sub\_group\_scan\_inclusive\_add函数执行的图示化过程

#### 8. 块读写函数

这批函数是为了Intel GPU硬件功能而专门提出，分为读写buffer和读写image两部分。读写buffer的函数原型如下所示。

uint intel\_sub\_group\_block\_read( const \_\_global uint\* p )

uint2 intel\_sub\_group\_block\_read2( const \_\_global uint\* p )

uint4 intel\_sub\_group\_block\_read4( const \_\_global uint\* p )

uint8 intel\_sub\_group\_block\_read8( const \_\_global uint\* p )

void intel\_sub\_group\_block\_write( \_\_global uint\* p, uint data )

void intel\_sub\_group\_block\_write2( \_\_global uint\* p, uint2 data )

void intel\_sub\_group\_block\_write4( \_\_global uint\* p, uint4 data )

void intel\_sub\_group\_block\_write8( \_\_global uint\* p, uint8 data )

在以往的OpenCL操作中，在读写global memory的时候，SIMD线程中每一个lane都有自己要读写的地址值，而在这批函数中，一个SIMD线程作为一个subgroup整体，只有一个要读写的地址p，所以被称为block块操作。

由于相应的硬件指令比较复杂，这里不做具体介绍，但是以

uint2 u2 = intel\_sub\_group\_block\_read2(const \_\_global uint\* p)

为例说明指令的执行结果是怎么样的。对于SIMD16指令，每个work item读取uint2，一共有16\*2\*4=128字节，即从p指向的内存中读取连续的128字节，如图3-20所示，假设uint2 u2变量被分配了寄存器g10/g11/g12/g13，从g10开始将4个寄存器以字节为单位按地址从低到高排列，可以形成一段连续地址空间。指令的执行结果，就是将p指向的128字节的数据，直接拷贝到这段地址空间上。根据SOA数据布局的特点，16个work item中的u2.x存放在寄存器g10/g11中，而u2.y则存放在寄存器g12/g13中。从中，我们还可以看出，同一个work item中的u2.x和u2.y在global memory中，差了64个字节，即16个uint，这也是在定义中提到的p[sub\_group\_local\_id + max\_sub\_group\_size]中max\_sub\_group\_size的由来。回顾之前介绍的vload2函数，假设返回值变量是s2，那么同一个work item中的s2.x和s2.y是来自global memory的连续地址空间，这是和本批函数最大的不同之处。



1. 以块的方式读取连续128字节到uint2变量中

以image为读写目标的函数原型如下所示。

uint intel\_sub\_group\_block\_read(image2d\_t image, int2 byte\_coord)

uint2 intel\_sub\_group\_block\_read2(image2d\_t image, int2 byte\_coord)

uint4 intel\_sub\_group\_block\_read4(image2d\_t image, int2 byte\_coord

uint8 intel\_sub\_group\_block\_read8(image2d\_t image, int2 byte\_coord)

void intel\_sub\_group\_block\_write(image2d\_t image, int2 byte\_coord, uint data)

void intel\_sub\_group\_block\_write2(image2d\_t image, int2 byte\_coord, uint2 data)

void intel\_sub\_group\_block\_write4(image2d\_t image, int2 byte\_coord, uint4 data)

void intel\_sub\_group\_block\_write8(image2d\_t image, int2 byte\_coord, uint8 data)

这批函数和刚介绍的函数相比，只是函数参数不同，从global gentype \*p换成了image2d\_t image，这也是为了Intel GPU硬件的专门功能而提出的函数扩展。一个SIMD线程作为一个subgroup整体，只有一个要读写的image2d和坐标值int2 byte\_coord，所以也被称为block块操作。由于涉及到的硬件指令比较复杂，而且存在可能的指令拆分和寄存器重排等情况，在此不做具体介绍，仅以

uint4 u4 = intel\_sub\_group\_block\_read4(image2d\_t image, int2 byte\_coord)

为例说明指令的执行结果是怎么样的。如图3-21所示，byte\_coord指出要了读取image的起始位置，宽度16来自SIMD的宽度，高度4来自uint4中的4，图中的第1行被读入寄存器g20/g21中，对应着16个work item的u4.x变量，第2行被读入g22/g23中，对应着16个work item的u4.y变量，以此类推。



1. 以块的方式读取image到uint4变量中

## 3.3 本章小结

本章首先介绍了并行计算的历史和分类，并且在指令级并行中重温了SIMD的概念，SIMD也是Intel GPU的基本计算单元。然后从SoC开始介绍Intel GPU计算架构，逐步深入到Slice、Subslice、执行单元和存储层次，中间穿插了基于Intel GPU的OpenCL的优化建议。接下来从AOS和SOA两个不同角度，介绍了SIMD指令的使用思路。最后，介绍了cl\_intel\_subgroups这个在深度学习中比较常见的OpenCL扩展在Intel GPU上的参考实现，希望通过这个介绍，可以加深对Intel GPU计算架构的理解，也可以更好的理解OpenCV深度学习模块中基于OpenCL加速的具体实现手段。

1. https://www.mersenne.org/ [↑](#footnote-ref-2)
2. 图片来源 https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf [↑](#footnote-ref-3)
3. 参见https://github.com/intel/cmrt [↑](#footnote-ref-4)
4. 参见https://www.khronos.org/registry/OpenCL/extensions/intel/cl\_intel\_subgroups.html。 [↑](#footnote-ref-5)
5. 参考实现详见 https://www.freedesktop.org/wiki/Software/Beignet/。 [↑](#footnote-ref-6)