Alveo 加速卡开发应用白皮书

2019 年 4 月 17 日

目录

第 1 章：采用 Alveo 启动设计，实现加速

介绍 5

提供的设计文件 5

第 2 章：6 Alveo 简介 ....
加速基础

加速概念 6

发现加速机会 8

赛灵思运行时 (XRT) 与 API 9

第 3 章：运行时软件设计

内存分配概念 11

Alveo 引导软件介绍 14

引导软件示例 15

提供的设计文件 15

硬件设计设置 15

构建软件设计 16

示例 0：加载 Alveo 镜像 17

- 简介 17
- 关键代码 17
- 运行应用 18
- 补充练习 19
- 要点总结 19

示例 1：简单内存分配 20

- 简介 20
- 关键代码 20
- 运行应用 24
- 补充练习 26
- 要点总结 26

示例 2：对齐内存分配 27
简介 27
关键代码 27
运行应用 27
补充练习 29
要点总结 29
示例 3：使用 OpenCL 分配内存 30
简介 30
关键代码 30
运行应用 32
补充练习 33
要点总结 34
示例 4：并行化数据路径 35
简介 35
关键代码 35
运行应用 37
补充练习 39
要点总结 40
示例 5：优化计算与传输 41
简介 41
关键代码 42
运行应用 44
补充练习 45
要点总结 46
示例 6：再遇波折 47
简介 47
关键代码 47
运行应用 47
补充练习 48
要点总结 49
示例 7：使用 OpenCV 缩放图像 50
简介 50
关键代码 51
运行应用 51
补充练习 53
要点总结 54

示例 8：使用 OpenCV 流水线化操作 55
简介 55
关键代码 55
运行应用 56
补充练习 57
要点总结 58

总结 59

附录 A：附加资源与法律提示

XILINX

第 1 章

采用 Alveo 启动设计，实现加速

介绍

赛灵思 FPGA 与 Versal ACAP 器件专门适用于高性能算法与工作负载的低时延加速。随着传统的摩尔定律走向式微，设计专用架构 (DSA) 正成为开发者用来实现功能、功耗、时延和灵活性的理想平衡的首选工具。然而，单纯从软件角度解决 FPGA 和 ACAP 的开发问题，似乎难度极大。

通过本套文档和教程，我们的目标是为您、我们的客户以及合作伙伴提供易于遵循的指导性介绍，以便运用赛灵思卡产品加速应用。我们先从基本加速原理出发：理解基本的架构方法，确定适用于加速的代码，与管理内存的软件 API 进行交互、并以最佳方式与 Alveo 卡进行交互。

本套文档主要提供给软件开发者使用，并不用作底层硬件开发者指南。RTL 编码、底层 FPGA 架构、高层次综合优化等主题请参见其他赛灵思文档的讲解。本白皮书的目的是帮助您快速掌握和使用 Alveo，信心满满地达成您的加速目标，并逐步提升您对赛灵思器件的熟悉程度，增强技能。

提供的设计文件

在本目录树下您将找到两个主目录：**doc** 和 **examples**。这两个目录下分别存放您目前阅读的文档源与文档随附的示例设计。示例设计与本指南中的特定章节对应。本文尽量让文中的代码例保持简洁明了，“点到即止”。

XILINX

第 2 章

加速基础

加速概念

先不要急于深入探究 API，特别是在您的加速经验不足时。我们先用一个简单的比喻帮助您理解加速系统的工作方式。

假设您得到了一份在您的城市当导游的工作。您的城市可能大可能小，人口分布可能密集可能稀疏，可能还必须要遵循特定的交通规则。这就是您的**应用空间**。一路上，您需要向游览者介绍您城市的风土人情，还必须去特定景点（和商店 - 毕竟您必须获得报酬）。这套您必须完成的事项就是您的**算法**。

在确定应用空间和算法后，您先从小处着手。为您的旅行购置一辆小轿车。每年您都会购买一辆更大更快的新车。年年如此。但随着您名气的增长，越来越多的人慕名前来请您做导游。您的车的行驶速度有限（即便是跑车），您该怎样提升？这就引入 **CPU 加速**问题。

答案是买辆旅游大巴。与您之前用的跑车相比，可能最大行驶速度较低，乘客上下车用时更长，但是您能搭载更多的游客（也就是更多的**数据**）通过您的算法。随着时间的推移，您的大巴车队不断壮大，您能导引越来越多的游客观光游览。这就是 **GPU 加速**模式。这种模式提供极为优秀的缩放能力，直到您的燃油费开始攀升，喷泉前发生交通拥堵。更麻烦的是，城里将要举办世界杯！

假设城里允许您建造单轨电车。由城里出资，而且许可证已经预先批准。电车不停发车，车车满载，停靠一路上的所有车站（还能灵活地根据需要开辟道路，改变路线）。对我们这个越来越走样的比喻中的导游来说，这纯属幻想。但对您来说，这就是 **Alveo 加速**模式。FPGA 将 GPU 的并行性与域专用架构的低时延流相结合，提供无与伦比的性能。

不过正如笑话所说，如果您不学会换档，法拉利也跑不快。那么让我们抛开比喻，实际地一探究竟。

发现加速机会

一般来说，采用加速系统的目的总是要达成特定的性能目标，无论目标是所谓的总体端到端时延、每秒帧数、原始吞吐量还是其他什么。

一般来说，适用于加速的解决方案是能以确定方式处理大量数据的算法处理块。

1976 年，吉恩 • 阿姆达尔 (Gene Amdahl) 提出了著名的阿姆达尔定律。该定律描述的是系统的加速潜力：

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

在这个方程中，*S latency* 是任务的理论加速潜力，*s 是受益于加速技术的算法部分的加速幅度，p 是加速前该任务占用的执行时间的比例。*

阿姆达尔定律表明，在特定应用中，加速带来的优势受到明显的限制。原因在于任务不能被加速的部分（一般涉及决策、I/O 或其他系统开销任务）往往将成为系统瓶颈。

$$\lim_{s \to \infty} S_{latency}(s) = \frac{1}{1-p}$$

但是如果阿姆达尔定律是正确的，那么为什么如此大量的现代系统使用通用加速或域专用加速？关键在于这里：现代系统处理的是不断增加的数据量，而阿姆达尔定律*只适用于工作量固定的情况*。也就是说，限值只发生在 *p 占整个执行时间的比例保持恒定的情况下。*

1988 年，约翰 • 古斯塔夫森 (John Gustafson) 和埃德温 • 巴西斯 (Edwin Barsis) 对方程进行调整，提出了著名的古斯塔夫森定律。该定律重新解释了这个问题：

$$S_{latency}(s) = 1 - p + sp$$

这里，*S latency* 仍代表任务的理论加速潜力，*s* 是受益于并行性的任务的时延改善幅度，*p* 是受益于并行性的任务部分的时延占总任务时延（施加并行性前）的比例。

古斯塔夫森定律从不同的角度阐述了阿姆达尔定律提出的问题。不是使用更多计算资源加速执行单个任务，而是使用更多计算资源在相同时间内完成更多计算。

两个定律从不同角度解释了同一个问题：要想加速应用，我们需要实现并行化。采取并行化后，我们要么能在相同时间内处理更多数据，要么能在更短时间内处理相同数据。两种方法都存在数学局限性，但也都受益于更多资源（虽然程度不同）。

一般来说，Alveo 卡适合加速在大型数据集上开展“数字运算”的算法。例如视频转码、财务分析、基因组学、机器学习和具有能并行处理的大数据块的其他应用。在开展加速时，必须准确理解向软件算法施加外部加速的方式和时间。随意让 Alveo 或其他任何加速器加速随机代码，并不总是能获得理想结果。原因有两个：首先也是最重要的是，要想充分发挥加速功能，有时需要重构顺序算法，提高其并行性。其次，虽然 Alveo 架构能够快速处理并行数据，但是

通过 PCIe 传输数据以及在 DDR 存储器间传输数据，会产生内在的*额外时延*。您可以将这种时延视为与任何外部加速器共享数据时必须负担的“加速代价”。

在了解这些前提条件后，接着寻找符合下列几项条件的代码段。这些代码段应：

- 以确定方式处理大型数据块。
- 有明确定义的数据相依关系，最好是基于顺序或流的处理。应避免随机访问，除非能设定界限。
- 有足够长的时间用来处理，CPU 和 Alveo 卡间的数据传输开销不在加速器运行时中占主导地位。

Alveo 简介

在深入探究软件之前，让我们首先熟悉 Alveo 卡本身的功能。每张 Alveo 卡都结合三大基本要素：用于加速的高性能 FPGA 或 ACAP、高带宽 DDR 存储器组、通过高带宽 PCIe Gen3x16 链路与主机服务器连接。该链路能在 Alveo 卡与主机间每秒传输大约 16GiB 的数据。

需要重申的是，与 CPU、GPU 或 ASIC 不同的是，FPGA 实际上是毫无预装控件。FPGA 提供大量底层逻辑资源，如触发器、门和 SRAM，但基本没有固定功能。包括 PCIe 和外部存储器链路在内的 FPGA 器件的每一个接口都是使用部分上述资源实现的。

为了确保主机处理器，PCIe 链路、系统监控和电路板健康接口随时可用，Alveo 的设计被划分为 **Shell** 和**角色**概念模型。**Shell** 内置所有静态功能：外部链路、配置、时钟等。通过使用定制逻辑执行您的特定算法，您的设计可实现在模型的**角色**部分。这一拓扑结构请参见图 2.1。

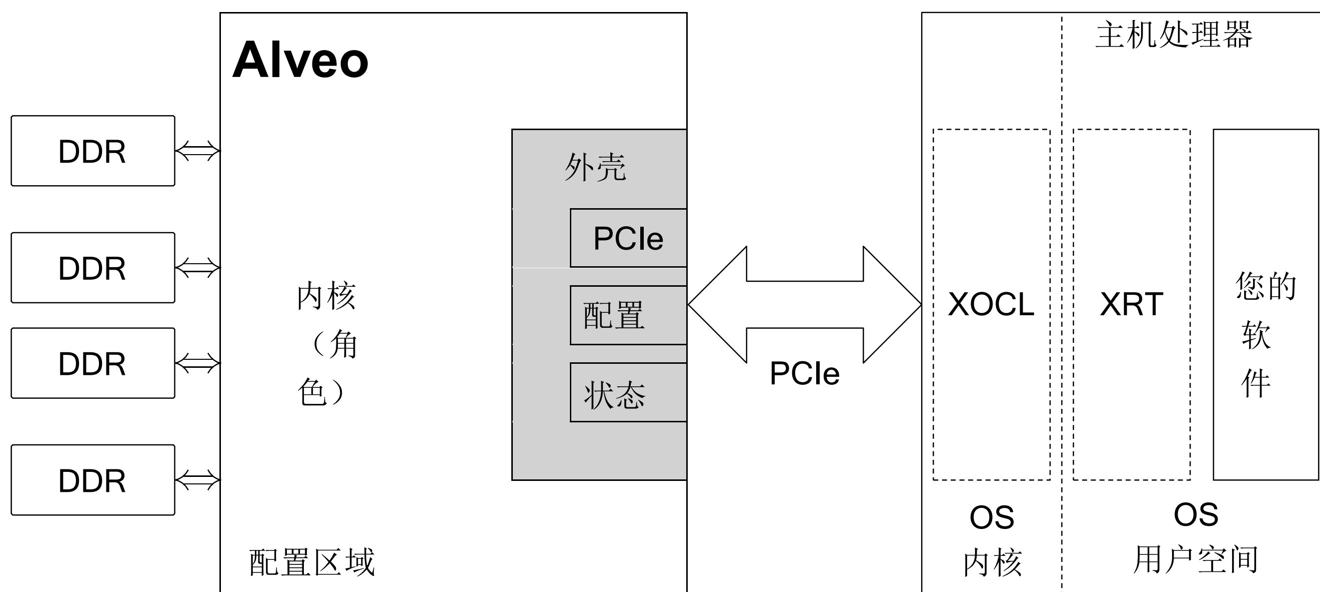

图 2.1：Alveo 概念拓扑结构

Alveo FPGA 进一步划分为多个超级逻辑区域 (SLR)，用于协助架构的

超高性能设计。不过这个略高级的主题在您刚开始涉足 Alveo 开发时，基本不会注意。

Alveo 卡提供多个卡载 DDR4 存储器。这些存储器与 Alveo 器件的往来带宽高。具体到 OpenCL，它们被集体称为*器件全局内存*。每个存储器组的容量为 16 GiB，运行频率为 2400MHz DDR。该存储器带宽极高，需要时可以轻易让您的内核饱和。然而，无论是读取还是写入内存都会发生时延，尤其是在访问的地址不连续或读写短猝发式数据时。

另一方面，PCIe 通道也有较大的带宽，但不如 Alveo 卡自带的 DDR 存储器带宽大。此外，通过 PCIe 传输数据产生的时延相当高。根据经验，应尽量避免通过 PCIe 传输数据。对于连续数据处理，应尝试将您的系统设计成在内核处理其他任务时传输您的数据。例如，在等待 Alveo 完成一帧视频的处理时，可以同时将下一帧数据从 CPU 传输到全局内存。

关于 FPGA 架构、Alveo 卡本身，还有许多知识可以深入了解。但就本文的介绍性目的而言，将不再赘言。从设计加速架构的角度，需要记住的要点有：

- 通过 PCIe 传输数据开销较高，即便使用 Gen3x16，时延也相当高。对较大量的数据传输，带宽容易成为系统瓶颈。
- DDR4 与 FPGA 之间的带宽和时延显著优于 PCIe，但是从整体系统性能而言，访问外部存储器仍然成本较高。
- 在 FPGA 架构内从一个运算流动到下一个运算基本不产生成本（请参见我们前文的电车类比。我们将在下文详细讲解）。

赛灵思运行时 (XRT) 与 API

虽然看似显而易见，任何硬件加速系统都可以从广义上分为两个部分进行讨论：硬件架构和硬件实现方案；与该硬件进行交互的软件。对于 Alveo 卡，无论您的应用里使用任何更高级的软件框架，如 FFmpeg、GStreamer 或其他，与 Alveo 硬件在底层交互的软件库都是赛灵思运行时 (XRT)。

虽然 XRT 由多个组件构成，但其主要用途可归结为简单的三个方面：

- **编程 Alveo 卡内核**并管理硬件生命周期
- **分配内存**并在主机 CPU 和加速卡之间迁移内存
- **管理硬件运行：**顺序执行内核，设置内核参数等

这三方面也是按照实现在 Alveo 加速卡上的开销水平，从高到低依次排序。下面我们深入了解它们。

编程 Alveo 卡内核本质上需要占用一定时间。根据加速卡上 FPGA 的容量大小、可用于传输配置镜像的 PCIe 带宽等，所需时间一般在

几十到几百毫秒。这一般只在您启动应用时“一次性发生”，因此配置开销可合并在总建立时延中，但必须予以注意。在某些应用中，Alveo 会在运行过程中被多次重编程，以提供不同的大型内核。如果您计划构建这样的架构，应将这种配置时间尽可能无缝地纳入您的应用中。还需要注意到是，虽然众多应用都能同时使用 Alveo 卡，但在任何时点只能编程一个镜像。

分配内存和迁移内存是 XRT 的真正“诀窍”。对于开发加速架构而言，有效地分配和管理内存是一项关键技能。如果您未能有效管理内存和内存迁移，将*严重*影响整体应用性能，而且是负面影响！幸运的是，XRT 提供众多用来与内存进行交互的功能，我们将在后文中详细讲解。

最后，XRT 通过设置内核参数和管理内核执行流程来**管理硬件运行**。内核能顺序或并行运行，从一个进程或多个进程运行，以及以阻断方式或非阻断方式运行。您的软件与您的内核的具体交互方式取决于您的控制。我们将在后文介绍部分示例。

需要注意到的是，XRT 是一种低级别 API。在非常高级或非常规的使用模式下，您可以选择与它直接进行交互，但大部分设计人员选择使用高级 API，如 OpenCL、赛灵思媒体加速器 (XMA) 框架等。图 2.2 所示的是可用 API 的顶层视图。在本文中，我们将主要讨论使用 OpenCL API，以便于读者理解。如果您曾经使用过 OpenCL，您将发现很多相似点（虽然赛灵思为 FPGA 特定任务提供了部分扩展）。

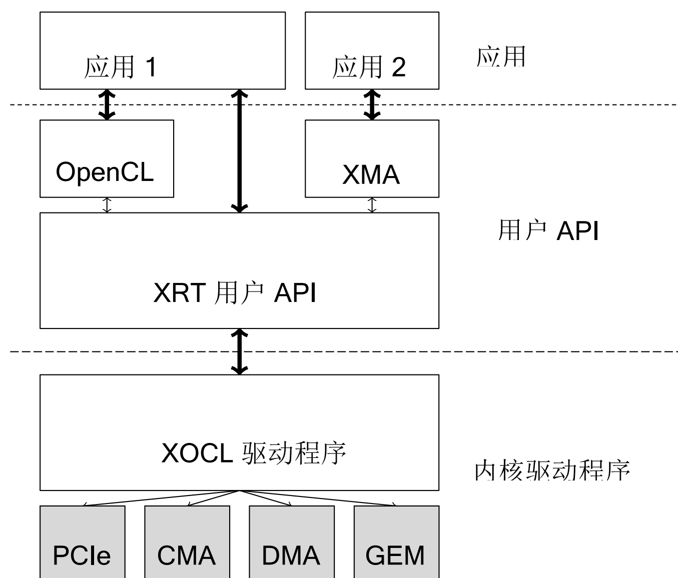

图 2.2：XRT 软件堆栈

第 3 章

运行时软件设计

内存分配概念

在 CPU 上运行程序时，您一般不用关心底层硬件如何管理内存。有时在某些处理器架构上存在对齐问题，但是大部分现代操作系统和编译器已经将此抽象到一定程度，除非要处理大量底层驱动程序工作（或加速），一般情况无需关心。

从根本上来说，可以将存储器视为在广义上拥有六大属性。以指向数据缓存的指针为例，数据指针可以是**虚拟**指针或**物理**指针。指针指向的内存可以是**页式**内存或**物理连续型**内存。最后从处理器的角度，内存可以是**可缓存**内存或**不可缓存**内存。

大部分现代操作系统都使用虚拟内存。使用虚拟内存的原因较多，但为避免该文档跑题，甚至变成冗长的计算机架构课本，您只需要知道最常运行 XRT 的 Linux 操作系统使用虚拟内存。因此，当您在使用标准的 C 或 C++ 用户空间 API 函数时，如 *malloc() 或 new，您最终是让指针指向虚拟内存地址而非物理存内存地址。*

此外，您也可能将指针指向**页式**内存中的地址块。基本上每种现代操作系统（包括 Linux）都将地址范围划分为页，一般每页大小 4 KiB（虽然可能会随操作系统不同而有所变化）。随后，每页将映射到物理内存中的对应页。如果有人认为这种表述是一种不准确的简化归纳，我在此礼貌地提醒您，这*不是*一堂计算架构课程。

但必须注意两个重要方面。首先，用标准 C API 从堆阵分配缓存时，返回的不是物理内存地址。其次，得到的并非单个缓存，而是 N 个内存*页*组成的集合，每页大小 4KiB。为便于理解，假设我们分配了 6 MiB 的缓存。那么可以得到：

$$\frac{6\ \text{MiB}}{4\ \text{KiB}} = 1536\ \text{页}$$

如果您想要将整个 6 MiB 缓存从主机复制到您的 Alveo 卡上，就需要将 1536 个虚拟页地址解析到物理内存地址。随后，需要将物理地址装配成*散集列表*，排队到有散集功能的 DMA 引擎，接着由 DMA 将这些页逐一复制到它们的目的地。如果想要将缓存从 Alveo 复制到主机内存中的虚拟分页地址范围，也可以反向操作。主机处理器一般速度较快，因此，创建列表不会花费大量时间。但鉴于缓存容量较大，这样做会

增加系统总体时延，所以必须注意这种做法对整体系统性能的影响。

图 3.1 所示的是这一系统的简化图。在本例中有两个缓存，分别命名为 A 和 B。A、B 均为虚拟缓存。在本例中，我们拟将缓存 A 传输给 Alveo，进行一定处理。接着将处理结果传输给缓存 B，然后再传回。从该图可以理解虚拟转物理映射的工作方式。在 Alveo 卡中，加速器只在物理内存地址上工作，数据一直连续存储；这主要是因为这种配置一般可提供最佳性能。

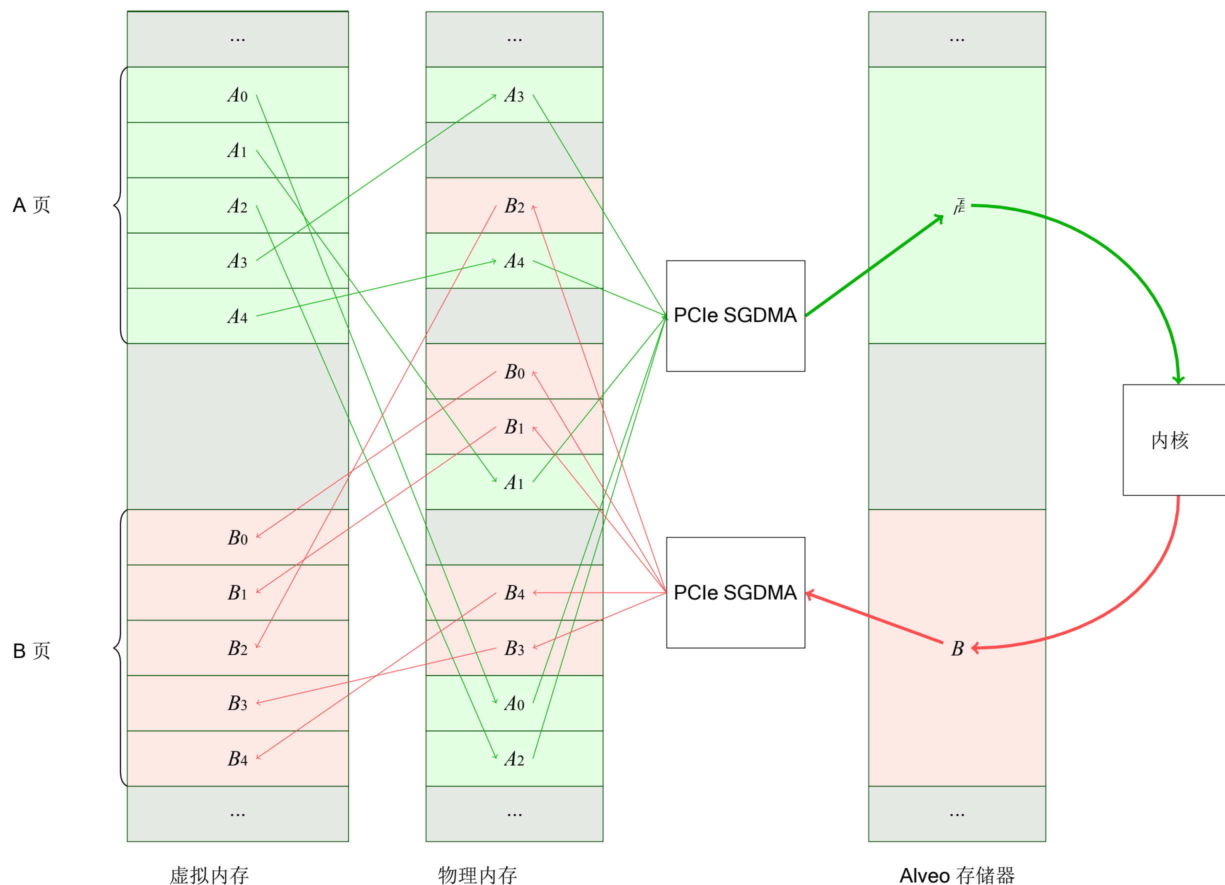

图 3.1：虚拟内存与 Alveo 之间的数据传输

正如您所看到的，我们的简化示例数据流也已经有些复杂。现在，假设这个数据流跨越一个很大的缓存：容量数兆字节（或数千兆字节），有成千上万个分页。可以注意到的是，就算是使用高速主机处理器，构建和管理这些散集列表以及管理页表也会相当地费时。实际上，存储器一般并不像我们示例中的那样碎片化。但是，因为一般事前不知道分页的物理地址，所以必须将它们中的每一个视为唯一地址。

然而，如果知道所有分页在物理内存中是**连续的**，那么构建散集列表就会简单很多。这就是说数据字节顺序排列，物理地址按[n+1]渐次增大。在这种情况下，可以在只知道缓存起始地址和它的容量的情况下构建散集列表。

这样做对许多 DMA 操作都有帮助，而不仅仅是使用 DMA 传输 Alveo 数据。现代操作系统提供专门用于该目的的内存分配器（一般通过内核服务）。在 Linux 中，这一操作通过**连续内存分配器**子系统来实现。这本属于内核空间功能，但通过各种机制提供给用户使用。这些机制包括 *dmabuf*、XRT API、各种图形驱动程序和其他机制。如果我们能连续分配之前的缓存，最终结果可得到大幅简化，如图 3.2 所示。

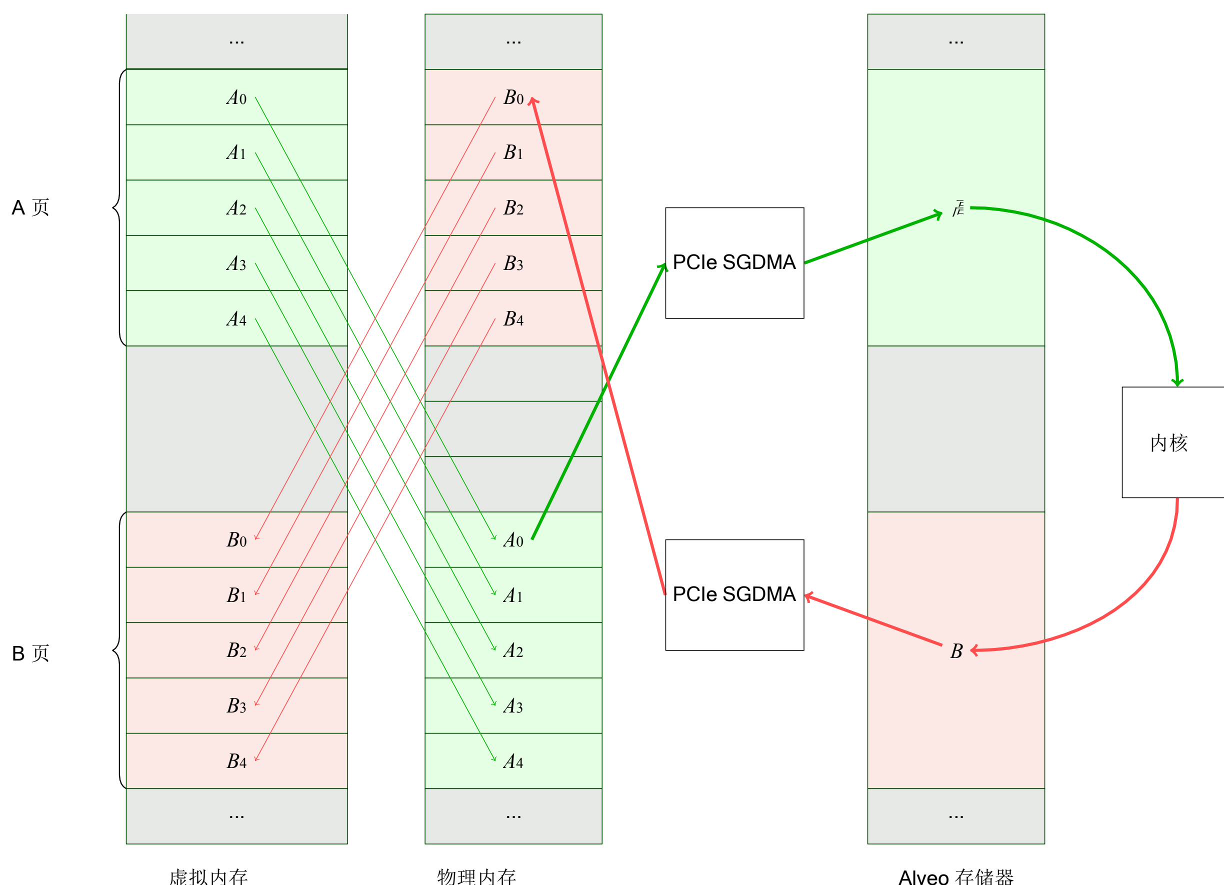

图 3.2：虚拟内存与 Alveo 之间的数据传输

您可能不禁要问，如果缓存不是从对齐的 4k 页边界开始，会发生什么情况？问得好！DMA 引擎一般要求一定程度的对齐，Alveo DMA 同样如此。如果您分配的内存没有对齐页边界，*那么运行时会为您对齐*。否则会引起计算成本较大的 **memcpy() 操作**。此外，您还将看到运行时发出警告（如果您没有关闭警告），因为这是一个需要尽快解决的问题。识别和修复这个问题将在后文讲解。

最后，我们必须确认我们的内存是**可缓存**内存还是**不可缓存**内存。因为外部存储器访问成本较大，所以几乎所有的现代处理器都内置时延极低的内部数据高速缓存。取决于处理器架构，高速缓存容量也在几十千字节到数兆字节不等。这种内部高速缓存存储器根据需要与外部物理内存同步。一般情况下，这一高速缓存管理对运行在处理器上的软件不可见。使用高速缓存的优势在于能够加快执行速度，但从一般开发角度来看，不必与它进行交互。

但是在使用 DMA 时需要注意的是，如果没有使用 CCIX 等高速缓存一致性共享技术，您必须确保处理器要共享给加速器的任何数据应在传输前首先与外部存储器同步。开始 DMA 传输前，您需要确保驻留在高速缓存中的数据已经**刷新**到外部存储器。类似地，一旦数据传回，您必须确保驻留在高速缓存中的数据已**失效**，从而方便从外部存储器刷新。x86 处理器处理这类操作速度极快，且运行时能够透明地进行处理。但其他架构的高速缓存管理可能造成性能劣化。为减轻性能劣化，该 API 提供分配和使用非高速缓存的功能。但必须注意的是，处理器在访问非高速缓存中的数据时，速度一般比首先运行高速缓存管理操作要慢得多。这种操作一般用于处理器为缓存区进行排序，但实际上并不访问其中存储的数据的情况。

Alveo 引导软件介绍

这部分要求大量背景知识！在开始介绍引导软件示例前，我们首先快速回顾要点。

- 加速一般为实现以下两个目的：**更快**完成同一项任务（阿姆达尔定律）；在**相同的时间内**完成更多任务（古斯塔夫森定律）。对于如何优化，以及优化的效果如何，每种实现方案有不同的意义。
- 本质上，加速会带来“加速代价”。为了在真实系统中实现加速，加速带来的优势必须远大于数据传输产生的额外时延。您得选好战场，然后聚焦于算法中效费比最高的部分（性能瓶颈）。
- 与 Alveo 卡的交互是通过 XRT 和 OpenCL 等更高级 API 抽象实现的。软件侧优化通过库实现，独立于硬件内核优化。
- 内存分配与管理能对整体应用性能产生重大影响。我们将在示例中探索这些主题。

引导软件示例

提供的设计文件

在本示例系列的安装包中，您可以看到两个主目录：**doc** 和 **examples**。doc 目录存放该文档的源文件，**examples** 目录存放所有构建示例和运行示例所需的源文件（除了您必须自行安装的 SDAccel、XRT 和 Alveo 开发 Shell 等构建工具）。

在 **examples** 目录下，有两个主目录：**hw_src** 和 **sw_src**。正如目录名所示，它们用来存放我们应用的软硬件源文件。硬件源文件通过赛灵思 **XOCC** 编译器综合成在 FPGA 上运行的算法。软件源文件采用标准 GCC 完成编译，以便在主机处理器上运行。

在本教程中，我们更侧重于软件而非硬件，因此我们将源文件分开以便组织。在真实的项目案例中，可以任意采用目录结构；您可以遵循您团队或机构的最佳实践。

因为某些示例需要使用外部库，所以我们对软件例使用 **CMake** 构建系统，因其可简化环境配置。但在硬件侧，我们使用标准的 **make**。这样您就可以很容易地看到传递给 XOCC 的命令行参数。

硬件设计设置

本指南将介绍适用于 Alveo 卡的加速概念。我们将从编写主机代码开始：为 FPGA 编程、分配内存并迁移内存。在早期示例中，我们的加速器非常简单。实际上，我们很可能看到算法在 CPU 上运行得更快，至少一开始是这样，因为我们的加速硬件规模太小。

此外，构建硬件设计还可能耗用大量时间。但实际上在数十亿个具有亚纳秒时序的晶体管上综合、布局和布线定制逻辑，只比编译机器代码*略*复杂一点。请耐心点，因为结果是值得的。为避免不必要地重构 FPGA 硬件，我们提供配备大量内核实例的单个 FPGA 设计。我们将在示例设计中混合匹配这些实例。本指南将略加探讨内核优化，但超过基本概念的内容，请参阅赛灵思目录中的其他文档。

本入门示例附带预构建硬件镜像，适用于 Shell 版本 **201830.1** 的 Alveo U200 加速器卡。如果您有该卡且版本符合，就无需设置硬件，可直接进入软件教程。

如果您以上两者缺一或均缺，则需要在运行前（无需担心，示例软件仍可工作）首先构建硬件设计。为此请转到如下目录：

```
onboarding/examples/hw_src
```

如果您安装在电路板上的 Shell 版本不同，则可跳过这步。如果您指向的平台并非 Alveo U200，那么请打开 Makefile 并更改第一行，指向您平台的

.xpfm 文件。

注释：为了使格式清晰，以下示例中添加了换行符。请勿向您 Makefile 中的路径添加换行符。

```
PLATFORM :=
            /opt/xilinx/platforms/xilinx_u200_xdma_201830_1
            / xilinx_u200_xdma_201830_1.xpfm
```

在编辑 Markefile 后，请确保您的 SDAccel 和 XRT 环境设置正确。如果您还没有编辑，则运行下列命令：

```
source
/opt/Xilinx/SDx/2018.3/settings.sh
source /opt/xilinx/xrt/setup.sh
```

如果 XRT 或 SDAccel 的安装路径不同于默认路径，则对命令行进行相应更新。然后，运行如下命令：

```
make
```

构建过程将会花费一定时间，最终将在本目录里产生名为 **alveo_examples.xclbin** 的文件。该文件内部包含所有在这次练习过程中将使用到的内核。一旦面向电路板和 Shell 完成该文件的编译，您就可以进行到下一部分。

构建软件设计

为避开系统特有的相依关系，在这些示例中，我们将使用 CMake 构建所有测试应用。所有示例专用代码均保存在 **sw_src** 目录下。每个源文件都根据它所对应的示例命名：例如，**00_load_kernels.cpp** 对应于示例 0。还有一些附加的“辅助”文件在应用之间共享，并被编译到它们自己的库中。

要构建源，首先应确保（对于硬件）在环境中正确设置 XRT。环境变量 $XILINX_XRT 应指向您的 XRT 安装区域。如果没有，运行（假定已将 XRT 安装到 /opt/xilinx/xrt）：

```
source /opt/xilinx/xrt/setup.sh
```

然后，完成 XRT 配置后，创建并转到构建目录：

```
cd
onboarding/examples
mkdir build
cd build
```

在构建区域内运行 `CMake`，配置构建环境。然后运行 `Make`，构建全部示例：

```
cmake ..
make
```

完成上一步操作后，会为不同编号的示例生成对应的可执行文件以及 **alveo_examples.xclbin** 文件的副本。

示例 0：加载 Alveo 镜像

简介

这里的第一个例子展示了如何将镜像加载到 Alveo 卡。在系统上电时，Alveo 卡将初始化它的 Shell（参见图 2.1）。正如前文介绍的，Shell 用于实现对主机 PC 的连接，但大部分逻辑保留空白，供您构建设计使用。在我们能将此逻辑用于应用前，我们必须首先配置逻辑。

也正如前文所述，部分操作固有更大的时延“开销”。配置 FPGA 是应用流程中耗时最大的部分之一。要直观地了解耗时*情况*，让我们尝试加载一个镜像。

关键代码

在本例中，我们为 XRT 初始化 OpenCL 运行时 API，创建命令队列，接着完成最重要的操作——配置 Alveo 卡 FPGA。该操作一般是一次性操作：一旦卡完成配置，一般将保持配置后状态，直至断电或用其他应用进行重配置。请注意，如果有多个独立应用试图将硬件加载到卡上，在头一个应用释放控制前，下一个应用将被拦阻。虽然多个独立应用可以共享运行在一张卡上的*相同镜像*。

首先，必须加入列表 3.1 列出的报头。请注意，该文档中的行次对应于文件 **00_load_kernels.cpp** 中的行次。

列表 3.1：XRT 和 OpenCL 报头

```
// Xilinx OpenCL and XRT includes
#include "xcl2.hpp"

#include <CL/cl.h>
```

在这两个中，只需要 `CL/cl.h`，xcl2.hpp 是由赛灵思提供的辅助功能库，内有所需的部分初始化功能。

一旦加入正确的报头，我们需要初始化命令队列，加载二进制文件，并将它编程到 FPGA，如列表 3.2 所示。这实际上是样板代码，基本上每个程序在某种情况下都会使用。

列表 3.2：XRT 和 OpenCL 报头

```
// This application will use the first Xilinx device found in the system
std::vector<cl::Device> devices = xcl::get_xil_devices();
cl::Device device              = devices[0];

cl::Context context(device);
cl::CommandQueue q(context, device);

std::string device_name    = device.getInfo<CL_DEVICE_NAME>();
std::string binaryFile     = xcl::find_binary_file(device_name, argv[1]);
```

```
cl::Program::Binaries bins = xcl::import_binary_file(binaryFile);

devices.resize(1);
cl::Program program(context, devices, bins);
```

可以将这里的工作流总结如下：

1. （33-34 行）：发现系统中的赛灵思器件并编号。在本例中，我们假设器件 0 是目标卡，但如果使用多加速器卡系统，就必须为它们编号以便识别。
2. （36-37 行）：初始化 OpenCL 环境与命令队列。
3. （39:41 行）：加载指向我们的 Alveo 卡的二进制文件。在这些示例中，我们使用命令行传递文件名，不过这也能硬编码或在应用专用例中另行处理。
4. （44 行）编程 FPGA。

44 行是实际触发编程操作的地方。在编程阶段，运行时将检查 Alveo 卡的当前配置。如果已经编程，在从 xclbin 加载器件元数据后就可以返回。如果没有编程，就立即编程器件。

运行应用

在 XRT 初始化后，从构建目录运行下列命令以运行应用：

```
./00_load_kernels alveo_examples
```

程序将输出类似于下列内容的消息：

```
-- Example 0: Loading the FPGA Binary --

Loading XCLBin to program the Alveo

board: Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO:
Importing ./alveo_examples.xclbin
Loading: ’ ./alveo_examples.xclbin’

FPGA programmed, example complete!

-- Key execution times --
OpenCL Initialization : 1624.634 ms
```

请注意，我们的 FPGA 用时 1.6 *秒*完成初始化。注意这个内核的加载时间；它包含有磁盘 I/O、PCIe 时延、配置开销和一些其他操作。一般您应该在应用启动时间内配置或预配置 FPGA。我们使用已经加载的比特流再次运行该应用：

```
-- Key execution times --
OpenCL Initialization : 262.374 ms
```

.26 耗时*远*低于 1.6 秒！我们仍然需要从磁盘读回文件、解析文件，并确保加载到 FPGA 中的 xclbin 是正确的，但总体初始化时间显著下降。

补充练习

本实验基础上的补充练习：

- 使用 **xbutil** 实用工具查询电路板。您能否看到加载的是哪一个 .xclbin 文件？
- 再次使用 **xbutil**，FPGA 上驻留的是哪些内核？您是否看到 FPGA 编程前后的差异？

要点总结

- FPGA 是一项高开销的操作。在理想情况下，应在使用前尽早初始化 FPGA。这可以通过您应用中的其他初始化任务，通过单独的线程完成，或是对于专用系统，在系统启动时完成。
- 一旦 FPGA 完成加载，后续加载速度会明显加快。

现在我们可以加载镜像到 FPGA，运行某些任务！

示例 1：简单内存分配

简介

我们加载的 FPGA 镜像内置非常简单的向量加法内核。它以两个任意长度的缓存作为输入，并生成一个相同长度的缓冲器区作为输出。正如它的名称，在这个过程中它完成两者的相加操作。

我们的代码还没有针对在 FPGA 上良好运行进行真正的优化。它大致相当于将列表 3.3 中的算法直接放到 FPGA 架构中。这样做效率不高。我们在每个时钟周期可以完成一次加法运算，但每次只能处理一个 32 位输出。

列表 3.3：向量加法算法

```
void vadd_sw(uint32_t *a, uint32_t *b, uint32_t *c, uint32_t size)
{
    for (int i = 0; i < size; i++)
        { c[i] = a[i] + b[i];
    }
}
```

特别值得注意的是，此时*该代码完全不能超越处理器性能*。FPGA 架构的时钟明显慢于 CPU 时钟。但这并不令人意外：试想我们最早的举例，我们列车的每节车厢只搭载了一位乘客。我们还有 PCIe 数据传输、DMA 设置等产生的开销。在下面的几个示例中，我们将了解如何高效率地为我们这项功能的输入输出管理缓存。只有在这一步之后，我们才能充分发挥从 Alveo 卡获得的加速效果。

关键代码

虽然不大，但本示例却是我们首次在 FPGA 上运行任务。要在 Alveo 卡上运行任务，必须要做四件事情：

1. 分配和填充用于收发卡数据的缓存。
2. 在主机内存空间和 Alveo 全局内存间传输缓冲器。
3. 运行内核以操作这些缓存。
4. 将内核运算的结果传输回主机内存空间，以便处理器访问。

您可以发现，这些操作中只有一件真正发生在加速卡上。存储器管理将决定您应用性能的成败。下面让我们了解这个方面的内容。

如果以前您未曾接触过加速功能，您可能会盲目地常规调用 *malloc() 或 new* 来分配内存。在本例中我们就这样操作，为主机和 Alveo 卡之间的传输分配一系列缓存。我们将分配四个缓存：两个加法用输入缓存，一个 Alveo 用输出缓存，以及一个额外的缓存，用于 vadd 功能的软件实现。这有助于我们

了解到一些有趣的地方：我们为 Alveo 分配内存的方式如何影响处理器的运行效率。

对缓存进行简单的分配，如列表 3.4 所示。在本例中，`BUFSIZE` 为 24 MiB，相当于 uint32_t 的 6 × 1024 × 1024 倍。此处任何未提及的代码或与前例完全相同，或功能等效。

列表 3.4：简单缓存分配

```
uint32_t *a = new uint32_t[BUFSIZE];
uint32_t *b = new uint32_t[BUFSIZE];
uint32_t *c = new uint32_t[BUFSIZE];
uint32_t *d = new uint32_t[BUFSIZE];
```

这个操作将分配**虚拟**内存、**页式**内存以及最重要的**未对齐**内存。特别是最后一个特点将导致某些问题。我们很快就会发现。

一旦完成缓存分配并用初始测试向量填充缓存，下一加速步骤便是将它们发送到 Alveo 全局内存。我们的操作方法是用 **CL_MEM_USE_HOST_PTR** 旗标创建 OpenCL 缓存对象。这样做的目的是告诉 API，无需分配它自己的缓存，我们就能提供我们的指针。这样做并不算错，但因为没有管理指针分配，它将劣化性能。

列表 3.5 中所列的代码用于将分配的缓存映射到 OpenCL 缓存对象。

列表 3.5：使用主机内存指针映射 OCL 缓冲器

```
std::vector<cl::Memory> inBufVec, outBufVec;
cl::Buffer a_to_device(context,
                       static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                                 CL_MEM_USE_HOST_PTR),
                       BUFSIZE * sizeof(uint32_t),
                       a,
                       NULL);
cl::Buffer b_to_device(context,
                       static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                                 CL_MEM_USE_HOST_PTR),
                       BUFSIZE * sizeof(uint32_t),
                       b,
                       NULL);
cl::Buffer c_from_device(context,
                         static_cast<cl_mem_flags>(CL_MEM_WRITE_ONLY |
                                                   CL_MEM_USE_HOST_PTR),
                         BUFSIZE * sizeof(uint32_t),
                         c,
                         NULL);
inBufVec.push_back(a_to_device);
inBufVec.push_back(b_to_device);
outBufVec.push_back(c_from_device);
```

这里的操作是分配由该 API 识别的 `cl::Buffer` 对象，并传入来自我们之前分配的缓存的指针 `a`、`b` 和 `c`。新增旗标 `CL_MEM_READ_ONLY` 和 `CL_MEM_WRITE_ONLY` 向运行时说明这些缓存对内核而言的可见性。换句话说，主机写入加速卡的 *a* 和 *b* 对内核而言**只读**。随后，主机从加速卡读回 *c*。对内核而言，它是**只写**。我们额外地将这些缓存对象添加给向量的目的在于能一次性传输多个缓存（请注意，我们只给向量添加*指针*，并非添加数据缓存本身）。

接下来，我们使用列表 3.6 中的代码，将输入缓存传输给 Alveo 卡。

列表 3.6：迁移主机内存到 Alveo

```
cl::Event event_sp;
q.enqueueMigrateMemObjects(inBufVec, 0, NULL, &event_sp);
clWaitForEvents(1, (const cl_event *)&event_sp);
```

在这个代码段中，“主事件”是对 108 行的 `enqueueMigrateMemObjects()` 的调用。我们传入缓存向量。0 代表这是主机到器件的传输。此外，我们还传入 `cl::Event` 对象。

这时正好简单了解一下同步。在我们为传输排队时，我们可以将它添加到运行时的“待运行清单”，但并不真的要等到清单走完。通过寄存 `cl::Event` 对象，我们可以决定等到未来任一时点触发事件。在需要等待时，这一般不是问题。但是我们要在代码中的各点等待，才能更方便地指示代码显示各操作的用时。这样做也会略为增大应用的开销。不过这是一个学习练习，并非是为极致性能进行优化的案例。

我们现在需要告诉运行时将什么传递给我们的内核，方式请参见列表 3.7。回忆一下，我们的参数列表外观如下：

```
(uint32_t *a, uint32_t *b, uint32_t *c, uint32_t size)
```

在我们的案例中，a 是参数 0，b 是参数 1，依次类推。

列表 3.7：设置内核参数

```
krnl.setArg(0, a_to_device);
krnl.setArg(1, b_to_device);
krnl.setArg(2, c_from_device);
krnl.setArg(3, BUFSIZE);
```

接下来，将内核本身添加到命令队列，以便开始执行。一般来说，我们会将传输与内核加入到队列，让它们依次执行，而非同步执行。将内核执行添加到命令队列的代码请参见列表 3.8。

列表 3.8：为内核运行排队

```
q.enqueueTask(krnl, NULL, &event_sp);
```

如果不想在这一点等待，可以传入 NULL 而非 `cl::Event` 对象。

最后，一旦内核运行完成，我们要将内存传输回主机，以便 CPU 访问新的值。操作方法请参见列表 3.9。

列表 3.9：将数据传输回主机

```
q.enqueueMigrateMemObjects(outBufVec, CL_MIGRATE_MEM_OBJECT_HOST, NULL, &event_sp);
clWaitForEvents(1, (const cl_event *)&event_sp);
```

在这个实例中，我们*需要*等待，以便进行同步。这一点很重要。记得当我们调用这些排队功能时，我们以**非阻塞**方式向命令队列添加记录。如果在完成传输排队后我们要立即访问缓存，缓存已经完成读回操作。

除了示例 0 中的 FPGA 配置，为运行内核而新增的内容有：

1. （60-63 行）：以正常方式分配缓存。我们很快将介绍更好的分配方式，但这种方式是众多首次实验加速功能的人的首选。

2. （81-102 行）：将分配的缓存映射到 `cl::Buffer` 对象。
3. （108 行）：为迁移输入缓存（a 和 b）到 Alveo 器件的全局内存排队。
4. （113-116 行）：设置内核参数，包括缓存和标量值。
5. （120 行）：运行内核。
6. （126-127 行）：将内核结果读回到 CPU 主机内存，在完成读取时同步。

如果是真实应用，只需要一次同步。如前文所述，我们使用多次同步以更准确地报告工作流中各项操作的用时。

运行应用

在 XRT 初始化后，从构建目录运行下列命令以运行应用：

```
./01_simple_malloc alveo_examples
```

程序将输出类似于下列内容的消息：

```
-- Example 1: Vector Add with Malloc() --

Loading XCLBin to program the Alveo board:

Found Platform
Platform Name:
Xilinx
XCLBIN File Name: alveo_examples
INFO:
Importing ./alveo_examples.xclbin
Loading:' ./alveo_examples.xclbin'
Running kernel test with malloc()ed buffers
WARNING: unaligned host pointer ' 0x154f7909e010'
detected, this leads to extra memcpy
WARNING: unaligned host pointer ' 0x154f7789d010'
detected, this leads to extra memcpy
WARNING: unaligned host pointer ' 0x154f7609c010'
detected, this leads to extra memcpy

Simple malloc vadd example complete!

--------------- Key execution times --------------
- OpenCL Initialization              :  247.371 ms
Allocating memory buffer             :    0.030 ms
Populating buffer inputs             :   47.955 ms
Software VADD run                    :   35.706 ms
Map host buffers to OpenCL buffers :   64.656 ms
Memory object migration enqueue      :   24.829 ms
Set kernel arguments                 :    0.009 ms
OCL Enqueue task                     :    0.064 ms
Wait for kernel to complete          :   92.118 ms
Read back computation results        :   24.887 ms
```

请注意，有部分警告提示未对齐主机指针。因为我们没有管理缓存分配，所以用于传输 Alveo 数据的缓存未与 Alveo DMA 引擎要求的 4KiB 边界对齐。因此，我们需要复制缓存内容，在传输前先行对齐。这项操作开销很大。

从我们示例的这一点起，我们将深入讲解这方面的指标。虽然每次运行的时延多少有些变化，但一般情况下我们要关注的是每个具体领域的变化。现在先用表 3.1 确立基线。

表 3.1：用时总结 - 示例 1

运行	示例 1
OCL 初始化	247.371 ms
缓存分配	30 µs
缓存填充	47.955 ms
软件 VADD	35.706 ms
缓存映射	64.656 ms
写出缓存	24.829 ms
设置内核参数	9 µs
内核运行时	92.118 ms
读入缓存	24.887 ms
$\Delta_{Alveo \rightarrow CPU}$	−418.228 ms
$\Delta_{Alveo \rightarrow CPU}$（仅算法）	−170.857 ms

显然这个指标并不理想。但我们是否要准备放弃？什么？赛灵思制造这样的产品总有什么理由。那么接下来让我们了解一下怎样提高性能！

补充练习

本实验基础上的补充练习：

- 改变分配的缓存的容量。您能否推导出缓存容量与单次操作用时之间的大致关系？它们是否以同样的比例缩放？
- 如果取消步骤间同步，对运行用时会产生怎样的量化影响？
- 如果在最后一个缓冲器从 Alveo 复制回主机后取消同步，会发生什么情况？

要点总结

- 我们还是必须承担 FPGA 配置“代价”。与 CPU 相比，我们必须省出至少 250 mS 的时间用于配置 FPGA。请注意，如果只是要处理单个缓存，我们这个小示例的性能***永远无法***超越 CPU。
- 简单分配的内存并不是传递给加速器的理想选择，因为必须进行内存复制以解决对齐问题。我们将在后续示例中探讨其影响。
- OpenCL 以命令队列的方式工作。开发者可以选择同步方式和同步时间，但在从 Alveo 全局内存读回缓存时必须注意，应确保在 CPU 访问缓存内数据之前完成同步。

示例 2：对齐内存分配

简介

在前面的示例中，我们只对内存做了简单分配，但是 DMA 引擎要求缓存与 4 KiB 页边界对齐。如果缓存未对齐（如果没有明确要求，很可能发生这种情况），运行时就会复制缓存，对齐其内容。

这是一项高开销的操作，但我们能否量化此开销？我们该如何分配对齐的内存？

关键代码

这是一个比较简短的示例，我们只改动了**示例 1** 中的四行，我们的缓存分配。有多种方法可以分配对齐的存储器，但本例中我们将使用 POSIX 函数——`posix_memalign()`。这次改动使用列表 3.10 中的代码替换列表 3.4 中的分配。还要增添一个列表 3.4 中没有的报头，**Memory**。

列表 3.10：分配对齐的缓存

```
uint32_t *a, *b, *c, *d = NULL;
posix_memalign((void **)&a, 4096, BUFSIZE * sizeof(uint32_t));
posix_memalign((void **)&b, 4096, BUFSIZE * sizeof(uint32_t));
posix_memalign((void **)&c, 4096, BUFSIZE * sizeof(uint32_t));
posix_memalign((void **)&d, 4096, BUFSIZE * sizeof(uint32_t));
```

注意我们对 `posix_memalign()` 的调用。我们传入要求的对齐，即前文所述的 `4 KiB`。

然而，这是与示例 1 相比对代码的唯一改动。请注意，我们更改了所有缓存的分配，包括只有 CPU 基本功能 VADD 使用的缓存 *d*。我们将查看是否这次更改能给加速器和 CPU 的运行时性能造成影响。

运行应用

在 XRT 初始化后，从构建目录运行下列命令以运行应用：

```
./02_aligned_malloc alveo_examples
```

程序将输出类似于下列内容的消息：

```
-- Example 2: Vector Add with Aligned Allocation -

- Loading XCLBin to program the Alveo board:

Found Platform
Platform Name:Xilinx
XCLBIN File Name: alveo_examples
INFO:
Importing ./alveo_examples.xclbin
Loading:' ./alveo_examples.xclbin'
Running kernel test with aligned virtual

buffers Simple malloc vadd example complete!

--------------- Key execution times --------------
- OpenCL Initialization              :  256.254 ms
Allocating memory buffer             :    0.055 ms
Populating buffer inputs             :   47.884 ms
Software VADD run                    :   35.808 ms
Map host buffers to OpenCL buffers  :    9.103 ms
Memory object migration enqueue      :    6.615 ms
Set kernel arguments                 :    0.014 ms
OCL Enqueue task                     :    0.116 ms
Wait for kernel to complete          :   92.110 ms
Read back computation results        :    2.479 ms
```

这一眼看上去就改善许多！让我们将这些结果与**示例 1** 的结果做对比，了解具体有什么变化。请查看表 3.2 了解详情。请注意，为了让比较更加清晰，我们从比较中剔除了轻微的运行间变化。

表 3.2：用时总结 - 示例 2

运行	示例 1	示例 2	$\Delta_{1\to2}$
OCL 初始化	247.371 ms	256.254 ms	-
缓存分配	30 µs	55 µs	25 µs
缓存填充	47.955 ms	47.884 ms	-
软件 VADD	35.706 ms	35.808 ms	
缓存映射	64.656 ms	9.103 ms	−55.553 ms
写出缓冲器	24.829 ms	6.615 ms	−18.214 ms
设置内核参数	9 µs	14 µs	-
内核运行时	92.118 ms	92.110 ms	-
读入缓存	24.887 ms	2.479 ms	−22.408 ms
$\Delta_{Alveo\to CPU}$	−418.228 ms	−330.889 ms	87.339 ms
$\Delta_{Alveo\to CPU}$（仅算法）	−170.857 ms	−74.269 ms	96.588 ms

好极了！仅更改四行代码，我们就让执行时间减少近 100 ms。CPU 速度仍然更快，但是只更改一个关于内存分配方式的小地方，我们就看到大幅改善。导致大量开销的是对齐所需的内存复制。在分配缓存时多花几毫秒对齐它们，在使用缓存时就能省下大量时间。

此外，还应注意软件运行时间保持不变，这符合本用例的要求。我们将更改已分配内存的对齐方式，而不是进行正常用户空间的内存对齐。

补充练习

本实验基础上的补充练习：

- 再次改变分配的缓存的容量。您从前面示例中总结的关系是否依然成立？
- 尝试用其他方法分配对齐内存（不使用 OCL API）。在轻微的运行间变化之外，您是否观察到方法间的不同？

要点总结

- 未对齐内存将严重劣化您的器件性能。始终确保与 Alveo 卡共享的缓存对齐。

现在我们已经有了一定基础！尝试使用 OpenCL API 分配内存并查看结果。

示例 3：使用 OpenCL 分配内存

简介

确保分配的内存对齐页边界，与最初配置相比可带来显著改善。不过在使用 OpenCL 时还可以用另一个工作流，即让 OpenCL 和 XRT 分配内存，然后将它们映射到用户空间指针，供应用使用。下面实验这种方式，查看它对我们时序的影响。

关键代码

从概念上讲，这是一个小改动，但与**示例 2** 相比，这个改动涉及更多的代码调整。这是因为我们没有使用标准的用户空间内存分配，而是让 OpenCL 运行时为我们分配缓存。一旦分配好缓存，我们需要将它们映射到用户空间，才能访问它们存储的数据。

在缓存分配方面，我们将列表 3.10 改为列表 3.11。

列表 3.11：使用 OpenCL 分配对齐的缓存

```
std::vector<cl::Memory> inBufVec, outBufVec;
cl::Buffer a_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                           CL_MEM_ALLOC_HOST_PTR),
                 BUFSIZE * sizeof(uint32_t),
                 NULL,
                 NULL);
cl::Buffer b_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                           CL_MEM_ALLOC_HOST_PTR),
                 BUFSIZE * sizeof(uint32_t),
                 NULL,
                 NULL);
cl::Buffer c_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_WRITE_ONLY |
                                           CL_MEM_ALLOC_HOST_PTR),
                 BUFSIZE * sizeof(uint32_t),
                 NULL,
                 NULL);
cl::Buffer d_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_READ_WRITE |
                                           CL_MEM_ALLOC_HOST_PTR),
                 BUFSIZE * sizeof(uint32_t),
                 NULL,
                 NULL);
inBufVec.push_back(a_buf);
inBufVec.push_back(b_buf);
outBufVec.push_back(c_buf);
```

在本例中，我们在程序中早期分配了 OpenCL 缓存对象，而且也还没有用户空间指针。但我们仍可以将缓存对象用于 MigrateMemObjects() 及

其他 OpenCL 函数的队列。此时，也分配了备份存储器，但我们未赋予它用户空间指针。

对 `cl::Buffer` 构造器的调用看似很像前面的操作。实际上只做了两处更改：我们传入旗标 `CL_MEM_ALLOC_HOST_PTR` 而非 `CL_MEM_USE_HOST_PTR`，告知运行时我们想要分配缓存，而不是使用既有的缓存。也不再需要传入指针给用户缓存（因为我们分配了一个新的缓存），所以传递 NULL 作为替代。

然后要将 OpenCL 缓存映射到我们马上在软件中使用的用户空间指针 *a*、*b* 和 *d*。此时不必将指针映射到 c，在内核执行完毕后需从缓存读取时，再映射到 c。我们使用列表 3.12 中的代码达成这个目的。

列表 3.12：将分配的对齐缓存映射到用户空间指针

```
uint32_t *a = (uint32_t *)q.enqueueMapBuffer(a_buf,
                                             CL_TRUE,
                                             CL_MAP_WRITE,
                                             0,
                                             BUFSIZE * sizeof(uint32_t));
uint32_t *b = (uint32_t *)q.enqueueMapBuffer(b_buf,
                                             CL_TRUE,
                                             CL_MAP_WRITE,
                                             0,
                                             BUFSIZE * sizeof(uint32_t));
uint32_t *d = (uint32_t *)q.enqueueMapBuffer(d_buf,
                                             CL_TRUE,
                                             CL_MAP_WRITE | CL_MAP_READ,
                                             0,
                                             BUFSIZE * sizeof(uint32_t));
```

一旦完成映射，我们可以正常使用用户空间指针访问缓存内容。但需要注意，OpenCL 运行时确实会对打开的缓存进行*引用计数*，所以对我们映射的每个缓存需要相应地调用 `enqueueUnmapMemObject()`。

内核的执行流程不变，但如果将输入缓存迁移回器件，就可以发现有所不同。无需手动排队迁移，我们就可以直接映射缓存。OpenCL 运行时将觉察缓存内容目前驻留在 Alveo 器件的全局内存中，并将为我们管理将缓存迁移回主机的操作。这是一个您必须做出的代码风格选择，但从根本上讲，列表 3.13 中的代码足以将 *c* 迁移回主机内存。

列表 3.13：将内核输出映射到用户空间指针

```
uint32_t *c = (uint32_t *)q.enqueueMapBuffer(c_buf,
                                             CL_TRUE,
                                             CL_MAP_READ,
                                             0,
                                             BUFSIZE * sizeof(uint32_t));
```

最后正如前文所述，您需要分别列出内存对象，以便运行时彻底删除它们。我们会在程序结束时列出，而不是像之前对缓存使用 `free()`。这一步必须在命令队列结束前完成，如列表 3.14 所示。

列表 3.14：分别列出 OpenCL 分配的缓存

```
q.enqueueUnmapMemObject(a_buf, a);
q.enqueueUnmapMemObject(b_buf, b);
q.enqueueUnmapMemObject(c_buf, c);
q.enqueueUnmapMemObject(d_buf, d);
```

```
q.finish();
```

总结这一用例的主要工作流时，请注意下列代码行：

1. （63-90 行）：使用 CL_MEM_ALLOC_HOST_PTR 旗标**分配**缓存。
2. （94-108 行）：将输入缓存**映射**到用户空间指针以填充它们。
3. 像往常一样运行内核
4. （144-148 行）：映射输出缓存，将它们迁移回主机内存。
5. （181-185 行）：使用完毕后分别列出我们所有的缓存，以便正确地删除它们。

运行应用

在 XRT 初始化后，从构建目录运行下列命令以运行应用：

```
./03_buffer_map alveo_examples
```

程序将输出类似于下列内容的消息：

```
-- Example 3: Allocate and Map Contiguous Buffers -

- Loading XCLBin to program the Alveo board:

Found Platform
Platform Name:Xilinx
XCLBIN File Name: alveo_examples
INFO:
Importing ./alveo_examples.xclbin
Loading:' ./alveo_examples.xclbin'
Running kernel test with XRT-allocated contiguous

buffers OCL-mapped contiguous buffer example complete!

--------------- Key execution times ---------------
OpenCL Initialization               :  247.460 ms
Allocate contiguous OpenCL buffers :   30.365 ms
Map buffers to userspace pointers   :    0.222 ms
Populating buffer inputs            :   22.527 ms
Software VADD run                   :   24.852 ms
Memory object migration enqueue     :    6.739 ms
Set kernel arguments                :    0.014 ms
OCL Enqueue task                    :    0.102 ms
Wait for kernel to complete         :   92.068 ms
Read back computation results       :    2.243 ms
```

表 3.3：用时总结 - 示例 3

运行	示例 2	示例 3	$\Delta_{2\rightarrow3}$
OCL 初始化	256.254 ms	247.460 ms	-
缓存分配	55 µs	30.365 ms	30.310 ms
缓存填充	47.884 ms	22.527 ms	−25.357 ms
软件 VADD	35.808 ms	24.852 ms	−10.956 ms
缓存映射	9.103 ms	222 µs	−8.881 ms
写出缓存	6.615 ms	6.739 ms	-
设置内核参数	14 µs	14 µs	-
内核运行时	92.110 ms	92.068 ms	-
读入缓存	2.479 ms	2.243 ms	-
$\Delta_{Alveo\rightarrow CPU}$	−330.889 ms	−323.996 ms	−6.893 ms
$\Delta_{FPGA\rightarrow CPU}$（仅算法）	−74.269 ms	−76.536 ms	-

您可能本来预计这里会实现速度提升，但我们看到并没有为任何特定运算加速，只是在系统中变化时延。实质上，我们从不同的银行账户付税，但到日终我们还得承担。在为处理器和内核提供统一存储器映射的嵌入式系统上，我们可以看到显著差异，但在服务器级的 CPU 上不会。

需要考虑的地方是，虽然用这种方式预先分配缓存会耗费更长时间，但您一般不会在应用的关键路径上分配缓存。但是使用这种方式，运行时使用缓存的速度大幅度加快。

您可能想知道为什么 CPU 访问这种内存的速度更快。虽然到目前阶段我们还没有探讨这个话题，但是可通过这个 API 分配内存，将虚拟地址*固定*到物理内存中。这样 CPU 和 DMA 访问它的效率显著提高。但和工程设计中的任何情况一样，这种做法也有代价。分配时间较长，而且如果分配众多小缓存，也存在可用内存碎片化的风险。

一般情况下，应避免在应用的关键路径上分配缓存。如果使用得当，这种方法能将负担转移出您的高性能区域。

补充练习

本实验基础上的补充练习：

- 再次改变分配的缓存的容量。您从前面例子中总结的关系是否依然成立？
- 尝试使用其他序列来分配内存并将传输加入队列。
- 如修改输入缓存并再次运行内核，会发生什么情况？

要点总结

- 使用 OpenCL 和 XRT API 能够获得局部性能改善，虽然从根本上并未避开代价。

但是我们的长板是内核运行时间，我们可以轻易地为它加速。下面介绍另外几个示例。

示例 4：并行化数据路径

简介

我们的整体系统吞吐量已有显著提升，但瓶颈现在显而易见，那就是加速器本身。请注意，我们可以从两个方向优化设计：我们可以向一个问题投入更多资源，加快求解时间。这种方法受**阿姆达尔定律**制约。我们也可以根据**古斯塔夫森定律**，并行完成更多基础运算。

在本例中，我们将尝试按照阿姆达尔定律的方向优化。截至目前我们的加速器一直采用基本与 CPU 相同的算法，每个时钟周期完成一次 32 位加法运算。但是由于 CPU 的时钟速度要快得多（而且拥有不通过 PCIe 传输数据的优势），CPU 性能一直比我们高。现在是做出改变的时候了。

我们的 DDR 控制器原生拥有 512 位内部接口宽度。如果我们并行加速器中的数据流，就可以每个时钟周期处理 16 个阵列元，而非仅一个。因此，只需实现输入向量化，就能立即获得 16 倍加速。

关键代码

在本例中，如果我们继续主要关注主机软件，并将内核当作黑箱对待，我们将得到基本与**示例 3** 完全相同的代码。我们所需做的仅是将内核名称从 `vadd` 改成 `wide_vadd`。

因为代码不变，这时是引入在 XRT 和 OpenCL 中使用内存的另一种概念的良机。Alveo 卡上有四个存储器组可用。虽然对于这些简单加速器无此必要，您可能会发现，您希望将运算分散在它们上面，帮助最大化每个接口的可用带宽。对于我们的简单向量加法例（只是为了说明），我们使用图 3.3 中所示的拓扑结构。

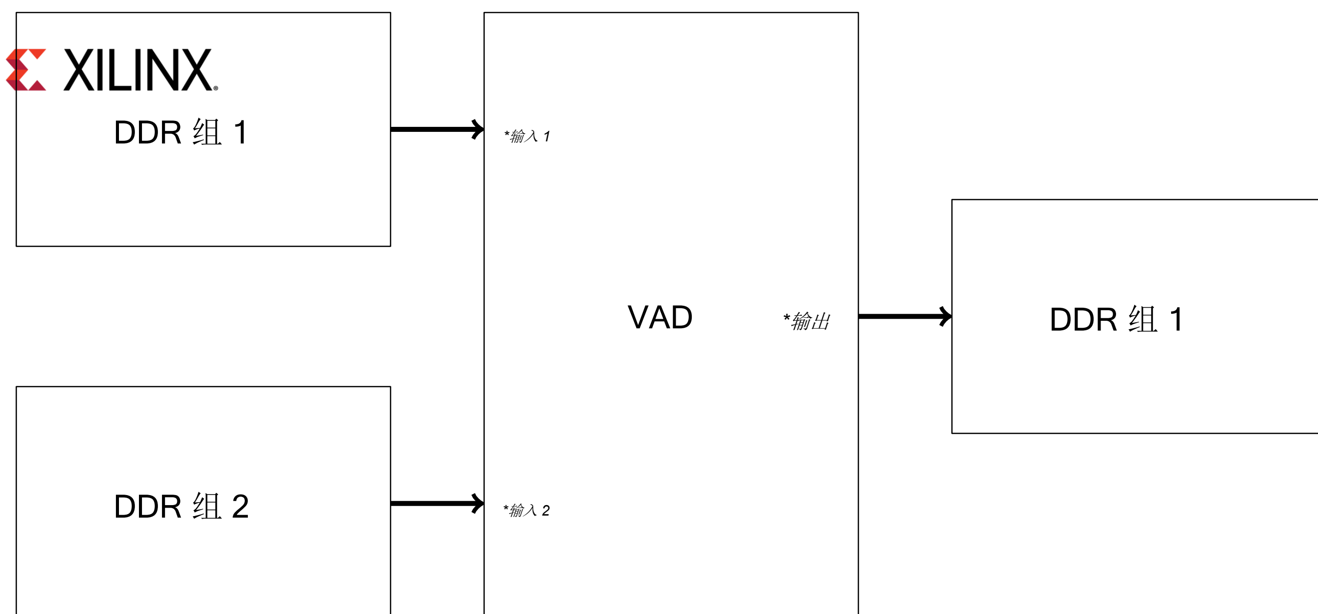

图 3.3：宽 VADD 内存连接

这样做的优势在于我们能够使用不同的外部存储器组同时执行高带宽事务。值得注意的是，长猝发对性能而言比大量小规模读写有利。但从根本上讲，不能在内存上同时执行两个操作。

通过命令行开关可以轻易地指定硬件连接，但要实际将缓存传输到硬件时，则需要指定用于 XRT 的内存。赛灵思通过扩展标准 OpenCL 库，将结构体 `cl_mem_ext_ptr_t` 与缓存分配旗标 `CL_MEM_EXT_PTR_XILINX` 相结合，提供这一功能。代码类似于列表 3.15。

列表 3.15：为 XRT 指定外部存储器组

```
cl_mem_ext_ptr_t bank1_ext, bank2_ext;
bank2_ext.flags = 2 | XCL_MEM_TOPOLOGY;
bank2_ext.obj   = NULL;
bank2_ext.param = 0;
bank1_ext.flags = 1 | XCL_MEM_TOPOLOGY;
bank1_ext.obj   = NULL;
bank1_ext.param = 0;
cl::Buffer a_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                           CL_MEM_EXT_PTR_XILINX),
                 BUFSIZE * sizeof(uint32_t),
                 &bank1_ext,
                 NULL);
cl::Buffer b_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                           CL_MEM_EXT_PTR_XILINX),
                 BUFSIZE * sizeof(uint32_t),
                 &bank2_ext,
                 NULL);
cl::Buffer c_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_READ_WRITE |
                                           CL_MEM_EXT_PTR_XILINX),
                 BUFSIZE * sizeof(uint32_t),
```

```
&bank1_ext,
NULL);
```

可以看到该代码非常类似于列表 3.11，区别在于我们将 cl_mem_ext_ptr_t 对象传递给 cl::Buffer 构造器，并使用 *flags 字段指定用于这个特定缓存的存储器组*。重申一下，对这样的简单示例完全不必这样大费周章，但由于该例结构上与之前的类似，似乎是个结合两者的良机。对于极重的工作负载，这是个优化性能的有用办法。

否则，除了切换为 wide_vadd 内核，除了将缓存容量增加到 1 GiB 以外，该代码完全没有改动。这样做的目的是强调软硬件差异。

运行应用

在 XRT 初始化后，从构建目录运行下列命令以运行应用：

```
./04_wide_vadd alveo_examples
```

程序将输出类似于下列内容的消息：

```
-- Example 4: Parallelizing the Data Path -

- Loading XCLBin to program the Alveo board:

Found Platform
Platform Name:Xilinx
XCLBIN File Name: alveo_examples
INFO:
Importing ./alveo_examples.xclbin
Loading:' ./alveo_examples.xclbin'
Running kernel test XRT-allocated buffers and wide data path:


OCL-mapped contiguous buffer example complete!

--------------- Key execution times ---------------
OpenCL Initialization              :  244.463 ms
Allocate contiguous OpenCL buffers :   37.903 ms
Map buffers to userspace pointers  :    0.333 ms
Populating buffer inputs           :   30.033 ms
Software VADD run                  :   21.489 ms
Memory object migration enqueue    :    4.639 ms
Set kernel arguments               :    0.012 ms
OCL Enqueue task                   :    0.090 ms
Wait for kernel to complete        :    9.003 ms
Read back computation results      :    2.197 ms
```

表 3.4：用时总结 - 示例 3

运行	示例 3	示例 4	$\Delta_{3\to4}$
OCL 初始化	247.460 ms	244.463 ms	-
缓存分配	30.365 ms	37.903 ms	7.538 ms
缓存填充	22.527 ms	30.033 ms	7.506 ms
软件 VADD	24.852 ms	21.489 ms	−3.363 ms
缓存映射	222 µs	333 µs	-
写出缓存	6.739 ms	4.639 ms	-
设置内核参数	14 µs	12 µs	-
内核运行时	92.068 ms	9.003 ms	−83.065 ms
读入缓存	2.243 ms	2.197 ms	-
$\Delta_{Alveo\to CPU}$	−323.996 ms	−247.892 ms	−76.104 ms
$\Delta_{FPGA\to CPU}$（仅算法）	−76.536 ms	5.548 ms	−82.084 ms

任务完成 - 我们已成功击败 CPU！

但这里有几点值得注意的地方。首先要注意的是，可能您已经知道，将数据传入传出 FPGA 的用时没有变化。正如我们在前面章节中讨论过的，这实际上是内存拓扑结构、数据量和整体系统内存带宽利用率共同决定的固定时间。虽然这里存在运行之间的微小变化，特别是在云数据中心这样的虚拟化环境中，但总体上可视为固定时间操作。

更值得一提的地方是实现了加速。您可能感到惊讶的是虽然数据路径被加宽到每时钟 16 个字，但并没有带来 16 倍提速。内核本身每时钟能处理 16 个字，但我们看到的计时结果是外部 DDR 时延的累加效应。每个接口将猝发输入数据，处理数据，猝发输出数据。但与 DDR 通信的内在延迟会减慢速度。与单个字的实现方案相比，我们实际只能实现 10 倍的速度提升。

实际上，我们遇到的是向量加法的根本问题：它太简单了。VADD 的计算复杂性是 $O(N)$，而且是非常简单的 $O(N)$。因此，很快就受到 I/O 带宽制约，而非受计算制约。在运行比较复杂的算法时，比如复杂程度为 $O(N^x)$ 的嵌套环路，或是计算上复杂的 $O(N)$ 算法，如需要大量计算的过滤器，通过在 FPGA 架构内完成更多计算，避免频繁访问存储器，能获得非常明显的加速。最适合用于加速的算法应具备低数据传输和大计算量。

此外，我们也可以使用更大的缓存运行另一个实验。更改缓存容量，如列表 3.16 所示：

列表 3.16：增大内存容量

```
#define BUFSIZE (1024 * 1024 * 256) // 256*sizeof(uint32_t) = 1 GiB
```

重新构建然后重新运行。现在切换到表 3.5 所示的简化指标集。我想现在您也已经理解 OpenCL 初始化的含义。那么我们只比较算法性能。这并非是说

初始化时间不重要，但您应在架构您的应用时让它成为其他初始化过程中发生的一次性代价。

我们将停止探讨缓存分配与初始化时间。下面这种做法是设置应用时常常爱做的事情。如果您习惯于在关键应用路径上分配大量缓存，您想以小代价获得大幅速度提升的概率就比较低。重新思考您的架构，别着急使用硬件加速。

表 3.5：1 GiB 缓存用时总结 - 示例 4

运行	示例 4
软件 VADD	820.596 ms
缓存 PCIe TX	383.907 ms
VADD 内核	484.050 ms
缓存 PCIe RX	316.825 ms
硬件 VADD（总计）	1184.897 ms
$\Delta_{Alveo \rightarrow CPU}$	364.186 ms

哦，不！我们进展如此顺利，到这里却突遭意外。使用更大的缓存，我们就完全没法击败 CPU 了。发生了什么情况？

仔细检查数值，我们可以发现虽然硬件内核本身的数据处理效率很高，但与 FPGA 间的数据往返传输占用大量时间。实际上，如果您想描绘出总体执行时间线，那么此时的情况可以用图 3.4 表示。

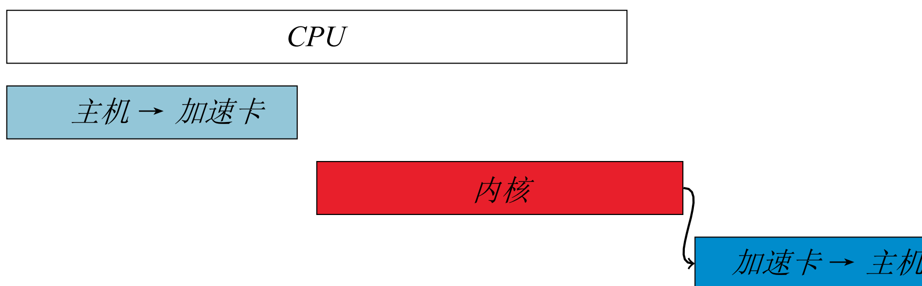

图 3.4：内核执行时间线

本图是说明性质的图示，但在相对尺度上。仔细观察数值，我们发现要超过 CPU，我们整个内核需要在约 120 ms 内完成运行。要实现这点，我们需要将运行速度提高 4 倍（可能性较低），每时钟周期处理 4 倍数据，或并行运行 4 个加速器。我们该选择哪种途径？我们留待下个示例进行讲解。

补充练习

本实验基础上的补充练习：

- 再次改动缓存容量。您能否找到 CPU 速度加快时的转折点？
- 尝试使用 SDAccel Timeline Trace 捕获运行时间变化并亲自查看。这项操作的指南详见 SDAccel 环境配置与优化指南 (UG1207)

要点总结

- 内核并行化能提供可观的结果，但应注意避免数据传输占用过多执行时间。如果不注意，就无法实现目标执行时间。
- 简单计算并非总是理想的加速对象，除非要释放处理器执行其他任务。更理想的加速对象是高度复杂的工作负载，特别是类似嵌套环路这样的 $O(N^x)$ 算法。
- 如果您必须等到全部数据传输完才能启动处理，即使内核都经过高度优化，也难以达成整体性能目标。

现在我们已经发现，只并行化数据路径还远远不够。下面让我们一起探讨如何进一步解决传输时间问题。

示例 5：优化计算与传输

简介

回顾上一个示例，一旦数据量超过一定水平，我们应用的数据传入传出就开始成为瓶颈。由于通过 PCIe 传输数据的用时是相对固定的，因此您可能倾向于创建四个 `wide_vadd` 内核的实例并将它们加入队列，以并行处理大型缓存。

这实际上是传统的 GPU 模式，通过 PCIe 批量发送大量数据到高带宽内存，然后在嵌入式处理器阵列上用超高带宽模式处理数据。

在使用 FPGA 时，处理这个问题的方法不同。如果内核能在每个时钟周期处理完整的 512 位数据，那么我们最大的问题不是并行化，而是如何提供足够的数据供处理。如果从/向 DDR 猝发数据，就能很容易完整利用这一带宽。在同一个缓存上并行放置多个核心并让它们连续工作，肯定会导致带宽竞争。这样做实际上适得其反，核心运行速度*降低*。

那么我们应该怎么做？记得上文提到过 `wide_vadd` 内核实际上并非理想的加速对象。计算过于简单，并不能充分发挥 FPGA 的并行优势。基本上，当算法简单到 A + B 时，速度提升空间并不大。但这个示例确实告诉我们一些优化方法，用来优化算法极为简单的应用。

为此目的，我们在 `wide_vadd` 内核的硬件设计上使用了一点技巧。我们不只是使用总线上的原始数据，而是使用 FPGA 块 RAM（基本上是速度极快的 SRAM）在内部稍微缓存数据，然后再进行计算。因此，我们能在前后相继的猝发间节约一点时间，并且充分利用。

记得前面我们已经将接口在多个 DDR 组间完成划分，如图 3.3 所示。因为 PCIe 的带宽远低于 Alveo 卡上任意给定 DDR 存储器的总带宽，因此我们将数据传输给两个不同的 DDR 组，在两者间交叉存取。因此由于和以前一样，我们能“挤进夹缝中”，我们可以*在数据到达时*就开始处理数据，不必遵循等待数据传输全部完成、处理数据、传输回数据的做法。这种做法的图示如图 3.5 所示。

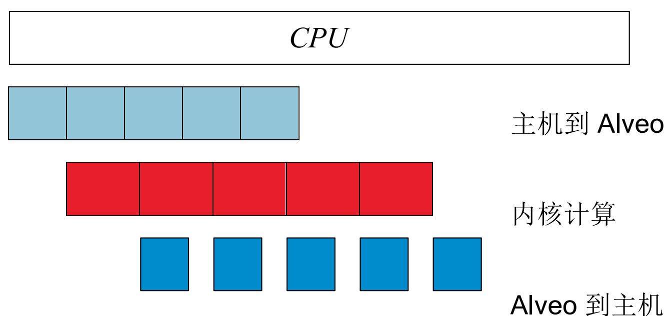

图 3.5：内核执行时间线（优化后）

通过这种方法细分缓存并选择理想的细分数，我们能够为应用均衡执行时间与传输时间，从而显著提高吞吐量。在硬件内核不变的情况下，

需要对主机代码进行如下设置。

关键代码

代码的算法流程基本相同。在为传输排队前，我们将遍历缓存并对它进行细分。但我们需要遵循一定的规则。首先，应在对齐边界上划分缓存，保持传输的高效率。其次，不应将缓存细分得过小，使之失去意义。我们通过定义常数 **NUM_BUFS** 来设置缓存数量，然后编写一个新函数来划分缓存，如列表 3.17 所示。

列表 3.17：细分 XRT 缓存

```
int subdivide_buffer(std::vector<cl::Buffer> &divided,
                     cl::Buffer buf_in,
                     cl_mem_flags flags,
                     int num_divisions)
{
    // Get the size of the buffer
    size_t size;
    size = buf_in.getInfo<CL_MEM_SIZE>();

    if (size / num_divisions <= 4096) {
        return -1;
    }

    cl_buffer_region region;

    int err;
    region.origin = 0;
    region.size   = size / num_divisions;

    // Round region size up to nearest 4k for efficient burst behavior
    if (region.size % 4096 != 0) {
        region.size += (4096 - (region.size % 4096));
    }

    for (int i = 0; i < num_divisions; i++) {
        if (i == num_divisions - 1) {
            if ((region.origin + region.size) > size)
                { region.size = size - region.origin;
            }
        }
        cl::Buffer buf = buf_in.createSubBuffer(flags,
                                                CL_BUFFER_CREATE_TYPE_REGION,
                                                &region,
                                                &err);

        if (err != CL_SUCCESS) {
            return err;
        }
        divided.push_back(buf);
        region.origin += region.size;
    }

    return 0;
}
```

这里的做法是在缓存内循环 NUM_BUFS 次，为我们想要创建的每个子缓存调用 cl::Buffer.createSubBuffer()。cl_buffer_region 结构体定义要创建的子缓存的起始地址和容量。值得注意到的是，子缓存可以重叠，虽然在我们的案例中不这么使用。

我们返回 cl::Buffer 对象的向量，用它为多次操作排队，如列表 3.18 所示。

列表 3.18：为细分的 XRT 缓存排队

```
int enqueue_subbuf_vadd(cl::CommandQueue &q,
                        cl::Kernel &krnl,
                        cl::Event &event,
                        cl::Buffer a,
                        cl::Buffer b,
                        cl::Buffer c)
{
    // Get the size of the buffer
    cl::Event k_event, m_event;
    std::vector<cl::Event> krnl_events;

    static std::vector<cl::Event> tx_events, rx_events;

    std::vector<cl::Memory> c_vec;
    size_t size;
    size = a.getInfo<CL_MEM_SIZE>();

    std::vector<cl::Memory> in_vec;
    in_vec.push_back(a);
    in_vec.push_back(b);
    q.enqueueMigrateMemObjects(in_vec, 0, &tx_events, &m_event);
    krnl_events.push_back(m_event);
    tx_events.push_back(m_event);
    if (tx_events.size() > 1)
        { tx_events[0] =
        tx_events[1];
        tx_events.pop_back();
    }

    krnl.setArg(0, a);
    krnl.setArg(1, b);
    krnl.setArg(2, c);
    krnl.setArg(3, (uint32_t)(size / sizeof(uint32_t)));

    q.enqueueTask(krnl, &krnl_events, &k_event);
    krnl_events.push_back(k_event);
    if (rx_events.size() == 1)
        { krnl_events.push_back(rx_events[0]);
        rx_events.pop_back();
    }
    c_vec.push_back(c);
    q.enqueueMigrateMemObjects(c_vec,
                               CL_MIGRATE_MEM_OBJECT_HOST,
                               &krnl_events,
                               &event);
    rx_events.push_back(event);

    return 0;
}
```

在列表 3.18 中，我们的事件排队方法与前面的做法基本相同：

1. 为从主机内存到 Alveo 存储器的缓存迁移排队。
2. 为当前缓存设置内核参数。
3. 为内核运行排队。
4. 为返回结果的传输排队。

不过，差别在于现在我们按实际排队的顺序执行操作。我们现在不用等到事件全部完成（就像之前的例子中那样），因为这样做会破坏流水线的意义。现在我们采用*基于事件的相依关系*。通过使用 `cl::Event` 对象，我们能够建立起事件链。事件链上的事件*必须*依次完成执行（未链接事件仍可以在任何时间调度）。

我们为内核的多次运行排队，然后等待它们全部完成运行。这样可以显著地改善调度效率。请注意：如果我们使用这个队列方法构建了与**示例 4** 相同的结构，我们的结果不会改变，因为运行时无从获知我们能否在所有数据发送之前安全地启动处理。身为设计人员，我们必须清楚调度器什么能做，什么不能做。

最后，如果我们漏掉一项重要操作，就不能按正确次序执行。我们必须规定，创建队列时我们可以通过传入旗标 CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE，使用*乱序命令队列*。

否则，本示例中的代码基本没有变化。我们现在调用这些函数，而不是直接从 `main()` 调用 `API`，除此之外其他不变。

但是在将缓存 *C* 映射回用户空间这个方面，*有*值得关注的地方。我们不必处理单个子缓存。因为它们已经迁移回主机内存，而且创建子缓存时底层指针没有改变，尽管有子缓存存在，我们仍然能够处理上级缓存（而且上级缓存甚至以某种方式处于休眠状态！）。

运行应用

在 XRT 初始化后，从构建目录运行下列命令以运行应用：

```
./05_pipelined_vadd alveo_examples
```

程序将输出类似于下列内容的消息：

```
-- Example 5:Pipelining Kernel Execution --

Loading XCLBin to program the Alveo board:

Found Platform
Platform Name:Xilinx
XCLBIN File Name: alveo_examples
INFO:
Importing ./alveo_examples.xclbin
Loading:' ./alveo_examples.xclbin'


 -- Running kernel test with XRT-allocated contiguous
    buffers and wide VADD (16 values/clock)

OCL-mapped contiguous buffer example complete!

--------------- Key execution times ---------------
OpenCL Initialization              :  263.001 ms
Allocate contiguous OpenCL buffers :  915.048 ms
Map buffers to userspace pointers  :    0.282 ms
Populating buffer inputs           : 1166.471 ms
Software VADD run                  : 1195.575 ms
Memory object migration enqueue    :    0.441 ms
Wait for kernels to complete       :  692.173 ms
```

表 3.6 是本次运行结果与上次运行结果的比较：

表 3.6：1 GiB 缓存用时总结 - 流水线与顺序对比

运行	示例 4	示例 5	$\Delta 4 \rightarrow 5$
软件 VADD	820.596 ms	1166.471 ms	345.875 ms
硬件 VADD（总计）	1184.897 ms	692.172 ms	−492.725 ms
$\Delta_{Alveo \rightarrow CPU}$	364.186 ms	−503.402 ms	867.588 ms

这次任务显然“大获成功”。看看这些裕量！

现在不可能反过来对我们不利，对吧？三十六计，走为上策，肯定不会拖泥带水。

补充练习

本实验基础上的补充练习：

- 再次改动缓存容量。练习中是否出现类似的转折点？

- 再次捕获运行时间变化，是否能看到差异？子缓存的数量选择如何影响运行时间（如果有影响）？

要点总结

- 智能地管理数据传输和命令队列可带来显著的速度提升。

示例 6：再遇波折

简介

您可能不会觉得这么快就可以结业了，对不对？正如我在本文开始时所提到的：*VADD 永远不可能超越处理器。* 它太简单了。如果不必传输数据，可以完全使用本地高速缓存，*最终*都是 CPU 胜出。就算您把您叔叔落满灰的 386 Turbo 从地下室里翻出来运行这个测试，也是一样。

上个示例中的结果看似不错，但只是纸上谈兵。我理解，我在搞营销。我想下班了。不过我好像想起我获得的是工程学位，虽然已经是很久之前的事。那么作为高级工程师，我给您讲解一下。

关键代码

对于简单算法，加速器不能胜出。让我们使用 OpenMP 并行化处理器循环。我们加入报头 `omp.h`，然后向 `CPU` 代码应用 `OpenMP` 编译指示，如列表 3.19 所示。

列表 3.19：使用 OpenMP 并行化软件

```
void vadd_sw(uint32_t *a, uint32_t *b, uint32_t *c, uint32_t size)
{
#pragma omp parallel for
    for (int i = 0; i < size; i++)
        { c[i] = a[i] + b[i];
    }
}
```

这就是代码。代码中有一些传递给 GCC 的命令行旗标，但 CMake 将管理它们（假定已安装 OpenMP），因此我们可以直接构建并运行。本示例中的代码与**示例 5** 中的代码基本完全相同。

运行应用

在 XRT 初始化后，从构建目录运行下列命令以运行应用：

```
./06_wide_processor alveo_examples
```

程序将输出类似下列内容的消息：

```
-- Example 6:VADD with OpenMP --

Loading XCLBin to program the Alveo

board:Found Platform
Platform Name:Xilinx
XCLBIN File Name: alveo_examples
INFO:
Importing ./alveo_examples.xclbin
Loading:' ./alveo_examples.xclbin'


 -- Running kernel test with XRT-allocated contiguous
    buffers and wide VADD (16 values/clock), with software
    OpenMP

OCL-mapped contiguous buffer example complete!

--------------- Key execution times --------------
- OpenCL Initialization            :  253.898 ms
Allocate contiguous OpenCL buffers :  907.183 ms
Map buffers to userspace pointers  :    0.307 ms
Populating buffer inputs           :1188.315 ms
Software VADD run                  :157.226   ms
Memory  object  migration  enqueue     :1.429  ms
Wait for kernels to complete        :618.231 ms
-- Example 5:Pipelining Kernel Execution --
```

表 3.7 是本次运行结果与上次运行结果的比较：

表 3.7：1 GiB 缓存用时总结 - 顺序与 OpenMP 对比

运行	示例 5	示例 6	$\Delta_{5\to6}$
软件 VADD	1166.471 ms	157.226 ms	−1009.245 ms
硬件 VADD（总计）	692.172 ms	618.231 ms	−73.94 ms
$\Delta_{Alveo\to CPU}$	−503.402 ms	461.005 ms	964.407 ms

加速器运行时的波动主要是因为在虚拟云环境中运行这些测试。这并非是这个练习的重点。

补充练习

本实验基础上的补充练习：

- 试图在向量加法上超越 CPU！
- 尝试 OpenMP 编译指示：要多少颗 CPU 核心才能超越硬件加速器？

要点总结

- 我之前曾提到，现在再重申：简单 $O(N)$ 无法胜出。

但是不要绝望！现在是动真格的时候了。

示例 7：使用 OpenCV 缩放图像

简介

图像处理是 FPGA 加速的一大应用领域，原因有几个。首先也是最重要的，如果您在进行任何类型的像素级处理，随着图像不断增大，计算量也随之增大。但是更重要的是，它非常符合我们前面提到的电车比喻。

先看一个非常简单的示例：双边缩放算法。该算法获取一个输入图像，并将其缩放到一个新的任意分辨率。缩放的步骤大致如下：

1. 从内存读取图像像素。
2. 必要时将其转换为合适格式。在本例中使用 OpenCV 库的默认格式 BGR。但是在真实系统中，会从多个源头的数据流（如摄像头等）接收数据，必须先进行格式处理。要么在软件中，要么在加速器中（加速器中基本是“无开销”操作，我们将在下个示例中讲解）。
3. 对于彩色图像，提取每个通道。
4. 在每个独立通道上使用双边缩放算法。
5. 重组通道，存储回内存。

如果想形象地思考，这个操作如图 3.6 所示：

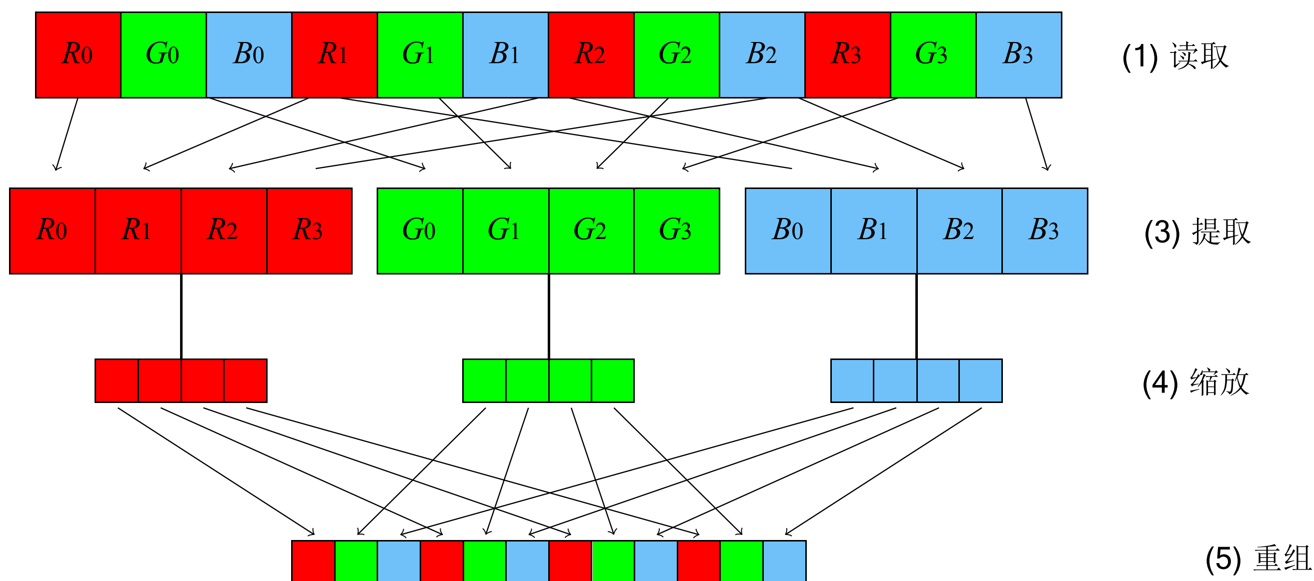

图 3.6：缩放算法数据流

为简化本图，未显示可选的色彩转换步骤。

值得注意的是，这次的操作*按顺序*依次完成。在之前的讨论中我们曾提到向内存往返传输数据会产生高成本。当我们从内存中输入数据时，如果我们能在 FPGA 中执行所有这些计算，则每个步骤或计算的

额外时延一般就会很小。我们可以利用这一点，构建定制的*域专用架构*，针对我们要实现的具体功能充分发挥硬件的作用。

然后您可以增大数据路径带宽，每时钟周期处理更多像素（**阿姆达尔定律**）。或使用多条流水线并行处理大量图像（**古斯塔夫森定律**）。理想情况下，我们可以同时从这两个方向进行优化。先构建能够尽可能高效处理您的算法的内核，然后在 FPGA 上实现您所需数量的内核。

关键代码

对这个算法，我们使用赛灵思 **xf::OpenCV 库。**这些硬件优化库实现了众多可以在应用中直接使用的通用 OpenCV 功能。此外，我们还可以根据需要，将它们与软件 OpenCV 功能或其他库调用混合匹配。

我们还能用这些库对其他内核进行图像预处理和后处理。例如，我们可能想从摄像头或网络流获得原始数据，预处理它，将结果馈入神经网络，然后使用结果进行进一步处理。所有这些操作都能在 FPGA 上完成，无需返回访问主机内存。而且所有这些操作都能实现并行化。构建流水线化功能流的唯一根本约束是带宽，不是寄存器空间。

在 xf::OpenCV 库中，您可以通过模板配置每时钟周期处理的像素数量等参数。此处不做详细讲解，请参阅赛灵思 OpenCV 用户指南 (UG1233) 进一步了解，或通过以下链接查看在线 OpenCV 转 xf::OpenCV 教程：

SDSoC 环境教程：将 OpenCV 迁移到 xfOpenCV

请注意，该教程专门针对赛灵思的嵌入式加速工具 SDSoC，但具体到使用 OpenCV 库编写内核时，其概念和方法也同样适用于您的 Alveo 电路板。

运行应用

在 XRT 初始化后，从构建目录运行下列命令以运行应用：

```
./07_opencv_resize alveo_examples <path_to_image>
```

因为本例使用的硬件配置方法，图像必须符合特定的要求。由于处理速度为每时钟 8 个像素，因此输入带宽必须为 8 的倍数。而且由于我们在 512 位总线上猝发数据，因此您的输入图像应满足如下要求：

$$in_width \times out_width \times 24 \% 512 = 0$$

如果不满足，程序将输出错误消息，告知您未满足什么条件。当然这并非是该库的基本要求；我们能够处理任意分辨率的图像，也能更改每时钟处理的像素数量。但为了实现理想性能，如何能确保输入图像满足特定要求，处理速度将会显著加快。

除了从软硬件 OpenCV 实现方案输出缩放后图像，该程序还将输出类似下列内容的消息：

```
-- Example 7: OpenCV Image Resize --

OpenCV conversion done!Image resized 1920x1080 to 640x360
Starting Xilinx OpenCL implementation...
Matrix has 3
channels Found
Platform Platform
Name:Xilinx
XCLBIN File Name: alveo_examples
INFO:
Importing ./alveo_examples.xclbin
Loading:’ ./alveo_examples.xclbin’
OpenCV resize operation          :    5.145 ms
OpenCL initialization            :  292.673 ms
OCL input buffer initialization  :    4.629 ms
OCL output buffer initialization :    0.171 ms
FPGA Kernel resize operation     :    4.951 ms
```

因为这次的做法有根本性改变，所以不与之前的运行结果进行对比。但*可以*对比在 CPU 上处理图像和在 Alveo 卡上处理图像的用时。

现在，我们注意到 Alveo 卡和 CPU 的使用效果不相上下。我们使用的双线性内插算法仍然是 $O(N)$，但这次它并非是简单的计算。因此，我们不像之前一样受 I/O 制约。我们可以超越 CPU，但程度有限。

这里值得注意的是，我们执行的是一系列基于输出分辨率而非输入分辨率的计算。我们需要传输相同数量的数据，但不是将图像缩小为输入大小的三分之一，而是倍增分辨率，看结果如何。更改本例的代码，如列表 3.20 所示，然后进行编译。

列表 3.20：倍增分辨率

```
uint32_t out_width = image.cols * 2;
uint32_t out_height = image.rows * 2;
```

再次运行示例。我们可以看到值得关注的地方。

```
-- Example 7:OpenCV Image Resize --

OpenCV conversion done!Image resized 1920x1080 to
3840x2160 Starting Xilinx OpenCL implementation...
Matrix has 3
channels Found
Platform Platform
Name:Xilinx
XCLBIN File Name: alveo_examples
INFO:
Importing ./alveo_examples.xclbin
Loading:' ./alveo_examples.xclbin'
OpenCV resize operation            :   11.692 ms
OpenCL initialization              :  256.933 ms
OCL input buffer initialization    :    3.536 ms
OCL output buffer initialization   :    7.911 ms
FPGA Kernel resize operation       :    6.844 ms
```

我们先把缓存初始化放在一边。一般在良好架构的系统中，不会在应用的关键路径上分配内存。我们现在只关心缩放操作和数据传输，速度比 CPU 快近 60%。

虽然 CPU 和 Alveo 卡处理该图像的速度都较快，试想实现每秒处理 60 帧的整体系统级吞吐量。在此情况下，每帧的处理时间仅有 16.66 ms。如果使用缩放后的帧作为神经网络的输入，您会发现您基本上已经耗尽可用的时间。

如果您想通过预分配缓存链来运行流水线，使用加速器可以让每帧处理速度再加快近 30%。

对于 1920x1200 的输入图像，我们得到的结果如表 3.8 所示。

表 3.8：图像缩放用时总结 - Alveo 与 OpenCV 对比

运行	缩小	放大
软件缩放	5.145 ms	11.692 ms
硬件缩放	4.951 ms	6.684 ms
$\Delta_{Alveo \rightarrow CPU}$	−194 µs	−5.008 ms

还有一个好消息，记得我前面提过的彩色转换步骤吗？在 FPGA 中它基本上是一个无开销操作。这会导致几个额外时钟周期的时延，但它基本上是另一种 $O(N)$ 运算。一种需对像素值只进行几次相乘相加的简单运算。

补充练习

本实验基础上的补充练习：

- 编辑主机代码，缩放图像。放大时运行时间如何变化？缩小时又如何变化？加速器不再起作用的转折点在什么地方？

- 向 FPGA 加速器添加色彩转换（请注意，将需要重构硬件）。查看是否需要更长的处理时间。

要点总结

- 计算复杂性更大的 $O(N)$ 运算是理想的加速目标，但与 CPU 相比优势不大。
- 使用 xf::OpenCV 等针对 FPGA 优化的库便于在处理速度与资源间权衡取舍，无需亲自重新实现通用算法。关注您应用中应该关注的地方！
- 部分可选的库优化功能可能制约您的设计。在硬件实现*前*，请反复核对您选择的库功能的说明书。

我们之前曾简要提到过，从用时角度来看，在 FPGA 架构中完成加法处理“开销很低”。在下一部分中我们通过实验来一探究竟。

示例 8：使用 OpenCV 流水线化操作

简介

在上个示例中我们了解了简单的双边缩放算法。虽然这个算法不是特别*理想的*加速对象，但或许您想同时将缓存内容转换为多个不同分辨率（如用于不同的机器学习算法）。或者您只想在帧窗口中卸载它，节省 CPU 资源，供其他处理使用。

但是让我们发挥 FPGA 的真正优势：流技术。记得曾介绍过，向内存往返传输数据开销很高。所以作为替代，我们只需要将图像的每个像素发送到另一个图像处理流水线阶段，而不需要通过简单地从一个操作流到下一个操作来返回内存。

在本例中，我们要修改之前的事件次序，以加入一个**高斯滤波器 (Gaussian Filter)**。这是一个处于边缘检测、角点检测等操作之前的极为常见的流水线级，用于消除图像噪声。我们还可以在它后面添加部分 2D 滤波，或是某些其他算法。

那么修改之前的工作流后，现在的工作流是：

1. 从内存读取图像像素。
2. 必要时将其转换为合适格式。在本例中使用 OpenCV 库的默认格式 BGR。但是在真实系统中，会从多个源头的数据流（如摄像头等）接收数据，必须先进行格式处理。要么在软件中，要么在加速器中（加速器中基本是“无开销”操作，我们将在下个示例中讲解）。
3. 对于彩色图像，提取每个通道。
4. 在每个独立通道上使用双边缩放算法。
5. 对每个通道执行高斯模糊。
6. 重组通道，存储回内存。

这样我们现在有两个“大”算法：双边缩放与高斯模糊。对于缩放后的图像 $wout \times hout$ 和宽度为 k 的正方形高斯窗口，我们整个流水线的计算时间大致为：

$$O(wout \cdot hout) + O(wout \cdot hout \cdot k2)$$

为取乐，我们可以让 k 取不超出界外的较大值。我们选择 7 × 7 窗口。

关键代码

对该算法我们继续使用赛灵思 **xf::OpenCV** 库。

和前面一样，我们通过配置库（在硬件中使用我们硬件源文件中的模板），每时钟周期处理 8 个像素。从功能上，我们的硬件算法等价于使用标准 OpenCV 表达的列表 3.21。

列表 3.21：示例 8：双边缩放与高斯模糊

```
cv::resize(image, resize_ocv, cv::Size(out_width, out_height), 0, 0, CV_INTER_LINEAR);
cv::GaussianBlur(resize_ocv, result_ocv, cv::Size(7, 7), 3.0f, 3.0f, cv::BORDER_CONSTANT);
```

我们按**示例 7** 中那样缩放图像，然后应用 7 × 7 的高斯模糊窗口。此外，我们也已经（任意地）选择 $\sigma_x = \sigma_y = 3.0$。

运行应用

在 XRT 初始化后，从构建目录运行下列命令以运行应用：

```
./08_opencv_resize alveo_examples <path_to_image>
```

和之前一样，因为本例使用的硬件配置方法，图像必须符合特定的要求。由于处理速度为每时钟 8 个像素，输入带宽必须为 8 的倍数。而且由于我们在 512 位总线上猝发数据，输入图像应满足如下要求：

$$in_width \times out_width \times 24 \% 512 = 0$$

如果不满足，程序将输出错误消息，告知您未满足什么条件。当然，这并非是该库的基本要求；我们能够处理任意分辨率的图像，也能更改每时钟处理的像素数量。但为了实现理想性能，如果能确保输入图像满足特定要求，那么处理速度会显著加快。

除了从软硬件 OpenCV 实现方案输出缩放后图像，该程序还将输出类似下列内容的消息：

```
-- Example 8:OpenCV Image Resize and Blur --

OpenCV conversion done!Image resized 1920x1080 to 640x360 and blurred
7x7!Starting Xilinx OpenCL implementation...
Matrix has 3
channels Found
Platform Platform
Name:Xilinx
XCLBIN File Name: alveo_examples
INFO:
Importing ./alveo_examples.xclbin
Loading:' ./alveo_examples.xclbin'
OpenCV resize operation            :    7.170 ms
OpenCL initialization              :  275.349 ms
OCL input buffer initialization    :    4.347 ms
OCL output buffer initialization   :    0.131 ms
FPGA Kernel resize operation       :    4.788 ms
```

在前面的示例中，CPU 和 FPGA 的使用效果不相上下。虽然我们为 CPU 功能增加了大量处理时间，但 FPGA 运行时间并未大幅增加。

现在让输入图像大小翻倍，从 1080p 图像变为 4k 图像。与**示例 7** 一样为本例更改代码，如列表 3.20 所示，然后进行编译。

再次运行本例，可以发现值得注意的地方。

```
-- Example 8:OpenCV Image Resize and Blur --

OOpenCV conversion done!Image resized 1920x1080 to 3840x2160 and blurred
7x7!Starting Xilinx OpenCL implementation...
Matrix has 3
channels Found
Platform Platform
Name:Xilinx
XCLBIN File Name: alveo_examples
INFO:
Importing ./alveo_examples.xclbin
Loading:' ./alveo_examples.xclbin'
OpenCV resize operation            :  102.977 ms
OpenCL initialization              :  250.000 ms
OCL input buffer initialization   :    3.473 ms
OCL output buffer initialization :    7.827 ms
FPGA Kernel resize operation      :    7.069 ms
```

这究竟是什么魔法？CPU 运行时增加了近 10 倍，FPGA 运行时基本没有变化！！！

FPGA 确实*非常擅长于*并行化处理。此算法不受 I/O 约束，受处理器约束。我们可以分解它，由每个时钟计算多个像素（**阿姆达尔定律**），或从一个操作流到下一个操作，并行执行更多操作（**古斯塔夫森定律**），加快数据处理速度。我们甚至还能将高斯模糊分解成单独的分量计算，然后并行运行（这个我们已经在 xf::OpenCV 库中做过）。

现在我们受*计算*而非*带宽*制约，我们能够清楚地看到加速的效果。如果用 FPS 来衡量，x86 级的 CPU 实例现在能每秒处理 9 帧，而 FPGA 卡能处理惊人的 141 帧。而且增加更多运算会不断拖累 CPU 性能。但只要 FPGA 的资源未被耗尽，就能有效地无止境地保持这一性能。事实上，与 Alveo U200 卡上的可用资源相比，我们内核的资源占用微乎其微。

为与上个示例进行比较，对于 1920x1200 输入图像，我们将获得的结果列在表中。
3.9. 比较栏将**示例 7** 的“放大”结果与本例的放大结果进行对比。

表 3.9：图像缩放与高斯模糊用时总结 - Alveo 与 OpenCV 对比

运行	缩小	放大	$\Delta_{7\rightarrow 8}$
软件缩放	7.170 ms	102.977 ms	91.285 ms
硬件缩放	4.788 ms	7.069 ms	385 µs
$\Delta_{Alveo\rightarrow CPU}$	−2.382 ms	−95.908 ms	−90.9 ms

我们希望您看到优势所在！

补充练习

本实验基础上的补充练习：

- 编辑主机代码，缩放图像尺寸。放大时运行时间如何变化？
  缩小时又如何变化？本例中使用加速器卡不再起作用的转折点在什么地方？
- 是否有额外的硬件时延？

要点总结

- 在架构中从一个操作到下一个操作的流水线和操作流是有益的。
- 使用 xf::OpenCV 等针对 FPGA 优化的库便于在处理速度与资源间权衡取舍，无需亲自重新实现通用算法。关注您应用中应该关注的地方！
- 部分可选的库优化功能可能制约您的设计。在硬件实现*前*，请反复核对您选择的库功能的说明书。

总结

我们希望本辅导教程能帮助您正确地发挥 Alveo 加速卡的性能。当然本教程不可能做到面面俱到。实际上我们只是触及了表面。如需了解有关如何为 Alveo 构建内核的更多信息，我们强烈建议继续学习下列主要讲解硬件内核的教程：

SDAccel 开发环境教程

SDAccel 环境优化指南

关于库：

赛灵思 OpenCV 用户指南 (UG1233)

SDSoC 环境教程：将 OpenCV 迁移到 xfOpenCV

当然，请记得使用赛灵思论坛！来自赛灵思团队和全球 SDAccel 爱好者的高技能、乐于助人的社区成员经常到访论坛。

关于 Alveo 卡的具体硬件问题：

Alveo 数据中心加速卡论坛

关于开发工具链：

SDAccel 论坛

此外，赛灵思也提供机器学习论坛。登录机器学习论坛，讨论我们的深度学习处理单元及其相关工具链“深度神经网络开发套件”：

DPU 与 DNNDK 论坛

感谢您花费宝贵时间学习这些教程。我们真诚地希望它们能有所帮助。我们期待与您合作！

附录 A

附加资源与法律提示

赛灵思资源

如需了解答复记录、技术文档、下载以及论坛等支持性资源，请参阅赛灵思技术支持 。

解决方案中心

访问赛灵思解决方案中心，在设计周期的各阶段获得器件、软件工具和知识产权支持。相关专题包括设计辅助、建议和故障排除提示等。

请阅读 ： 重要法律提示

本文向贵司/您所提供的信息（下称“资料”）仅在对赛灵思产品进行选择和使用时参考。在适用法律允许的最大范围内：（1）资料均按“现状”提供，且不保证不存在任何瑕疵，赛灵思在此声明对资料及其状况不作任何保证或担保，无论是明示、暗示还是法定的保证，包括但不限于对适销性、非侵权性或任何特定用途的适用性的保证；且（2）赛灵思对任何因资料发生的或与资料有关的（含对资料的使用）任何损失或赔偿（包括任何直接、间接、特殊、附带或连带损失或赔偿，如数据、利润、商誉的损失或任何因第三方行为造成的任何类型的损失或赔偿），均不承担责任，不论该等损失或者赔偿是何种类或性质，也不论是基于合同、侵权、过失或是其他责任认定原理，即便该损失或赔偿可以合理预见或赛灵思事前被告知有发生该损失或赔偿的可能。赛灵思无义务纠正资料中包含的任何错误，也无义务对资料或产品说明书发生的更新进行通知。未经赛灵思公司的事先书面许可，贵司/您不得复制、修改、分发或公开展示本资料。部分产品受赛灵思有限保证条款的约束，请参阅赛灵思销售条款：https://china.xilinx.com/legal.htm#tos；IP 核可能受赛灵思向贵司/您签发的许可证中所包含的保证与支持条款的约束。赛灵思产品并非为故障安全保护目的而设计，也不具备此故障安全保护功能，不能用于任何需要专门故障安全保护性能的用途。如果把赛灵思产品应用于此类特殊用途，贵司/您将自行承担风险和责任。请参阅赛灵思销售条款：http://china.xilinx.com/legal.htm#tos。

关于与汽车相关用途的免责声明

如将汽车产品（部件编号中含“XA”字样）用于部署安全气囊或用于影响车辆控制的应用（“安全应用”），除非有符合 ISO 26262 汽车安全标准的安全概念或冗余特性（“安全设计”），否则不在质保范围内。客户应在使用或分销任何包含产品的系统之前为了安全的目的全面地测试此类系统。在未采用安全设计的条件下将产品用于安全应用的所有风险，由客户自行承担，并且仅在适用的法律法规对产品责任另有规定的情况下，适用该等法律法规的规定。

© Copyright 2019 年赛灵思公司版权所有。Xilinx、赛灵思标识、Artix、ISE、Kintex、Spartan、Virtex、Vivado、Zynq 及本文提到的其它指定品牌均为赛灵思在美国及其它国家的商标。所有其它商标均为各自所有方所属财产。