Compute Express Link Overview

Ben Widawsky 系统范畴论 2022-08-13 09:00 Posted on 新加坡

2022-06-01

Intro

Compute Express Link (CXL) is the next spec of significance for **connecting hardware devices**. It will replace or supplement existing **stalwarts** like PCIe. The adoption is starting in the datacenter, and the specification definitely provides interesting possibilities for client and embedded devices. **A few years ago, the picture wasn't so clear**. The original release of the CXL specification wasn't Earth shattering.

There were competing standards with intent hardware vendors behind them. The drive to, and release of, the Compute Express Link **2.0 specification changed much of that**.

Compute Express Link（CXL）是连接硬件设备的下一个重要规格。它将取代或补充现有的中坚力量，如PCIe。数据中心正在开始采用，该规范无疑为客户端和嵌入式设备提供了有趣的可能性。几年前，情况还不是那么清楚。最初发布的CXL规范并不惊天动地。当时有一些相互竞争的标准，这些标准的背后都有专门的硬件供应商。Compute Express Link 2.0规范的推动和发布改变了这种状况。

There are a bunch of really great materials hosted by the CXL consortium. I find that these are primarily geared toward use cases, hardware vendors, and sales & marketing. This blog series will **dissect CXL with the scalpel of a software engineer**. I intend to provide an in **depth overview of the driver architecture**, and go over **how the development was done before there was hardware**. This post will go over the important parts of the specification as I see them.

CXL联盟提供了许多非常棒的材料。我发现这些资料主要是面向使用案例、硬件供应商以及销售和营销。这个博客系列将用软件工程师的手术刀来剖析CXL。我打算提供一个关于驱动架构的深入概述，并回顾在有硬件之前是如何进行开发的。这篇文章将介绍我所看到的规范中的重要部分。

All spec references are **relative to the 2.0 specification** which can be obtained here.

所有的规范参考都是相对于2.0规范的，可以在这里获得。

What it is

Let's start with two practical examples.

让我们从两个实际例子开始.

1. If one were creating **a PCIe based memory expansion device** today, and desired that device **expose coherent byte addressable memory** there would really be only two viable options.

   One could **expose this memory mapped input/output (MMIO) via Base Address Register (BAR)**. Without horrendous hacks, and to have ubiquitous CPU support, the only sane way this can work is if you **map the MMIO as uncached (UC)**, which has a distinct performance penalty. For more details on **coherent memory access to the GPU**, see my previous blog post. **Access to the device memory should be fast (at least, not limited by the protocol's restriction)**, and we haven't managed to accomplish that. In fact, **the NVMe 1.4 specification introduces the Persistent Memory Region (PMR) which sort of does this** but is still limited.

   如果今天有人要创建一个基于PCIe的内存扩展设备，并希望该设备能够暴露相干字节寻址的内存，那么实际上只有两个可行的选择。一个人可以通过基础地址寄存器（BAR）暴露这个内存映射的输入/输出（MMIO）。如果没有可怕的黑客技术，并且要有普遍的CPU支持，唯一合理的方法是将MMIO映射为未缓存（UC），这对性能有明显的影响。关于对GPU的一致性内存访问的更多细节，请看我之前的博文。对设备内存的访问应该是快速的（至少，不受协议的限制），而我们还没有设法完成这个目标。事实上，NVMe 1.4规范引入了持久性内存区域（PMR），它可以做到这一点，但仍然是有限的。

2. If one were creating a PCIe based device **whose main job was to do Network Address Translation (NAT)** (or some other IP packet mutation) that was to be done by the CPU, **critical memory bandwidth would be need for this**. This is because the CPU will have to **read data in from the device, modify it, and write it back out** and the only way to **do this via PCIe is to go through main memory**. (More on this below

   如果创建一个基于PCIe的设备，其主要工作是进行网络地址转换（NAT）（或其他一些IP数据包变异），这将由CPU完成，为此需要关键的内存带宽。这是因为CPU将不得不从设备中读取数据，对其进行修改，并将其写回，而通过PCIe的唯一方法就是通过主内存来完成。(下面有更多关于这方面的内容

CXL defines 3 protocols that work on top of PCIe that enable (Chapter 3 of the CXL 2.0 specification) **a general purpose way to implement the examples**. Two of these protocols help address our 'coherent but fast' problem above. I'll call them **the data protocols**. The third, **CXL.io can be thought of as a stricter set of requirements over PCIe config cycles**. I'll call that **the enumeration/configuration protocol**. We'll not discuss that in any depth as it's not particularly interesting.

CXL定义了3个协议，这些协议在PCIe之上工作，能够（CXL 2.0规范的第3章）以一种通用的方式来实现这些例子。这些协议中有两个有助于解决我们上面的 "相干但快速 "问题。我把它们称为数据协议。第三个，CXL.io可以被认为是对PCIe配置周期的一套更严格的要求。我把它称为枚举/配置协议。我们不会深入讨论这个问题，因为它不是特别有趣。

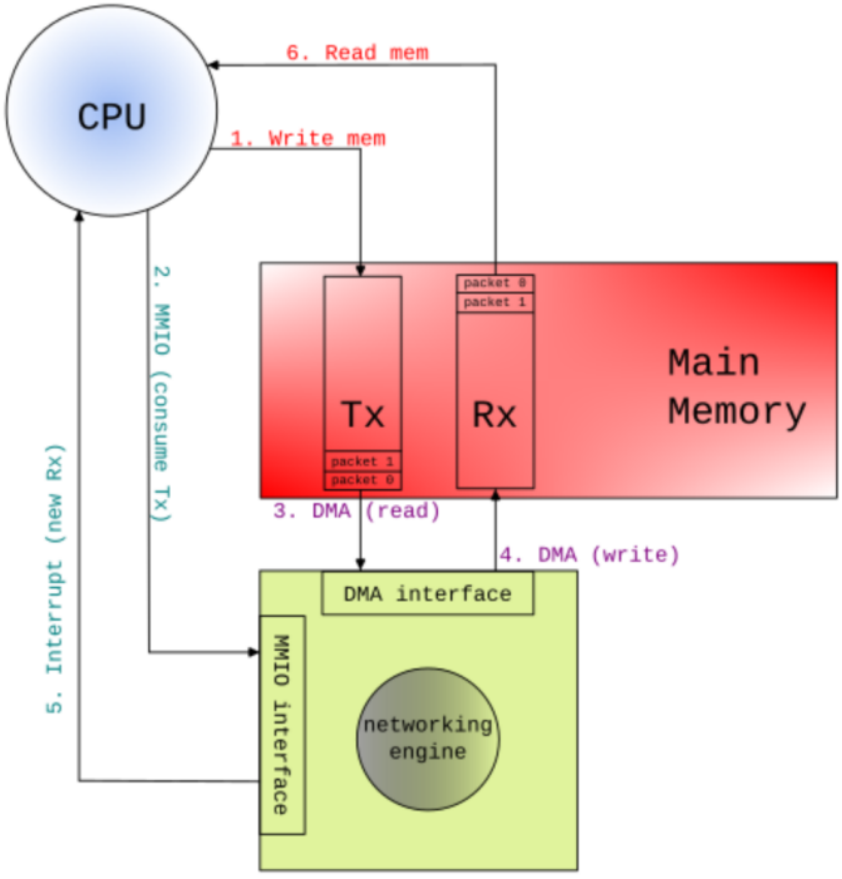
There's plenty of great overviews such as this one. The point of this blog is to **focus on the specific aspects driver** writers and reviewers might care about.

有很多很好的概述，比如这个。这篇博客的重点是专注于司机作者和评论员可能关心的具体方面。

CXL.cache

But first, a bit on **PCIe coherency**. Modern x86 architectures have **cache coherent PCIe DMA**. For DMA reads this simply means that **the DMA engine obtains the most recent copy of the data by requesting it from the fabric**. For writes, **once the DMA is complete the DMA engine will send an invalidation request to the host(s) to invalidate the range that was DMA'd**. Fundamentally however, using this is generally not optimal since **keeping coherency would require the CPU basically snoop the PCIe interconnect all the time**. This would be **bad for power and performance**. As such, general **drivers manage coherency via software mechanisms**. There are exceptions to this rule, but I'm not intimately familiar with them, so I won't add more detail.

但首先，要介绍一下PCIe一致性。现代x86架构有缓存一致性的PCIe DMA。对于DMA读取，这仅仅意味着DMA引擎通过从结构中请求获得数据的最新副本。对于写入，一旦DMA完成，DMA引擎将向主机发送一个无效请求，以使被DMA的范围无效。然而，从根本上说，使用这种方法通常不是最佳选择，因为保持一致性需要CPU一直窥探PCIe互连的情况。这对功耗和性能都是不利的。因此，一般驱动程序通过软件机制管理一致性。这条规则也有例外，但我对它们并不熟悉，所以我不会添加更多细节。



Driving a PCIe NIC

CXL.cache is interesting because it allows the device to participate in the CPU cache coherency protocol as if it were another CPU rather than being a device.

From a software perspective, it's the less interesting of the two data protocols. Chapter 3.2.x has a lot of words around what this protocol is for, and how it is designed to work.

This protocol is targeted towards accelerators which do not have anything to provide to the system in terms of resources, but instead *utilize host-attached memory and a local cache*.

The CXL.cache protocol if successfully negotiated throughout the CXL topology, host to endpoint, should just work. **It permits the device to have coherent view of memory without software intervention and on x86, without the potential negative ramifications of the snoops**.

Similarly, the host is able to **read from the device caches without using main memory as a stopping point**.

Main memory can be skipped and instead data can be directly transferred over CXL.cache protocol.

The protocol describes snoop filtering and the necessary messages to keep coherency.

As a software person, I consider it **a more efficient version of PCIe coherency, and one which transcends x86 specificity.**

CXL.cache很有意思，因为它允许设备参与CPU缓存一致性协议，就好像它是另一个CPU而不是一个设备。从软件的角度来看，它是两个数据协议中不太有趣的一个。第3.2.x章围绕这个协议的用途，以及它的设计工作方式有很多话要说。这个协议针对的是那些在资源方面没有向系统提供任何东西的加速器，而是利用主机连接的内存和本地缓存。CXL.cache协议如果在整个CXL拓扑结构中成功协商，从主机到终端，应该就能发挥作用。它允许设备在没有软件干预的情况下对内存有一致的看法，在x86上，没有潜在的窥探的负面影响。同样地，主机能够从设备缓存中读取，而不使用主内存作为停止点。主内存可以被跳过，而数据可以直接通过CXL.cache协议传输。该协议描述了窥探过滤和必要的信息以保持一致性。作为一个软件人员，我认为这是PCIe一致性的一个更有效的版本，而且超越了x86的特殊性。

Protocol

CXL.cache has **a bidirectional request/response protocol** where a request can be made from **host to device (H2D) or vice-versa (D2H)**. The set of commands are what you'd expect to **provide proper snooping**. For example, H2D requests with one of the 3 Snp* opcodes defined in 3.2.4.3.X, these allow to **gain exclusive access to a line, shared access, or just get the current value**; while **the device uses one of several commands in Table 18 to read/write/invalidate/flush (similar uses).**

CXL.cache有一个双向的请求/响应协议，一个请求可以从主机到设备（H2D）或反过来（D2H）。这套命令是你所期望的，以提供适当的窥探。例如，H2D用3.2.4.3.X中定义的3个Snp*操作码中的一个请求，这些允许获得对一条线的独占访问，共享访问，或只是获得当前值；而设备使用表18中的几个命令之一来读/写/无效/冲洗（类似用途）。

One might also notice that the peer to peer case isn't covered. **The CXL model however makes every device/CPU a peer in the CXL.cache domain**. While the current CXL specification **doesn't address skipping CPU caches in this matter entirely**, it'd be a safe bet to assume **a specification so comprehensive would be getting there soon**. **CXL would allow this more generically than NVMe.**

人们可能还注意到，点对点的情况没有被涵盖。然而，CXL模型使每个设备/CPU都是CXL.cache域中的对等体。虽然目前的CXL规范并没有完全解决跳过CPU缓存的问题，但可以肯定的是，如此全面的规范很快就会出现。CXL将比NVMe更普遍地允许这样做。

To summarize, **CXL.cache essentially lets CPU and device caches to remain coherent without needing to use main memory as the synchronization barrier.**

总而言之，CXL.cache本质上让CPU和设备缓存保持一致，而不需要使用主内存作为同步屏障。

CXL.mem

If CXL.cache is for devices that don't provide resources, CXL.mem is exactly the opposite. **CXL.mem allows the CPU to have coherent byte addressable access to device-attached memory while maintaining its own internal cache**. Unlike CXL.cache where every entity is a peer, the device or host sends requests and responses, **the CPU, known as the "master" in the CXL spec, is responsible for sending requests, and the CXL subordinate (device) sends the response**. Introduced in **CXL 1.1,**

CXL.mem was added for Type 2 devices. Requests from the master to the subordinate are "M2S" and responses are "S2M".

如果说CXL.cache是针对不提供资源的设备，那么CXL.mem则恰恰相反。CXL.m允许CPU对设备连接的内存进行连贯的字节寻址访问，同时保持其自身的内部缓存。与CXL.cache不同的是，每个实体都是对等体，设备或主机发送请求和响应，CPU（在CXL规范中被称为 "主"）负责发送请求，而CXL下属（设备）则发送响应。在CXL 1.1中引入的CXL.mem是为2类设备添加的。从主站到下级的请求是 "M2S"，响应是 "S2M"。

When CXL.cache isn't also present, **CXL.mem is very straightforward**. All requests boil down to a read, or a write. When CXL.cache is present, the situation is more tricky. For performance improvements, **the host will tell the subordinate about certain ranges of memory which may not need to handle coherency between the device cache, device-attached memory, and the host cache.** There is also meta data passed along to inform the device about the current cacheline state. **Both the master and subordinate need to keep their cache state in harmony.**

当CXL.cache不存在时，CXL.mem是非常直接的。所有请求都归结为读或写。当CXL.cache存在时，情况就比较棘手了。为了提高性能，主机会告诉下级某些内存的范围，这些内存可能不需要处理设备缓存、设备附加内存和主机缓存之间的一致性。还有一些元数据的传递，以告知设备当前的缓存线状态。主控方和从属方都需要保持其缓存状态的和谐。

Protocol

CXL.mem protocol is straight forward, especially when the device doesn't also use CXL.cache (ie. it has no local cache). CXL.mem协议是直接的，特别是当设备不使用CXL.cache的时候（即它没有本地缓存）。

The requests are known as

这些请求被称为

1. Req - Request without data. These are generally **reads and invalidates**, where **the response will put the data on the S2M's data channel.**

   Req - 没有数据的请求。这些通常是读和无效的，响应会把数据放在S2M的数据通道上。

2. RwD - Request with data. These are **generally writes**, where **the data channel has the data to write.**

   RwD - 带有数据的请求。这些通常是写的，数据通道有数据要写。

The responses:

1. NDR - No Data Response. These are generally **completions on state changes** on the device side, **such as, writeback complete.**

   NDR - 无数据响应。这些通常是设备端状态变化的完成，例如，回写完成。

2. DRS - Data Response. This is used for returning data, ie. on a read request from the master.

   DRS - 数据响应。这是用于返回数据，即在主站的读请求上。

Bias controls

Unsurprisingly, **strict coherency often negatively impacts bandwidth, or latency, or both.** While it's generally ideal from a software model to be coherent, it likely won't be ideal for performance. The CXL specification has a solution for this. Chapter 2.2.1 describes a knob which allows a mechanism to **provide the hint over which entity should pay for that coherency (CPU vs. device)**.

For many HPC workloads, such as weather modeling, large sets of **data are uploaded to an accelerator's device-attached memory via the CPU, then the accelerator crunches numbers on the data, and finally the CPU download results**. In CXL, at all times, the model data is coherent for both the device and the CPU. **Depending on the bias however, one or the other will take a performance hit.**

不足为奇的是，严格的一致性常常对带宽或延迟产生负面影响，或者两者都有。虽然从软件模型上看，连贯性通常是理想的，但对性能来说可能并不理想。CXL规范对此有一个解决方案。第2.2.1章描述了一个旋钮，它允许一个机制对哪个实体应该为该一致性付费（CPU与设备）提供提示。对于许多HPC工作负载，如天气建模，大量的数据集通过CPU上传到加速器的设备连接内存，然后加速器对数据进行计算，最后由CPU下载结果。在CXL中，在任何时候，模型数据对设备和CPU都是一致的。但是，根据不同的偏向，其中一个会受到性能的影响。

Host vs. device bias

Using the weather modeling example, there are 4 interesting flows.

使用天气建模的例子，有4个有趣的流。

1. CPU writes data to device-attached memory. CPU将数据写到设备连接的内存中。
2. GPU **reads data from** device-attached memory. GPU从设备连接的内存中读取数据。
3. ***GPU writes data to device_attached memory**. GPU将数据写入设备连接的内存。
4. CPU reads data from device attached memory. CPU从设备附加内存中读取数据。

*#3 above poses an interesting situation that was possible only with bespoke hardware. The GPU could in theory **write that data out via CXL.cache and short-circuit another bias change**. In practice though, many such usages would blow out the cache.

#3上面提出了一个有趣的情况，只有在定制硬件的情况下才有可能。理论上，GPU可以通过CXL.cache把数据写出来，并短路另一个偏置变化。但在实践中，许多这样的使用会耗尽高速缓存。

The CPU coherency engine has been a thing for a long time. One might ask, why not just use that and be done with it. Well, easy one first, **a Device Coherency Engine (DCOH) was already required for**

CXL.cache protocol support.

More practically however, **the hit to latency and bandwidth is significant if every cacheline access required a check-in with the CPU's coherency engine**.

What this means is that **when the device wishes to access data from this line, it must first determine the cacheline state (DCOH can track this), if that line isn't exclusive to the accelerator, the accelerator needs to use CXL.cache protocol to request the CPU make things coherent, and once complete, then can access it's device-attached memory.**

Why is that? If you recall, **CXL.cache is essentially where the device is the initiator of the request, and CXL.mem is where the CPU is the initiator.**

CPU的一致性引擎已经存在了很长时间了。有人可能会问，为什么不直接使用这个引擎，然后就可以了。好吧，首先很简单，设备一致性引擎（DCOH）对于CXL.cache协议的支持已经是必需的。然而，更实际的是，如果每个缓存线的访问都需要在CPU的一致性引擎上进行检查，那么对延迟和带宽的冲击是很大的。这意味着，当设备想要访问这一行的数据时，它必须首先确定缓存线的状态（DCOH可以跟踪这一点），如果这一行不是加速器独有的，加速器需要使用CXL.cache协议来请求CPU使其协调一致，一旦完成，就可以访问它的设备连接的内存。这是为什么呢？如果你记得，CXL.cache本质上是设备作为请求的发起者，而CXL.mem是CPU作为发起者。

So suppose we continue on this CPU owns coherency adventure. #1 looks great, the CPU can quickly upload the dataset. **However, #2 will immediately hit the bottleneck just mentioned. Similarly for #3, even though a flush won't have to occur, the accelerator will still need to send a request to the CPU to make sure the line gets invalidated.** To sum up, we have coherency, but half of our operations are slower than they need to be.

所以，假设我们继续进行这个CPU拥有一致性的冒险。#1号看起来不错，CPU可以快速上传数据集。然而，#2会立即遇到刚才提到的瓶颈。同样，对于#3，即使不需要进行刷新，加速器仍然需要向CPU发送请求，以确保该行被废止。总而言之，我们有连贯性，但有一半的操作比它们需要的要慢。

To address this, a [fairly vague] description of bias controls is defined. **When in host bias mode, the CPU coherency engine effectively owns the cacheline state (the contents are shared of course) by requiring the device to use CXL.cache for coherency. In device bias mode however, the host will use CXL.mem commands to ensure coherency. This is why Type 2 devices need both CXL.cache and CXL.mem.**

为了解决这个问题，定义了一个[相当模糊的]偏置控制的描述。当处于主机偏置模式时，CPU一致性引擎通过要求设备使用CXL.cache进行一致性，有效地拥有了缓存线状态（当然内容是共享的）。然而在设备偏向模式下，主机将使用CXL.mem命令来确保一致性。这就是为什么2类设备同时需要CXL.cache和CXL.mem。

Device types

I'd like to know why they didn't start numbering at 0. I've already talked quite a bit about device types. I believe it made sense to define the protocols first though so that device types would make more sense. **CXL 1.1 introduced two device types, and CXL 2.0 added a third. All types implement CXL.io**, the less than exciting protocol we ignore.

我想知道为什么他们不从0开始编号，我已经谈了不少关于设备类型的内容。我相信先定义协议是有意义的，这样设备类型才会更有意义。CXL 1.1引入了两种设备类型，而CXL 2.0增加了第三种。所有类型都实现了CXL.io，也就是我们所忽视的不那么激动人心的协议。

Type	CXL.cache	CXL.mem
1	y	n
2	y	y
3	n	y

Just from looking at the table it'd be wise to ask, if Type 2 does both protocols, **why do Type 1 and Type 3 devices exist**. In short, gate savings can be had with Type 1 devices not needing CXL.mem, and Type 3 devices offer gate savings and increased performance because they don't have to manage internal cache coherency. More on this next...

仅仅从表上看，我们就会问，如果类型2同时执行两种协议，为什么还有类型1和类型3的设备存在。简而言之，第一类设备不需要CXL.mem就可以节省门数，第三类设备可以节省门数并提高性能，因为它们不需要管理内部缓存一致性。接下来会有更多的内容...

CXL Type 1 Devices

These are your **accelerators without local memory.**

这些是没有本地内存的加速器。

The quintessential Type 1 device is the NIC. A NIC **pushes data from memory out onto the wire**, or **pulls from the wire and into memory**. It might perform many steps, such as repackaging a packet, or encryption, or reordering packets (I dunno, not a networking person). Our NAT example above is one such case.

最典型的1类设备是NIC。网卡将数据从内存推到线上，或者从线上拉到内存中。它可能会执行许多步骤，如重新包装数据包，或加密，或重新排序数据包（我不知道，不是网络人士）。我们上面的NAT例子就是这样一种情况。

How you might envision that working is the PCIe device would write the incoming packet into the Rx buffer. The CPU would copy that packet out of the Rx buffer, update the IP and port, then write it into the Tx buffer. This set of steps would use memory write bandwidth when the device wrote into the Rx buffer, memory read bandwidth when the CPU copied the packet, and memory write bandwidth when the CPU writes into the Tx buffer. Again, **NVMe has a concept to support a subset of this case for peer to peer DMA called Controller Memory Buffers (CMB), but this is limited to NVMe based devices, and doesn't help with coherency on the CPU**. Summarizing, (D is device cache, M is memory, H is Host/CPU cache)

你可能设想的工作方式是PCIe设备将传入的数据包写进Rx缓冲区。CPU将从Rx缓冲区复制该数据包，更新IP和端口，然后将其写入Tx缓冲区。当设备写入Rx缓冲区时，这组步骤将使用内存写带宽，当CPU复制数据包时，使用内存读带宽，当CPU写入Tx缓冲区时，使用内存写带宽。同样，NVMe有一个概念，支持这种情况的一个子集，用于对等DMA，称为控制器内存缓冲区（CMB），但这只限于基于NVMe的设备，并且对CPU的一致性没有帮助。总结一下，（D是设备缓存，M是内存，H是主机/CPU缓存）

1. (D2M) Device writes into Rx Queue (D2M) 设备写进Rx队列
2. (M2H) Host copies out buffer (M2H) 主机复制出缓冲区
3. (H2M) Host writes into Tx Queue (H2M) 主机写进Tx队列
4. (M2D) Device reads from Tx Queue (M2D) 设备从Tx队列读出

Post-CXL this becomes a matter of managing cache ownership throughout the pipeline. The NIC would write the incoming packet into the Rx buffer. **The CPU would likely copy it out so as to prevent blocking future packets from coming in**. Once done, the CPU has the buffer in its cache. **The packet information could be mutated all in the cache, and then delivered to the Tx queue for sending out**. Since the NIC may decide to mutate the packet further before going out, **it'd issue the RdOwn opcode (3.2.4.1.7), from which point it would effectively own that cacheline.**

在CXL之后，这就变成了在整个管道中管理缓冲区所有权的问题。NIC将把传入的数据包写进Rx缓冲区。CPU可能会将其复制出来，以防止阻挡未来的数据包进入。一旦完成，CPU的缓存中就有了这个缓冲区。数据包的信息可以在缓冲区内全部变异，然后送到Tx队列中发送出去。由于NIC可能决定在发送前进一步修改数据包，它将发出RdOwn操作码(3.2.4.1.7)，从那时起它将有效地拥有该缓存线。

1. (D2M) **Device writes into Rx Queue** (D2M) 设备写进Rx队列
2. (M2H) **Host copies out buffer** (M2H) 主机复制出缓冲区
3. (H2D) **Host transfers ownership into Tx Queue** (H2D) 主机将所有权转移到Tx队列中

With accelerators that don't have the possibility of causing backpressure like the Rx queue does, step 2 could be removed.

如果加速器没有像Rx队列那样造成背压的可能性，那么第2步就可以去掉。

CXL Type 2 Devices

These are your **accelerators with local memory**.

这些是你的具有本地内存的加速器。

Type 2 devices are mandated to support both data protocols and as such, **must implement their own DCOH engine** (this will vary in complexity based on the underlying device's cache hierarchy complexity). One can think of this problem the same way as multiple **CPUs where each has their own L1/L2, but a shared L3 (like Intel CPUs have, where L3 is LLC)**. Each CPU would need to track transitions between the local L1 and L2, and **the L3 to the global L3**. TL;DR on this is, for Type 2 devices, there's **a relatively complex flow to manage local cache state on the device in relation to the host-attached memory they are using.**

第二类设备被要求支持这两种数据协议，因此，必须实现它们自己的DCOH引擎（这将根据底层设备的缓存层次结构的复杂性而有所不同）。我们可以把这个问题看作是多个CPU的问题，每个CPU都有自己的L1/L2，但有一个共享的L3（像英特尔CPU那样，L3是LLC）。每个CPU都需要跟踪本地L1和L2之间的转换，以及L3到全局L3的转换。简而言之，对于第二类设备，有一个相对复杂的流程来管理设备上的本地缓存状态，与它们使用的主机连接的内存有关。

In a pre-CXL world, if a device wants to access its own memory, caches or no, it would have the logic to do so. For example, in GPUs, the sampler generally has a cache. If you try to access texture data via the sampler that is already in the sampler cache, everything remains internal to the device. **Similarly, if the CPU wishes to modify the texture, an explicit command to invalidate the GPUs sampler cache must be issued before it can be reliably used by the GPU (or flushed if your GPU was modifying the texture).**

在CXL之前的世界里，如果一个设备想访问它自己的内存，不管是否有缓存，它都有这样的逻辑。例如，在GPU中，采样器通常有一个缓存。如果你试图通过取样器访问已经在取样器缓存中的纹理数据，那么一切都会保持在设备内部。同样，如果CPU希望修改纹理，在GPU能够可靠地使用它之前，必须发出明确的命令使GPU的采样器缓存失效（如果你的GPU正在修改纹理，则需要刷新）。

Continuing with this example in the post-CXL world, the texture lives in graphics memory on the card, and **that graphics memory is participating in CXL.mem protocol.** That would imply that should the CPU want to inspect, or worse, modify the texture it can do so in a coherent fashion. **Later in Type 3 devices, we'll figure out how none of this needs to be that complex for memory expanders.**

在后CXL时代继续这个例子，纹理存在于显卡的显存中，而显存是参与CXL.mem协议的。这意味着，如果CPU想检查，或者更糟糕的是，修改纹理，它可以以一种连贯的方式进行。在第三类设备的后面，我们将弄清楚对于内存扩展器来说，这些都不需要那么复杂。

CXL Type 3 Devices

These are your **memory modules**. They provide memory capacity **that's persistent, volatile, or a combination.**

这些是你的内存模块。它们提供持久性、易失性或组合性的内存容量。

Even though a Type 2 device could technically behave as a memory expander, it's not ideal to do so. The nature of a Type 2 device is that it has a cache which also needs to be maintained. Even with meticulous use of bias controls, extra invalidations and flushes will need to occur, and of course, extra gates are needed to handle this logic. The host CPU does not know a Type 2 device has no cache. **To address this, the CXL 2.0 specification introduces a new type, Type 3, which is a "dumb" memory expander device.** Since this device has no visible caches (because there is no accelerator), a reduced set of CXL.mem protocol can be used, the CPU will never need to snoop the device, which means the CPU's cache is the cache of truth. **What this also implies is a CXL type 3 device simply provides device-attached memory to the system for any use**. **Hotplug is permitted.**

Type 3 peer to peer is absent from the 2.0 spec, and unlike CXL.cache, it's not as clear to see the path forward because CXL.mem is a Master/Subordinate protocol.

即使第二类设备在技术上可以作为一个内存扩展器，但这样做并不理想。第二类设备的性质是它有一个缓存，而这个缓存也需要被维护。即使仔细地使用偏置控制，也需要进行额外的失效和刷新，当然也需要额外的门来处理这个逻辑。主机CPU不知道2类器件没有缓存。为了解决这个问题，CXL 2.0规范引入了一种新的类型，即类型3，它是一种 "哑巴 "内存扩展器设备。由于这种设备没有可见的缓存（因为没有加速器），所以可以使用一套减少的CXL.mem协议，CPU永远不需要窥探设备，这意味着CPU的缓存是真理的缓存。这也意味着CXL 3型设备只是向系统提供设备连接的内存，供任何使用。热插拔是允许的。2.0规范中没有第3类对等设备，而且与CXL.cache不同的是，由于CXL.mem是一个主/从协议，它的发展路径并不清晰。

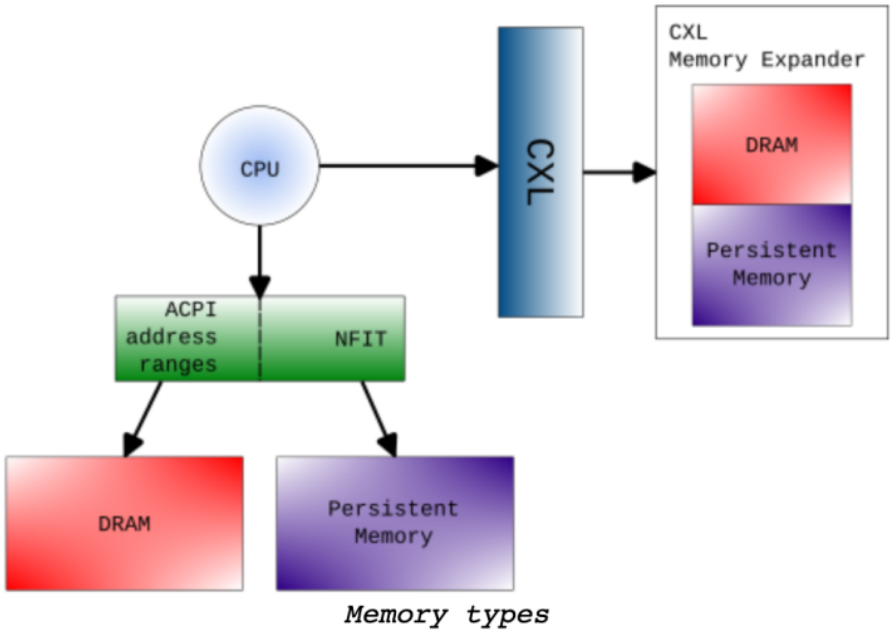
In a pre-CXL world the closest thing you find to this are **a combination of PCIe based NVMe devices (for persistent capacity), NVDIMM devices, and of course, attached DRAM**. Generally, DRAM isn't available as expansion cards because a single DDR4 DIMM (which is internally dual channel), only has 21.6 GB/s of bandwidth. P**CIe can keep up with that, but it requires all 16 lanes**, which I guess isn't scalable, or cost effective, or something. **But mostly, it's not a good use of DRAM when the platform based interleaving can yield bandwidth in the hundreds of gigabytes per second.**

在CXL之前的世界里，你能找到的最接近的东西是基于PCIe的NVMe设备（用于持久性容量）、NVDIMM设备，当然还有附加的DRAM。一般来说，DRAM不能作为扩展卡使用，因为单个DDR4 DIMM（内部是双通道），只有21.6GB/s的带宽。PCIe可以跟上这个速度，但它需要所有的16条通道，我想这是不可扩展的，或者是不符合成本效益的，或者是其他什么。但最主要的是，当基于平台的交织可以产生每秒数百GB的带宽时，这并不是对DRAM的良好使用。

Type	Max bandwidth (GB/s)
PCIe 4.0 x16	32
PCIe 5.0 x16	64
PCIe 6.0 x16	256
DDR4 (1 DIMM)	25.6
DDR5 (1 DIMM)	51.2
HBM3	819

In a post-CXL world the story is changed in the sense that the **OS is responsible for much of the configuration and this is why Type 3 devices are the most interesting from a software perspective.** Even though CXL currently runs on PCIe 5.0, **CXL offers the ability to interleave across multiple devices thus increasing the bandwidth in multiples by count of the interleave ways**. **When you take PCIe 6.0 bandwidth, and interleaving, CXL offers quite a robust alternative to HBM, and can even scale to GPU level memory bandwidth with DDR.**

在CXL之后的世界里，故事发生了变化，操作系统负责大部分的配置，这就是为什么从软件的角度来看，3类设备是最有趣的。尽管CXL目前在PCIe 5.0上运行，但CXL提供了在多个设备间交错的能力，从而以交错方式的倍数增加带宽。当你使用PCIe 6.0带宽和交错时，CXL提供了相当强大的HBM替代品，甚至可以扩展到GPU级别的内存带宽与DDR。



Host physical address space management

This would apply to Type 3 devices, but technically could also apply to Type 2 devices.

这将适用于第三类设备，但在技术上也可以适用于第二类设备。

Even though the protocols and use-cases should be understood, t**he devil is in the details with software enabling**. Type 1 and Type 2 devices will largely gain benefit just from hardware; perhaps some flows might need driver changes, ie. **reducing flushes and/or copies which wouldn't be needed**. **Type 3 devices on the other hand are a whole new ball of wax.**

尽管协议和使用情况应该被理解，但魔鬼在软件启用的细节中。第1类和第2类设备将在很大程度上从硬件上获得好处；也许有些流量可能需要改变驱动，即减少冲刷和/或复制，这是不需要的。

另一方面，第三类设备是一个全新的球。

Type 3 devices will need host physical address space allocated dynamically (it's not entirely unlike memory hot plug, but it is trickier in some ways).

The devices will need to be programmed to accept those addresses.

And last but not least, **those devices will need to be maintained using a spec defined mailbox interface.**

第三类设备将需要动态分配主机物理地址空间（这与内存热插拔并不完全不同，但在某些方面更棘手）。设备将需要被编程以接受这些地址。最后但并非最不重要的是，这些设备将需要使用一个规范定义的邮箱接口来维护。

The next chapter will start in **the same way the driver did**, the mailbox interface used for device information and configuration.

下一章将以驱动程序的方式开始，即用于设备信息和配置的邮箱接口。

Summary

Important takeaways are as follow:

重要的启示如下:

1. CXL.**cache allows CPUs and devices to operate on host caches uniformly.**

   CXL.cache允许CPU和设备在主机缓存上统一操作。

2. CXL.**mem allows devices to export their memory coherently.**

   CXL.mem允许设备连贯地输出其内存。

3. Bias controls mitigate performance penalties of CXL.mem coherency

偏置控制减轻了CXL.mem一致性的性能损失。

4. Type 3 devices provide a subset of CXL.mem, for memory expanders.

   3类设备提供CXL.mem的一个子集，用于内存扩展器。

Related:

- https://www.computeexpresslink.org/download-the-specification
- https://www.computeexpresslink.org/resource-library
- https://developers.redhat.com/blog/2016/03/01/reducing-memory-access-times-with-caches#
- https://en.wikipedia.org/wiki/Bus_snooping#Snoop_filter