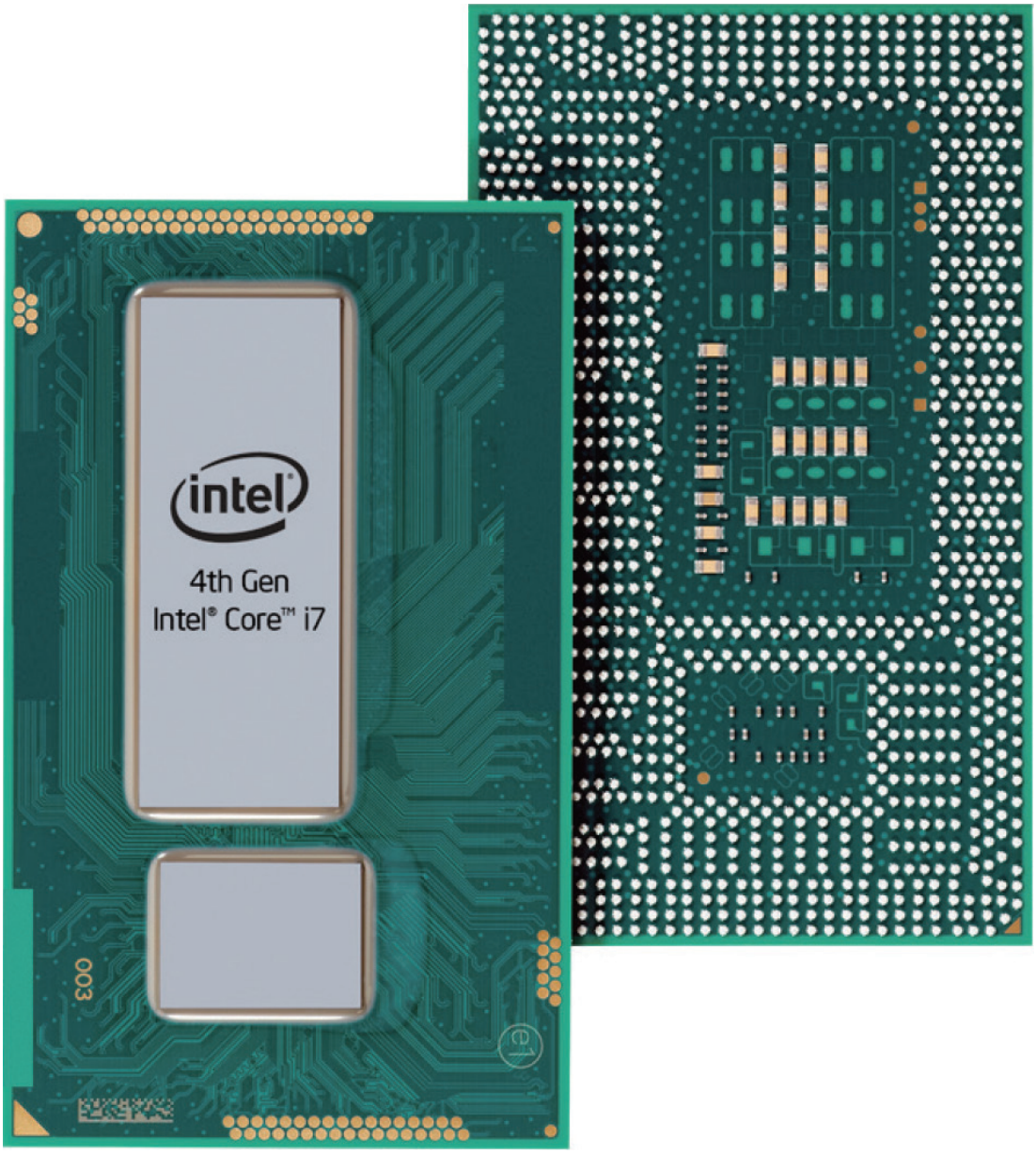
CPU、GPU 间的数据交互

Bruce Bao
Intel Corporation
September 2014



1 摘要

现在越来越多的视频监控产品开始采用 GPU 加速的视频解决方案，特别像基于 Intel 平台的产品，单芯片实现 24 路、32 路高清视频解码的方案也已经比较普遍。同时随着技术不断地更新换代，智能处理器的性能也日益增强，视频监控产品自然而然的被赋予了除视频编解码之外更多的如视频分析、存储转发、多任务管理等工作内容。如何充分利用系统中不同处理器各自的优势发挥系统整体的最大潜力日益受到主流视频监控厂商的关注，而这种需求在类似于 Intel CPU/GPU 同芯片的异构系统中表现得更为直接。

就像连接两座城市之间的桥梁，在 CPU、GPU 协同工作的场景中如何进行有效的数据交互，从而发挥出两种不同架构不同特性的处理器的最大性能是系统整体构建的关键。只有高效的数据传输才能最大化系统的异构优势并实现产品性能最大化；反之，如果不能正确处理数据的交换和访问，不但损失了异构系统的优异特性，甚至会使两个处理器都处于工作闲置状态，甚至导致“1+1<1”的反效果。

本文主要针对当前视频监控产品设计中最常见的 CPU/GPU 内存数据访问问题进行讨论，先介绍一下 CPU、GPU 之间数据互访的相关背景知识，然后结合视频监控的最常见的一种应用场景 — “CPU 访问 GPU 解码后的画面”介绍几种基本的实现方法以及在 Intel 平台上几种对应的解决方案。

1.1 术语/缩略语

表 1. 术语/缩略语

CPU	中央处理器 / Central Processing Unit
GPU	图形处理器 / Graphics Processing Unit
DSP	数字信号处理器 / Digital Signal Processor
SVM	虚拟内存共享 / Shared Virtual Memory
SPM	物理内存共享 / Shared Physical Memory
OpenCL	开放计算语言 / Open Computing Language
Linear 格式	线性数据格式，CPU 的数据存储格式，GPU 内通常用于一维数据的存储
Tile 格式	GPU 在显存内对二维或三维图形图像的特殊数据排列格式，通常以块（Block）为单位
Aperture	GPU 内将 Tile 格式转化成线性格式输出的硬件单元
VPP	视频像素处理 / Video Pixel Process

2 背景知识

CPU、GPU 都是智能处理器，因此有各自的地址空间和寻址方式，这一特性决定了即使两个处理器共享同一组物理上的内存，实际运行过程中还是有着各自独立的内存空间划分。显而易见，任何处理器处理自己内存区域的数据是最高效的，访问其它处理器内存空间内的数据则需要做地址空间/排列格式等转换，或者通过拷贝的方式直接传输数据。

在寻址方式上，CPU 配合现代的操作系统如 Windows 和 Linux 可以实现 32 位或 64 位保护模式下的虚拟寻址，大的内存数据在实际分配中通常是由非常多物理地址并不连续的页表组成。内存地址空间被划分成用户模式和核心模式，核心模式下的地址空间由系统统一管理，用户模式下则给予每个不同的应用各自独立的的地址空间，从而避免了应用程序间的相互影响提高了整个系统的鲁棒性。与之相反，当前主流的 GPU 并不支持虚拟寻址的方式，和 CPU 运行在 DOS 下的实模式类似，GPU 访问显存时是按照地址连续访问的，所有的 GPU 工作任务都限定在同一个地址空间内。

除了各自独立的地址空间，CPU 和 GPU 在数据的内存排部方式上也截然不同。CPU 是直观的按照线性方式排列数据，数据内容按照逻辑上的顺序在内存空间内由低位地址向高位地址排列。GPU 则会根据数据类型按照线性或 Tile 格式排列数据，按线性方式排列的比如 Vertex Buffer/Index Buffer 等一维的数据对象。而二维或三维的图形图像数据则主要是用 Tile 方式排列，如 Surface/Texture 等等。Tile 格式可以简单直观的理解成数据（像瓦片一样）按块状排列，虽然地址仍从低向高分布，但相邻的物理字节之间已经不再一一对应图片上的逻辑前后关系了。GPU 的这种数据排列方式和 GPU 的工作特性是直接相关的 — 以图形图像为例，GPU 需要对大量的像素类数据执行统一的操作，各个像素执行的操作基本一致并且各像素点相互间的逻辑关联性不强，此时对数据进行 Tile 方式的排列读写效率要大大高于线性方式，这也是 GPU 在很多数据并行的计算场景中要优于 CPU 的主要原因之一。当然，架构设计上的差异并不意味着 GPU 在所有计算中都有优势，比如散发性突发性数据计算，逻辑类计算，或者数据间有强依赖关系等场景中 CPU 的设计架构仍有先天优势，相关内容请参考其它技术文档，本文不做进一步讨论。

另一方面，因为操作系统本身仍是运行在 CPU 上的，因此用户的应用在访问 CPU 管理的内容时比较直接，对系统内存的分配和释放以及读出写入都由 CPU 实时完成。而 GPU 的地址空间管理目前主要还是依靠 GPU 的显示驱动完成，用户应用必须透过图形驱动实现对显存内容的间接访问，因此 GPU 对显存的实际管理对用户而言相对不透明。

最后，因为两个处理器有自己的工作上下文环境和内存读写逻辑，所以在数据互访时必须要进行同步操作以保证数据的完整和一致性。具体来说，显存数据的读写都是由 GPU 执行并完成的，但 GPU 的执行指令则需要显示驱动根据应用程序的 CPU 指令来动态解析编译生成。因此应用程序中对一段显存的渲染指令从 CPU 运行时调用到最终 GPU 执行完之间有一个时间差，当两个处理器都需要访问同一共用区域时一定要保证相互间的同步与互斥。例如 CPU 需要访问显存内容时 GPU 必须先执行完 CPU 之前下达的和该段显存相关的 GPU 渲染命令，这是同步操作；而之后的 CPU 访问期间 GPU 不能同时对该显存段进行读写操作，这是互斥机制。同步和互斥逻辑需要在应用程序在设计功能时就考虑，通过正确的函数调用关系，操作系统和显示驱动会帮助应用实现具体的功能。

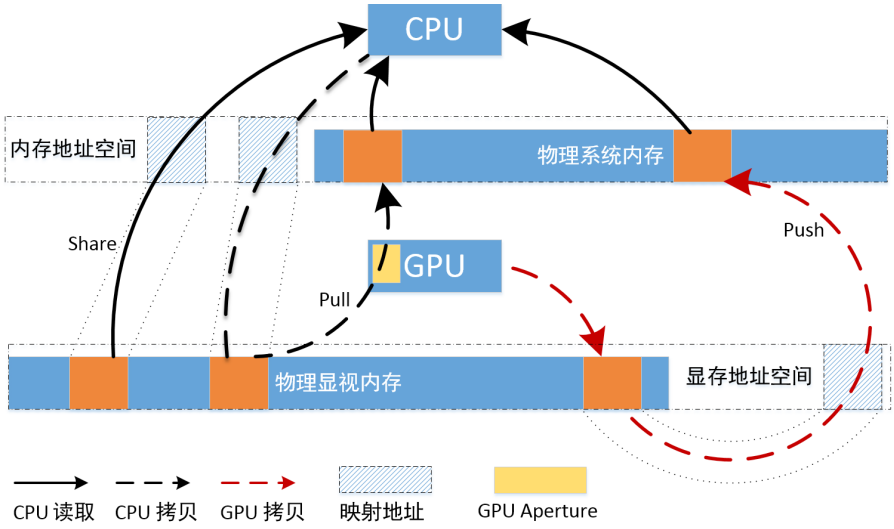
3 基本方法

了解了 CPU、GPU 两种处理器的不同之处，我们再来看看它们之间进行数据交互的几种基本方法。以 CPU 访问 GPU 输出的显存内容为例，其实两个不同的智能处理器之间数据交互的基本方法无外乎下图所示的三种形式：共享访问（share）、CPU 拷贝数据（pull）、GPU 推送数据（push）。

3.1 内存共享访问（share）：CPU 将 GPU 显存地址映射（Map）到 CPU 地址空间，并通过映射后的地址空间直接读取显存数据。

这种方式是最直接的，在应用程序调用地址映射操作的接口函数时显示驱动会首先确保 GPU 在该段内存的渲染指令全部完成；然后显示驱动会根据对应显存的数据类型和排列方式判断是否需要分配一种特殊的硬件资源（Aperture）来执行 Tile 到线性格式的转换，该操作主要是针对图像数据；接着显示驱动将该段地址映射到 CPU 的地址空间中，并在对应的图像结构数据中将其标记为 Locked（锁定）状态，防止 GPU 在 CPU 访问期间对该地址的读写操作；最后显示驱动将系统内存的首地址指针和其它相关信息如图像宽、高、图像间隙（Pitch）等作为返回值返回给上层应用，自此 CPU 才能根据所返回的信息访问显存里的数据内容。

图 1. CPU、GPU 数据交互基本途径



CPU 访问结束后，应用程序调用解除地址映射的接口函数，显示驱动依次释放系统内存指针和 Aperture 等硬件资源，并将图像结构数据内的标记恢复为 UnLocked（未锁定）状态从而恢复 GPU 对该段显存内容的读写操作。

很多如 Intel 产品的芯片中 Lock/Unlock 操作可能还会涉及 CPU/GPU 各自数据缓存的清空操作，在此不再深入讨论。

3.2 CPU 拷贝数据（Pull）：CPU 将数据从显存拷贝到系统内存。

这种方式实际上是基于内存共享访问的，但是因为共享访问时显存内容需要经过地址映射，格式转化，地址转

换（核心态到用户态）几种转换后才能让 CPU 访问到，所以直接访问的效率相比于普通系统内存要低。在很多场景中当 CPU 需要频繁且不连续的读取显存中的数据时，一次性将访问效率较低的图像数据拷贝到系统内存中进行访问反而能提高整体性能。

这种做法的另一个好处是，CPU 在映射完成后先一次性将显存内容拷贝到系统内存，减少了 GPU 和 CPU 之间的同步等待时间，让 GPU 可以很快重复使用该段显存。缺点是基于 CPU 的内存拷贝通常会成为整个软件运行的性能瓶颈，再强制两个处理器相互等待的同时也占用了大量的 CPU 的计算资源用于数据拷贝，从而压缩了 CPU 真正用于数据处理的空间。

3.3 GPU 拷贝数据（Push）：利用 GPU 将显存数据推送到系统内存。

这种方法利用了 GPU 的并行计算能力来搬运数据，不但可以大大提高数据拷贝的效率缩短拷贝时间，还降低了两个处理器之间的耦合度，同时释放了宝贵的 CPU 资源，让 CPU 可以从事更重要的数据处理工作。虽然有各种好处，但这种方法缺少标准的应用函数接口，具体的实现需要 GPU 设计厂商来完成，因此不同厂商甚至相同厂商的不同平台以及不同的操作系统都可能有差异。

4 实现方案

理解了三种不同方法的原理和优缺点，现在我们来看看在 Intel 平台上这三种方法的对应实现情况。

4.1 内存共享

内存共享让两个不同的处理器访问同一段内存区域。在实现上也有两种方法：SPM（Shared Physical Memory）和在现代的异构体系中经常被提及的 SVM（Shared Virtual Memory）。

SPM 是传统的物理内存共享，通常由显示驱动动态完成，因此 CPU 在应用程序运行的不同时间对同一段 GPU 显存请求共享后得到的虚拟地址可能完全不同，反之亦然。例如，当 CPU 希望访问一段 GPU 输出的显存数据时，显示驱动会将物理显存映射到 CPU 的地址空间，并给出映射后的起始地址和图像 Pitch 等信息。而当 GPU 希望读取一段系统内存的内容时，显示驱动会将系统内存的页表描述符填入 GPU 的 GART 表（Graphics Address Remapping Table），让 GPU 可以访问系统内存。

SVM 在最新的 OpenCL2.0 规范中有明确的定义，包含 coarse-grained sharing 和 fine-grained sharing 两种不同的共享方式。和 SPM 不同，SVM 不但实现了内存物理上的共享，在虚拟内存指针上也实现了一定程度的共享，CPU 和 GPU 可以使用相同的虚拟地址访问同一块内存空间，具体 SVM 的定义用户可以参考 OpenCL2.0 的规范。

目前主流的操作系统还没有提供对 SVM 的原生支持，所以用户如果要使用 SVM 技术只能依赖于 OpenCL2.0 的软硬件环境，OpenCL2.0 的规范是在 2013 年底完成的，作为行业先驱者之一，英特尔率先在 2014 年的第五代酷睿产品中对 OpenCL2.0 提供了完整的硬件（包括 CPU 和 GPU）支持。

实际上无论是 SPM 还是 SVM 都是基于最基本的内存共享方法，如前文所述这种数据交互方式是最直接的，优点例如：逻辑简单，内存占用量小，同时因为没有多余的内存拷贝操作

CPU 和 GPU 的执行指令更少。但是内存映射方式也有其弊端：

- 内存映射将强制在指定内存上的 CPU/GPU 同步，如果 GPU 都在对该段内存进行读写操作，这种同步将极大的影响系统性能。

- 对图像数据的转换效率低。之前说过 GPU 在进行图像渲染时会选用 Tile 格式排列数据，而 CPU 则只能访问线性的内存数据，因此内存映射的同时要求进行数据格式的重排，让 CPU 能正确读取重排后的线性数据。Tile/Linear 间的转换虽然使用的是 GPU 内部的一个硬件资源来完成的，但是其转换效率仍然远低于处理器直接访问内存的效率。

- SVM 只适用于线性数据（Buffer）。因为历史原因，Tile 格式并没有行业标准，不同厂商的 GPU 甚至不同产品系列的 GPU 中使用的 Tile 格式都各有差异。因此 Tile 格式的不统一也成为了 OpenCL2.0 规范中 SVM 不能支持 Image 类型数据共享的一个主要原因，这对以视频图像为主的视频监控用户无疑是美中不足的。

基于上述几个原因，共享数据交互方式比较适合低频率的数据交互。同时对图像数据来说最好还是连续的数据读写，反之则效率很低。因为频繁的进行地址映射（开启共享）无疑会增加 CPU/GPU 相互同步等待的次数，而非连续的读写则会让硬件的数据格式转换效率更低。

不论 SPM 还是 SVM，CPU 对线性数据的共享方式访问执行效率仍是非常高的，因此对于图像数据一方面我们可以考虑将其连续读取（或写入）到系统内存，另一方面可以考虑在 CPU 通过共享方式访问前将其转换成线性数据。下文中我们将做进一步探讨。

4.2 CPU 拷贝数据

及 CPU 从显存拷贝数据到系统内存，这种方式在视频监控领域是最常见的。本质上执行 CPU 拷贝前已经先完成了内存共享，并在内存共享的基础上执行拷贝的操作。但通过一次性将数据从显存拷贝到系统内存的操作，提高了 GPU 数据排列转换的执行效率，缩短了 CPU/GPU 同步的时间，又因为 CPU 之后的数据处理全部在系统内存中进行，从而也极大的提高了 CPU 访问的执行效率。在诸如需要对图像等大块数据进行多次交互处理的场景中，用 CPU 先将数据拷贝到系统内存中再处理的方式会比直接通过内存共享访问效率高很多。

C 语言中的 memcpy 是最常见的 CPU 拷贝方式，用户通常也用该函数将显存内容读取到系统内存中。但是 memcpy 在实际执行中通常效率很低，原因是 memcpy 是按字节逐一完成拷贝过程的，在 Intel 平台上不能充分发挥出 CPU 的硬件性能优势。因此在 Intel 平台上我们推荐使用 Fastcopy 的方式来拷贝比较大的数据（如图像数据），充分利用 CPU 本身的 SSE/AVX 等并行计算指令来快速高效的完成拷贝过程。以从第二代酷睿产品开始支持的 AVX 指令集为例，256 位的 AVX 指令可以并行执行 8 组 32 位的双精度浮点或长整形运算，对于内存拷贝而言一组 AVX 指令理论上可以同时传输 32 个字节，这将大大提高 CPU 在

copy 时的执行效率并缩短拷贝时间。有关 Fastcopy 的具体实现过程 Intel 提供了相关的白皮书进行了详细的描述，不在本文的讨论范围。需要注意的是使用 SSE/AVX 指令集某种程度上也是对 CPU 内部特定的硬件单元编程，因此具有硬件编程一个常见的特点：内存地址有对齐要求。对小内存数据（例如小于一个内存页表的数组型数据等）而言，因为数据量本身比较小使用 Fastcopy 得到的性能提升有限，另一方面小数据的起始地址也不一定能满足 Fastcopy 硬件对齐的条件，因此仍可以使用传统的 memcpy 方式。

相对于直接的内存共享访问，CPU 拷贝的优点是提高了 CPU 对显存内容访问的效率，缩短了 CPU/GPU 的交互时间，特别是在数据量较大的数据传输场景中对系统的性能提升更显著，另外如果客户在 Intel 平台选择 Fastcopy 的方式无疑将带来更显著的系统性能提升，在以视频内容为主的视频监控应用中应该优先考虑。但无论是 Fastcopy 还是传统的 memcpy，基于 CPU 的内存拷贝方式都会占用宝贵的 CPU 资源，而过高的 CPU 占用率不但会限制系统内能同时执行的任务数量，还会导致系统对鼠标键盘等突发性请求响应变慢等问题，所以仍有一定的局限性。

4.3 GPU 拷贝数据

前文介绍过 GPU 作为一个独立的智能处理器非常适合做数据并行方面的计算。所谓数据并行，即利用数量众多的可执行单元对大量输入数据进行统一的相同或相似的算术运算后并行输出。而内存拷贝完全可以当做一种典型的数据并行操作：将连续的大量的数据从一个存储位置转移到另一个存储位置。所以理论上使用者完全可以用 GPU 来实现内存拷贝功能，而且如果能使用 GPU 完成内存拷贝不但可以利用 GPU 的并行处理能力大大缩短拷贝的执行时间，另一方面还能释放宝贵的 CPU 资源让系统处理更多的任务以及提高系统的响应能力，而这两方面在视频监控的多种应用场景中都非常需要。

虽然有各种优点，但目前 GPU 拷贝的功能并没有普及。最主要的原因是当前主流的图形标准（如 Microsoft® DirectX、OpenGL 等）中的接口函数定义中没有使用 GPU 做内存拷贝的功能定义。其次 GPU 对内存读写有更苛刻的硬件对齐要求而且不能直接以纯线性的方式读写二维图像数据，这就要求底层驱动在对 GPU 编程拷贝时把“Tile 格式数据的输入”通过数据格式的重排列转换成“纯线性数据的输出”。因此并不是每家 GPU 公司都有相应的解决方案。作为视频监控市场智能芯片的主流提供商之一，Intel 充分考虑了市场的实际需求并提供了通过 Intel® Media SDK 隐式实现基于 GPU 的内存拷贝方案。

Intel® Media SDK 是 Intel 从 2010 年开放的一套视频开发软件，主要帮助用户来实现基于 Intel GPU 硬件加速的视频数据的解码、像素处理（VPP：Video Pixel Process）、编码等功能，同时提供了在 Intel 产品上的向后兼容性，简化了接口调用函数数量，降低了应用程序的开发难度。在视频监控领域，非常多的客户已经在使用 Intel® Media SDK 并且开发出了很多性能优异的产品（关于 Media SDK 产品的详细介绍，读者可以访问 Intel 的 Media SDK 产品主页获取更多内容）。

在以编解码为主的视频应用开发中，像素处理 VPP 模块是一个非常关键的环节。用户既可以通过 VPP 完成诸如色彩格式转换、输入输出帧率转换、

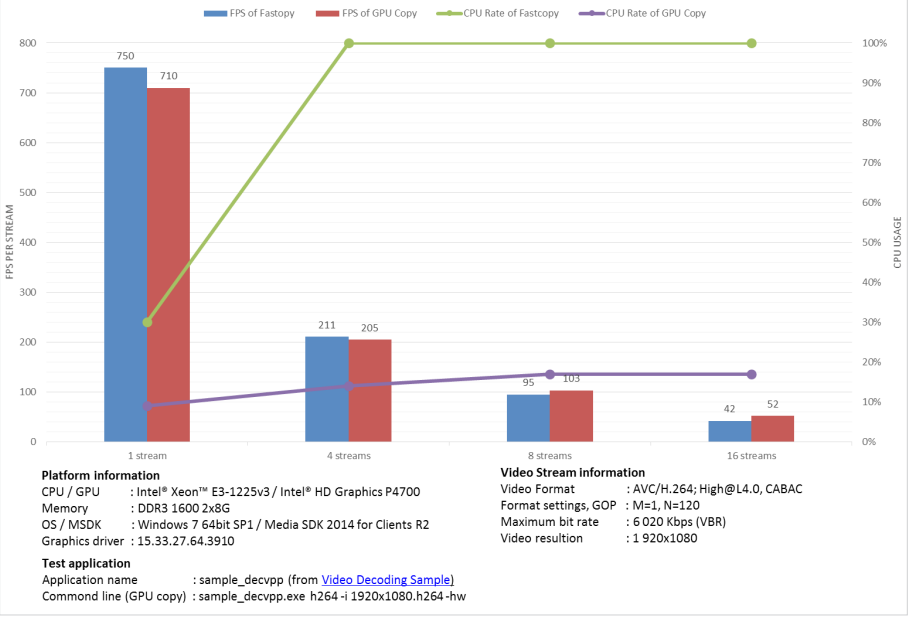
隔行逐行转换等等对视频画面的处理，也可以通过 VPP 实现显存到内存或者内存到显存的转换，后一种转换主要是为了帮助用户实现某些转码应用场景中软解硬编、硬解软编的特殊需求而产生的。因此 Intel® Media SDK 的 VPP 模块式允许输入输出指定不同类型的内存，而这恰恰是一个隐式的内存拷贝过程 — Intel® Media SDK 内部会实现数据从显存道内存的拷贝，反之亦然。

因为这个隐式的拷贝是在 Intel® Media SDK 内部执行的，所以 Intel 对 VPP 的输入输出转换做了充分的优化。早期的 Intel® Media SDK 实现的是基于 CPU 的 Fastcopy，这种软件拷贝的方式可以很好的满足不同年代不同规格 Intel 产品的内存拷贝需求，同时兼顾新产品的新指令集的优势。而今年（2014 年）配合新版本的显示驱动程序 Intel® Media SDK 终于实现了基于 GPU EU 单元的内存拷贝，使得用户在视频解码后有更多的 CPU 资源来处理基于高清视频的视频分析等工作。

图 2 是基于 Intel® Xeon™ E3-1225v3 处理器的 GPU 拷贝与 CPU Fastcopy 的测试比较，由图可见，在不同路数下测试 1080P 硬件高清解码并拷贝解码图像到内存的 CPU 占用率以及每路平均帧率。其中 E3-1225v3 的 CPU 主频基频 3.2GHz 最高 3.6GHz，GPU 主频基频 350MHz 最高 1.2GHz。由图可见，在使用 GPU 拷贝后 CPU 的占用率大幅度降低，16 路全速解码加数据拷贝的 CPU 占用率仍不到 20%，而使用 Fastcopy 的方式在 4 路时 CPU 已经处于 100% 的饱和状态，系统甚至无法及时响应键盘鼠标事件。进一步的分析我们还可以得到以下结论：

- 1.2GHz 的 HD Graphics P4700 GPU 内存拷贝的效率完全不弱于 3.6GHz 四核的 Xeon™ CPU 处理器，而且当并行解码路数提高后 CPU 因为需要增加多进程/线程的调度工作，拷贝性能会呈下降趋势，GPU 的数据拷贝的优势体现得更加明显。
- GPU 拷贝不会降低硬件解码性能。Intel GPU 内部的 MFX 解码单元和 EU 可编程执行单元是相互独立的，因此 GPU 进行多路拷贝时不会影响其解码性能，这一点可以从各项测试中不论是 GPU 或是 CPU 拷贝但解码出的总帧数基本相同可以看出。这也从另一个侧面说明了 Intel GPU 内部异构设计的优势。当然，解码加拷贝的帧率还

图 2.　CPU、GPU 拷贝性能比较



来源：英特尔公司 IOTG 事业部

是会低于纯解码的帧率，因为每帧视频解码后都增加了拷贝的时间，单位时间能处理的总帧数一定会降低。

- 单路执行时的总帧率略低于多路执行时的帧率。这是因为测试中单路执行流水线的异步深度为 1，GPU 在串行执行每一帧的解码加拷贝任务，流水线的各个单元有一定的串行等待。多路并发时各个单元可以在等待当前任务时去执行另一路视频的任务，因此多路并发时 GPU 的执行效率更高，总帧率也有所提升。

目前 Intel 首先在 Windows 平台的显示驱动中实现了基于 GPU 的隐式内存拷贝的功能，用户可以在 Intel 下载中心（downloadcenter.intel.com）搜索最新的图形驱动程序，驱动程序版本号大于等于 3740 的驱动均能满足要求。Linux 相关驱动正在进一步优化中，有望配合新版本的 Media SDK for Server 在 2014 年底前发布。

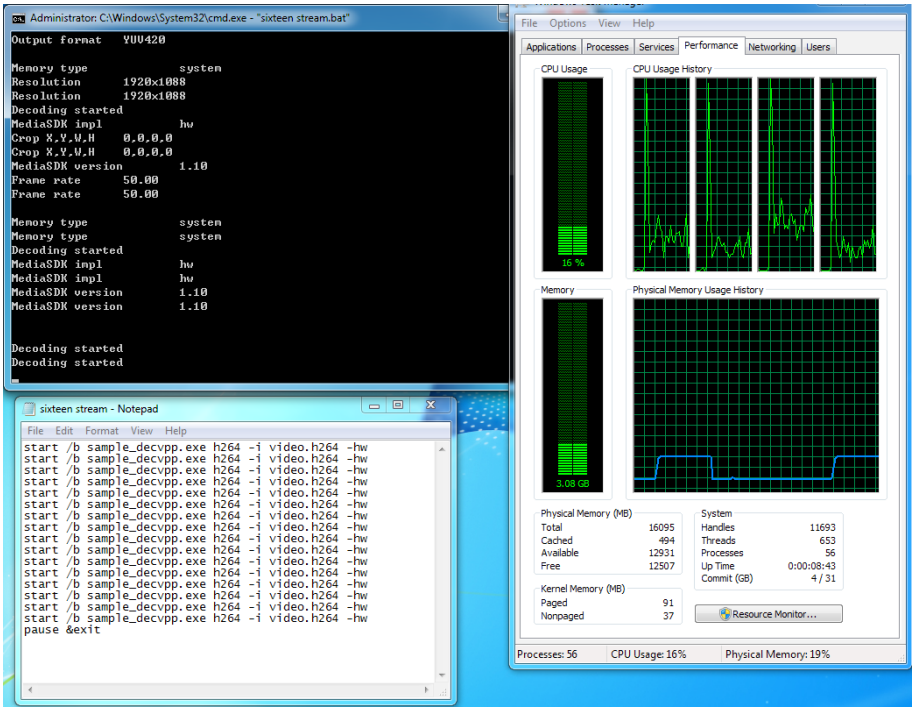
实现 Intel® Media SDK 隐式的 GPU 内存拷贝方法有两个需求：硬件上需要选择 Intel HD Graphics 第 7 代及以后的产品，相关的产品如 2012 年的第三代酷睿，2013 年的第四代酷睿，以及 2013 年的凌动产品等；软件上选择 Windows7、Windows8/8.1 以及最新版本的 Intel® Media SDK2014 和显示驱动程序（版本号大于等于 3740）。当软硬件配置满足要求并通过 Media SDK 内的 msdk_system_analyzer 确认 MSDK 的运行环境正常后用户就可以通过 Intel® Media SDK 编程来实现隐式内存拷贝功能了。对初学者来说可以下载 Intel 开发者网站内的 Media Solution Portal 主页发布的“Video Decoding Sample”，安装后参考 sample_decvpp 的 VPP 代码来学习使用 VPP 模块实现内存拷贝，基本原则就是指定 VPP 的输入为显存，VPP 输出为系统内存，此时仍可以指定 VPP 的其它如色彩格式转换等功能，设定完成后 VPP 即可完成数据由显存到内存的转移过程。

学习 sample_decvpp 的源代码可以看到当命令行参数使用“-hw”时该程序会使用 GPU 硬件加速的方式执行，同时创建显存页面作为内部运行时的视频解码输出以及 VPP 模块输入。另一方面，如果命令行参数没有指定最终的输出内存类型的话，该程序会默认创建系统内存作为最终 VPP 模块的输出，因此 VPP 模块内部实现了显存内容到系统内存的转移。下图是 E3-1225V3 平台在最新的 Intel® Media SDK 和显示驱动环境下运行 16 路 1080p 解码并输出到系统内存的运行过程截图，可以看到此时 CPU 占用率仅有 16% 左右。

4.4 利用 OpenCL 加速

除了使用 Intel® Media SDK 外，用户也可以利用 OpenCL 的软件实

图 3. 16 路高清解码并 GPU 拷贝到系统内存运行情况



来源：英特尔公司 IOTG 事业部

现来辅助加速 CPU 和 GPU 之间的数据交互。OpenCL（全称 Open Computing Language）既是一个面向异构系统通用目的并行编程的开放式、免费标准，也是一个统一的编程环境，便于软件开发人员为高性能计算系统编写高效轻便的代码，可以充分利用系统中的多核心 CPU、GPU、Cell、DSP 等其他并行处理器完成计算任务。Intel 从 2012 年起发布了 Intel® SDK for OpenCL™ Applications 的开发套件，为开发人员在 Intel 平台上开发 OpenCL 应用提供了编辑、编译、调试、分析的开发环境，支持 Windows，Linux、Android 等操作系统，除 Intel 的 CPU 和 GPU 外还支持 Intel® Xeon Phi™ 高性能协处理器。现在 Intel® SDK for OpenCL™ Applications 已经实现了对 OpenCL1.2 规范的 CPU、GPU 全面支持，并将为第五代酷睿平台提供 OpenCL2.0 规范的支持。

OpenCL 规范中也没有直接可用于显存到内存拷贝的命令，但是如前文所说，如果将 Tile 格式的图像数据先转换成线性数据再共享给 CPU 访问的话也能大大提高 CPU 的访问效率。这使我们可以利用 OpenCL 中像 clEnqueueCopyImageToBuffer 这种标准接口函数在 Intel 平台上实现基于 GPU 加速的数据格式转换，该函数将帮助客户将一个 Tile 格式的图像数据转换并输出到一个线性的缓冲区（Buffer）中。执行完 clEnqueueCopyImageToBuffer 之后，用户可以通过 clEnqueueMapBuffer 的方式对线性的缓冲区进行共享访问。这种方法实际上仍执行了一次 GPU 的拷贝操作，区别是拷贝的源和目的位置都在显存内部，拷贝后再共享出缓冲区的地址供 CPU 读取。OpenCL 的方法可以为客户提供 Intel® Media SDK 之外的另一种选择，同时也有利于之后基于 GPU 和 CPU 协同工作的视频分析实现，但仍要注意 GPU 和 CPU 对访问区域的互斥性。

5 总结

殊途同归，基于本文的讨论可以看出实现 CPU/GPU 之间的数据交互可以有多种方式，用户可以根据不同的应用场景和所使用的硬件平台选择最适合的方式

来实现数据交互，例如，线性的 Buffer 数据可以直接采用地址共享进行数据交互，对 Tile 格式且尺寸较大的图像数据采用 MSDK 的 GPU 拷贝方式效率更高等等。

总而言之，Intel 为客户提供了丰富的单芯片异构系统内部的数据交互解决方案，让用户可以充分利用 Intel 产品中的 CPU 和 GPU 的计算能力来实现和扩展自己的产品功能。

表 2. CPU/GPU 数据交互总结

方案	优点	缺点
直接地址共享	a. 逻辑简单 b. 有标准接口函数 c. 节省内存使用量	a. 共享会强制 CPU/GPU 在访问数据上的同步 b. CPU 对 Tile 数据访问效率低
CPU 拷贝	a. 缩短了地址共享导致的同步时间 b. Fastcopy 在低负载情况效率很高	a. 消耗大量 CPU 资源 b. Memcpy 效率较低 c. Fastcopy 需要 SSE/AVX 指令集支持
基于 MSDK 的 GPU 拷贝	a. CPU 占用率低 b. 充分利用 Intel GPU 内的 EU 单元，对解码性能无影响 c. CPU/GPU 同步依赖性非常低	a. 依赖于 Media SDK，只适用于视频解码数据 b. 隐含在VPP模块中，缺少专用接口函数
基于 OpenCL 的实现	a. CPU 占用率低 b. 有标准接口函数 c. 跨平台兼容性好	a. 不是完整的数据拷贝，数据仍需要地址映射后共享