



DEEP  
LEARNING  
INSTITUTE

## DRIVE AGX 上的 CUDA

本实验包含以下主题：

### [预备知识和登录说明](#)

#### 1. [简介](#)

##### [1.1 DRIVE AGX 平台概览](#)

##### [1.2 Tegra 上的 CUDA](#)

#### 2. [内存管理](#)

##### [2.1 内存布局](#)

##### [2.2 内存类型](#)

###### [2.2.1 常规内存](#)

###### [2.2.2 固定内存](#)

###### [2.2.3 统一内存](#)

###### [2.2.3 内存总结](#)

#### 3. [优化](#)

##### [3.1 WMMA内在函数](#)

##### [3.2 MNIST示例](#)

###### [3.2.1 FP16 算术基元](#)

###### [3.2.2 INT8 算术基元](#)

###### [3.2.3 CUDA 流](#)

###### [3.2.4 负载均衡](#)

###### [3.2.5 优化总结](#)

## 预备知识

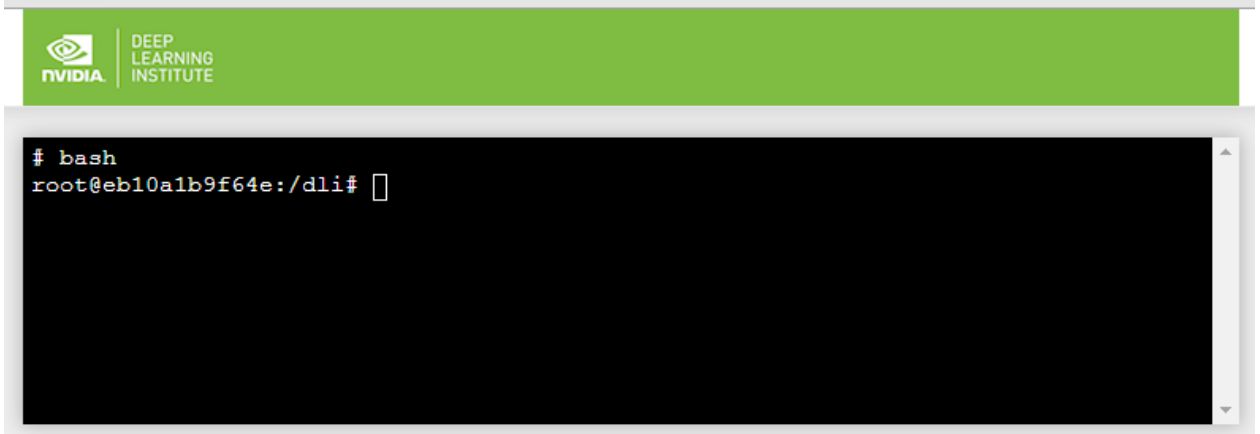
要充分利用本实验，您应具备以下知识：

1. C/C++ 编程基础知识。
2. 基本了解 CUDA 编程模型。不过，即便没有任何 CUDA 编程经验，也能完成实验。

## DRIVE AGX 平台登录说明

我们已通过 DLI 为您提供 DRIVE AGX 平台的云实例。这包括 VPN 访问凭据和 DRIVE AGX 单元的特定 IP 地址。您将使用此 Jupyter Notebook 的终端功能，通过通向 DRIVE AGX 平台实例的 VPN 隧道以 SSH 连接到终端。（如果您可以直接访问 DRIVE AGX 平台，请使用 [CTRL-T] 打开终端以执行这些命令。）

[单击此处 \(./terminals/vpn\)](#) 以在浏览器中打开终端窗口。在提示符处输入 `bash`。



```

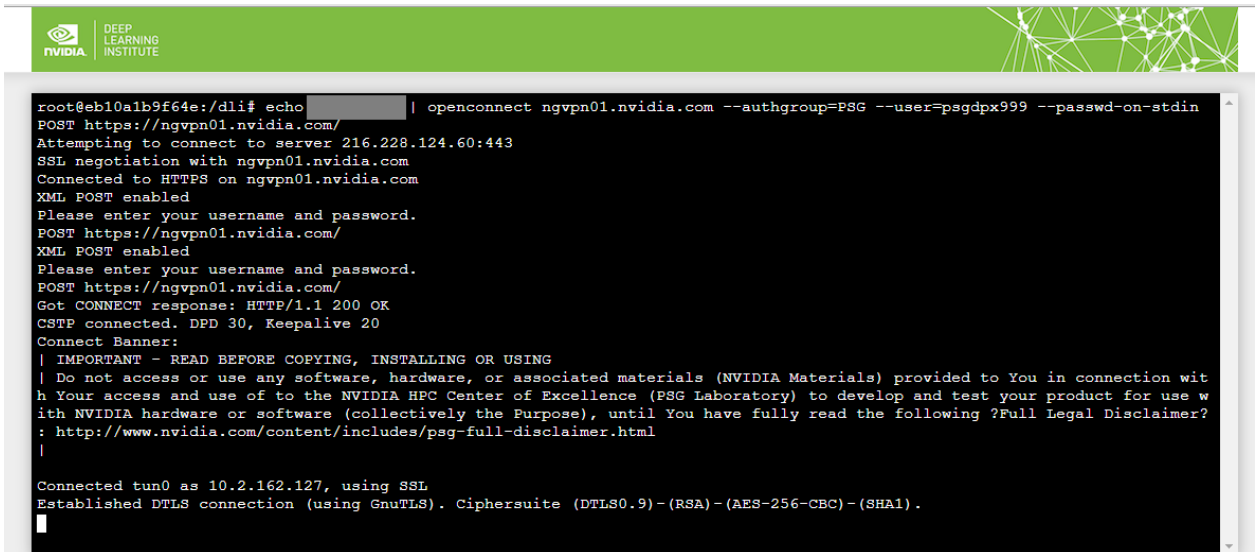
# bash
root@eb10a1b9f64e:/dli#

```

使用您的 VPN 凭据，将以下代码行输入终端，正确插入您特定的 `vpn_password` 和 `vpn_username`：

```
# echo <vpn_password> | openconnect ngvpn01.nvidia.com --authgroup=PSG --user=
<vpn_username> --passwd-on-stdin
```

响应应类似于下面显示的示例。



```

root@eb10a1b9f64e:/dli# echo [redacted] | openconnect ngvpn01.nvidia.com --authgroup=PSG --user=psgdp999 --passwd-on-stdin
POST https://ngvpn01.nvidia.com/
Attempting to connect to server 216.228.124.60:443
SSL negotiation with ngvpn01.nvidia.com
Connected to HTTPS on ngvpn01.nvidia.com
XML POST enabled
Please enter your username and password.
POST https://ngvpn01.nvidia.com/
XML POST enabled
Please enter your username and password.
POST https://ngvpn01.nvidia.com/
Got CONNECT response: HTTP/1.1 200 OK
CSTP connected. DPD 30, Keepalive 20
Connect Banner:
| IMPORTANT - READ BEFORE COPYING, INSTALLING OR USING
| Do not access or use any software, hardware, or associated materials (NVIDIA Materials) provided to You in connection with
| Your access and use of the NVIDIA HPC Center of Excellence (PSG Laboratory) to develop and test your product for use with
| NVIDIA hardware or software (collectively the Purpose), until You have fully read the following Full Legal Disclaimer?
| : http://www.nvidia.com/content/includes/psg-full-disclaimer.html
|
Connected tun0 as 10.2.162.127, using SSL
Established DTLS connection (using GnuTLS). Ciphersuite (DTLS0.9)-(RSA)-(AES-256-CBC)-(SHA1).

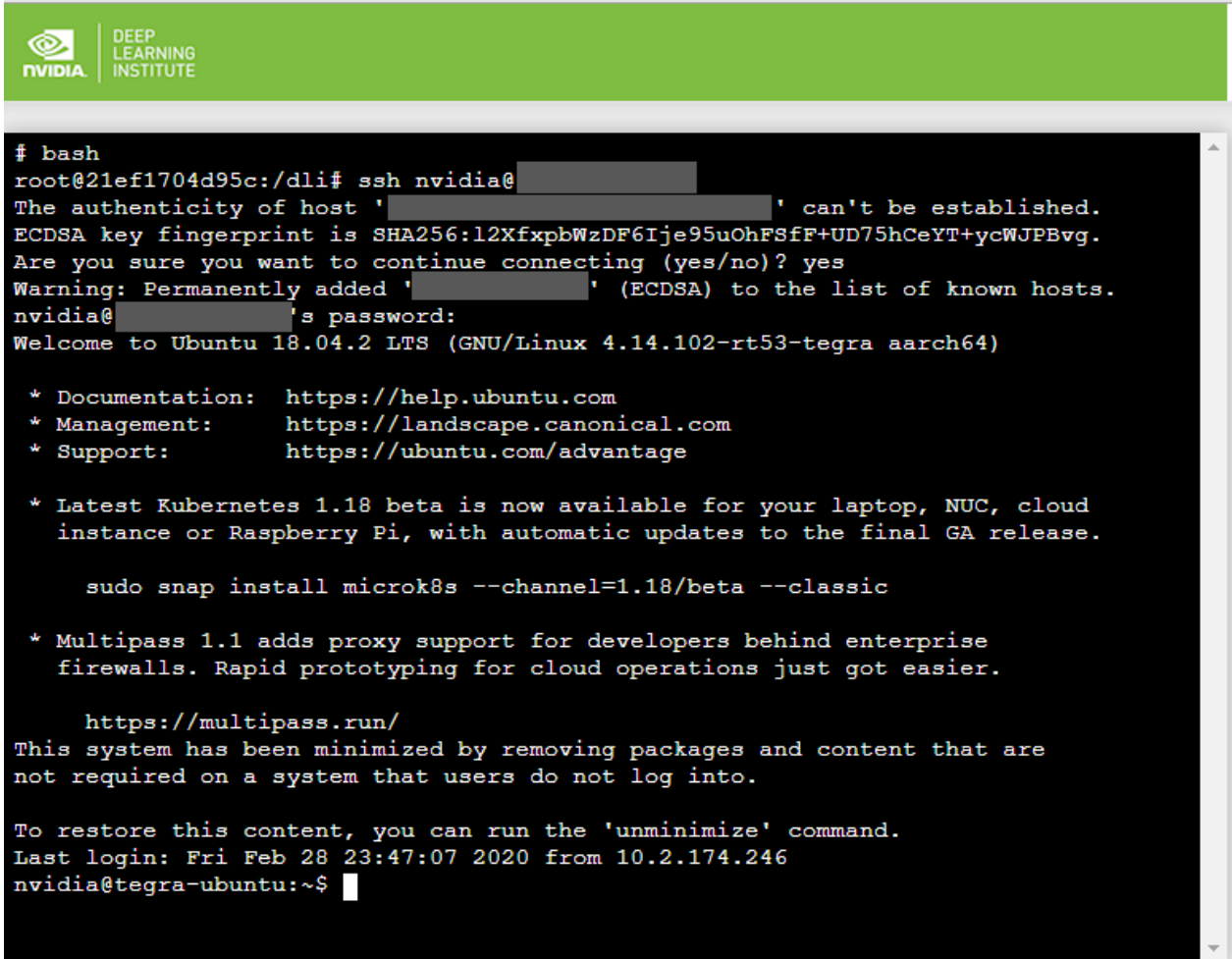
```

[单击此处 \(./terminals/agx\)](#) 以在浏览器中打开一个新的终端窗口（不要关闭 VPN 终端），用于通过 SSH 访问 DRIVE AGX。

- 与之前一样，在提示符处输入 `bash`。
- 使用提供给您的 DRIVE AGX IP 地址，复制/粘贴下方的命令进行登录。
- 如果系统询问“Are you sure you want to continue connecting (yes/no)?”（是否确定要继续连接 [是/否]?），回答 `yes`
- 系统提示时，输入密码 `nvidia`。

```
# ssh nvidia@<DRIVE_AGX_IP>
```

您应看到类似下面的内容，最后一个提示符表示成功登录。



```
# bash
root@21ef1704d95c:/dli# ssh nvidia@
The authenticity of host ' ' can't be established.
ECDSA key fingerprint is SHA256:l2XfxpbWzDF6Ije95uOhFSfF+UD75hCeYT+ycWJPBvg.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added ' ' (ECDSA) to the list of known hosts.
nvidia@'s password:
Welcome to Ubuntu 18.04.2 LTS (GNU/Linux 4.14.102-rt53-tegra aarch64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

 * Latest Kubernetes 1.18 beta is now available for your laptop, NUC, cloud
   instance or Raspberry Pi, with automatic updates to the final GA release.

   sudo snap install microk8s --channel=1.18/beta --classic

 * Multipass 1.1 adds proxy support for developers behind enterprise
   firewalls. Rapid prototyping for cloud operations just got easier.

   https://multipass.run/
This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Fri Feb 28 23:47:07 2020 from 10.2.174.246
nvidia@tegra-ubuntu:~$
```

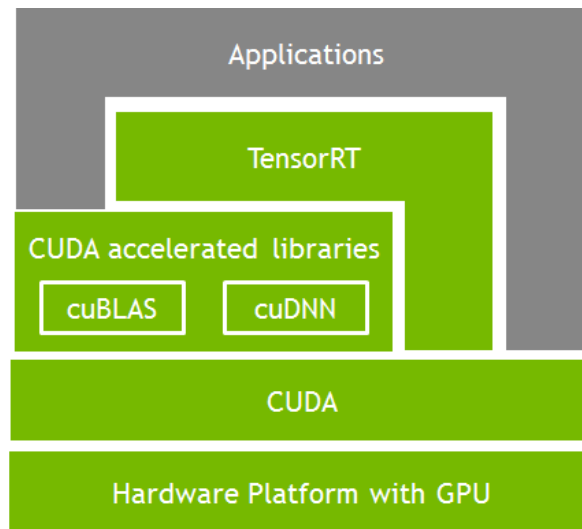
由于本实验专门针对 DRIVE AGX 平台设计，因此灰色（或米色）单元格中提供的所有命令均应通过 DRIVE AGX 平台上的终端执行。为方便起见，您可以复制/粘贴这些命令。

在本实验中 DRIVE AGX 平台的 `/home/nvidia` 中，应有一个名为 `CUDA_on_DriveAGX` 的目录。

## 1. 简介

NVIDIA DRIVE AGX 平台是用于打造自动驾驶汽车的热门平台之一。它提供了构建自动驾驶应用程序所需的大量计算资源。NVIDIA DRIVE 平台使用可扩展架构来支持不同自动驾驶级别（级别 2 至级别 5），并附带丰富的软件堆栈。它为软件堆栈提供了多个入口点，这使开发者能够灵活使用较低级别的功能块（如 CUDA）。

CUDA 是 NVIDIA 专为图形处理器 (GPU) 上的通用计算而开发的并行计算平台和编程模型。CUDA 提供了 GPU 编程的基本方法，并基于 GPU 构建了所有 GPU 加速库（如 cuDNN 和 TensorRT）。这使开发者能够构建高度定制的应用程序，从而利用 GPU 的强大功能。例如，如果用户希望在其深度神经网络中尝试使用新类型的层，而标准的 GPU 加速库（如 CuDNN 或 TensorRT 框架）不支持该类型的层，那么用户仍可通过为新层开发 CUDA 核函数，实现在 DRIVE AGX 平台上使用 GPU 加速应用程序。

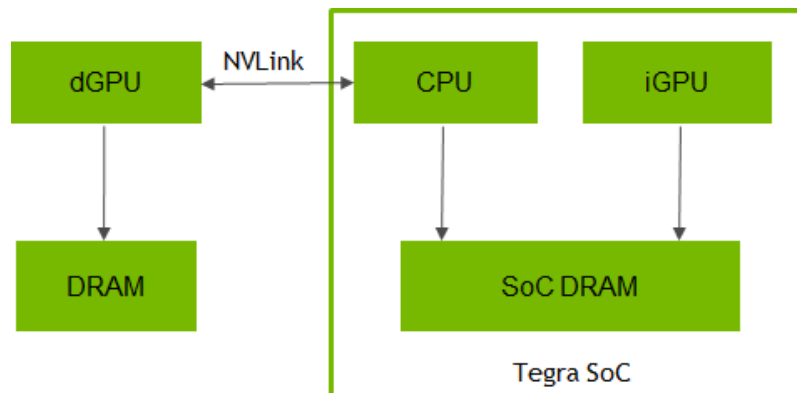


本实验的目的是综述 DRIVE AGX 平台上的 CUDA 编程。它分为以下三个部分：

1. 简介：首先是基本介绍和 DRIVE AGX 平台系统概述。
2. 内存管理：使用任意大小的矩阵乘法（更改矩阵大小将改变数据传输的大小）为例，展现 DRIVE AGX 平台上提供的不同类型内存的有效性。
3. 优化：从只在 CPU 上对 MNIST 数据集的 50,000 张图像进行推理这一简单问题入手，检查使用 CPU 时的运行时间，然后使用带有 CUDA 内核的 GPU 演示加速性能。接着，使用混合精度，在准确度没有或很小的下降的情况下进一步加快推理速度。最后，使用 CUDA 流让数据传输和内核执行重叠进行，以缩短运行时间，并将图像推理分配到 iGPU 和 dGPU 上以执行负载均衡。

## 1.1 DRIVE AGX 平台概览

下图显示了 DRIVE AGX 平台的系统概览。NVIDIA DRIVE AGX 平台是支持 DRIVE AGX Xavier 和 DRIVE AGX Pegasus 两种设计的通用平台。DRIVE AGX Xavier 包含两个类似的系统副本（名为 TegraA 和 TegraB），每个副本包含一个 Tegra SoC（片上系统）。而 DRIVE AGX Pegasus 则还包含连接每个 Tegra SoC 的额外的独立 GPU (dGPU)。Tegra SoC 包含 CPU 和集成 GPU (iGPU)。CPU 复合体由 8 个 [Carmel 核心](https://en.wikichip.org/wiki/nvidia/microarchitectures/carmel) (<https://en.wikichip.org/wiki/nvidia/microarchitectures/carmel>) 组成。CPU 和 iGPU 共享通用的 SoC DRAM。与 iGPU 相比，dGPU 具有更多的流多处理器 (SM)，性能相对更加强大。如下所示，通过运行 `deviceQuery` 示例应用程序，可以获得 DRIVE AGX 平台上与 GPU 相关的完整的统计信息。



//在 DRIVE AGX 终端上尝试以下命令

```
$ cd ~/CUDA_on_DriveAGX/deviceQuery/
$ make
$ ./deviceQuery
```

deviceQuery 中的信息总结如下。iGPU 和 dGPU 均基于 [Volta 架构](https://en.wikipedia.org/wiki/Volta_%28microarchitecture%29) ([https://en.wikipedia.org/wiki/Volta\\_%28microarchitecture%29](https://en.wikipedia.org/wiki/Volta_%28microarchitecture%29)) 和 [Turing 架构](https://en.wikipedia.org/wiki/Turing_%28microarchitecture%29) ([https://en.wikipedia.org/wiki/Turing\\_%28microarchitecture%29](https://en.wikipedia.org/wiki/Turing_%28microarchitecture%29))，计算能力分别为 7.2 和 7.5。通过使用 `cudaSetDevice()` ([http://docs.nvidia.com/cuda/cuda-runtime-api/group\\_\\_CUDART\\_DEVICE.html#group\\_\\_CUDART\\_DEVICE\\_1g159587909ffa0791bbe4b40187](http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART_DEVICE.html#group__CUDART_DEVICE_1g159587909ffa0791bbe4b40187)) API 设置活跃的设备，用户可以在运行时选择 iGPU 或 dGPU。

PROPERTY	iGPU	dGPU
Name	Xavier	Graphics Device
SM/Cores per SM	8/64	44/64
Global Memory	24GB (shared RAM)	7.5 GB
L2 Cache	512 KB	4 MB
Compute Capability	7.2	7.5

## 1.2 Tegra 上的 CUDA

在 DRIVE AGX 平台上进行编程时所用的 CUDA 环境与使用 x86 工作站上的典型独立 GPU 进行编程时所用的环境类似。CUDA 驱动程序将底层硬件的差异抽象化。以下命令将通过 CUDA 编译器 (`nvcc` (<http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#ixzz59ogvyQjK>)) 编译 CUDA 代码示例。`nvcc` 编译器有助于抽象出 CUDA 编译的平台特定细节。这样就可以非常轻松地将为工作站设置编写的 CUDA 代码移植到其他环境（如 DRIVE AGX Tegra）。

```
$ cd ~/CUDA_on_DriveAGX
$ cd MatMul
$ make clean
$ make
$ ./matrixMulMem
```

此环境还提供开发者工具（如 Nsight Systems）来帮助分析 CUDA 代码。上面构建的示例可由 `nsys profile` 运行以分析应用程序。所有与 CUDA 相关的任务（如 CUDA 内核执行、内存传输、CUDA API）均可进行分析。

```
$ sudo /usr/local/cuda/bin/nsys profile --stats=true ./matrixMulMem
```

虽然可以轻松地将最初为 x86-dGPU 工作站设置开发的 CUDA 应用程序移植到 Tegra 环境，但为了实现最大性能，请务必记住一些硬件差异。下一节将介绍移植到 Tegra 环境时要考虑的这样一个主要因素：内存管理。

## 2. 内存管理

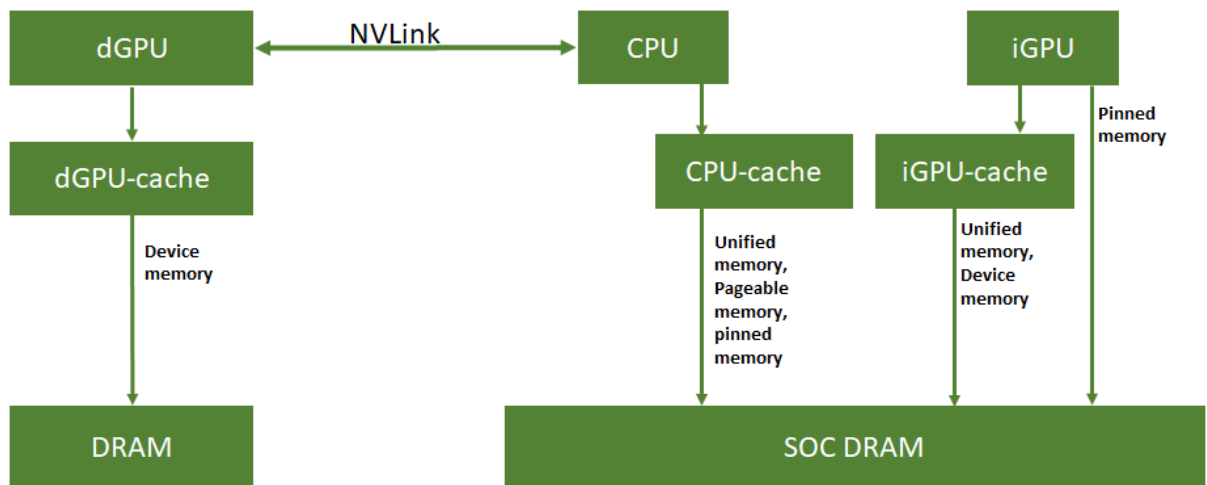
CUDA 编程模型提供三种不同类型的内存来处理 CPU 和 GPU 之间的数据交换。请务必根据应用程序的特性选择正确的内存类型，以便通过 GPU 获取最大性能。我们以矩阵乘法  $\mathbf{AB} = \mathbf{C}$  为例，查看这些内存类型对运行时间的影响。在输入矩阵 **A** 和 **B** 中填充随机值，并且可更改矩阵大小，以改变为了计算结果矩阵 **C**而需要通过 GPU 传输/访问的数据量。

### 2.1 内存布局

下图中显示了 DRIVE AGX 平台内存布局。iGPU 和 CPU 可共享 SoC DRAM。在 iGPU 和 CPU 之间不会对此内存进行硬分割而是按需分配内存，即只要 SoC-DRAM 上有足够的可用内存存，就可以执行对 iGPU 或 CPU 的内存请求。

CPU 和 iGPU 具有自己的缓存。对 SoC DRAM 的访问是否会进行缓存取决于所用的内存类型。

dGPU 包含专用 DRAM 和缓存。通过 NVLink 在 CPU 和 dGPU 之间进行存储数据的传输。



### 2.2 内存类型

如前所述，存在三种不同类型的内存缓冲区。对于 iGPU，所有类型的内存缓冲区都在同一物理内存 (SoC DRAM) 上分配，但会因 CPU 和 GPU 之间的地址空间管理方式、分页和缓存方式而有所不同。在后续章节中，术语“主机”指的是 CPU，而“设备”指的是 DRIVE AGX 平台上的 GPU。

#### 2.2.1 常规内存

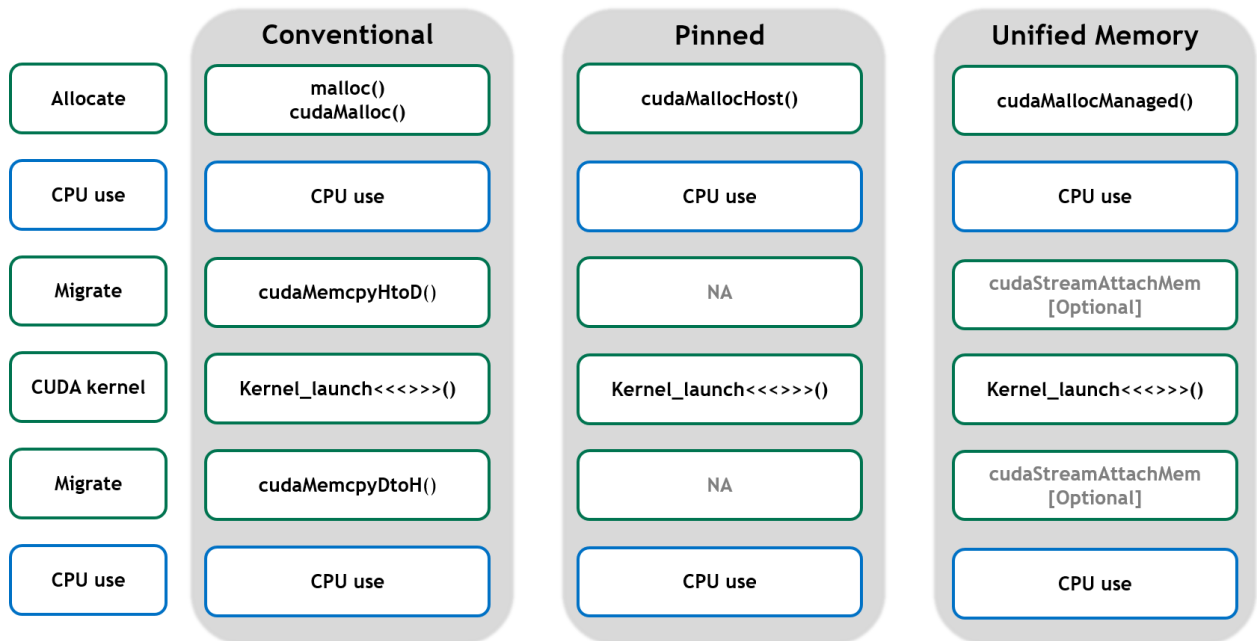
这是 CUDA 处理内存的常规方式。在此内存类型中，主机内存是指仅通过 CPU 直接寻址的内存空间。通过 CUDA 之外的调用（如 `malloc()`）来完成主机内存分配。对仅通过 GPU 直接寻址的设备内存，则使用 `cudaMalloc()` ([http://docs.nvidia.com/cuda/cuda-runtime-api/group\\_\\_CUDART\\_MEMORY.html#agroup\\_\\_CUDART\\_MEMORY\\_1g37d37965bfb4803b6d4e59](http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART_MEMORY.html#agroup__CUDART_MEMORY_1g37d37965bfb4803b6d4e59)) 等 CUDA API 进行分配。主机和设备无法访问对方的内存，必须将主机内存中的数据显式地复制到设备内存中，GPU 才能访问这些数据，反之亦然。在物理位置方面，iGPU 的主机内存和设备内存均在 SoC DRAM 上进行分配。dGPU 的设备内存存在专用的 dGPU DRAM 上进行分配。



下图总结了使用常规 CUDA 内存的工作流程（后续章节定义了固定内存和统一内存）。在矩阵乘法示例中，使用 `malloc()` 在主机内存上为输入矩阵 **A** 和 **B**，以及输出矩阵 **C** 分配内存。

同样，使用 `cudaMalloc()` 调用为 **A**、**B** 和 **C** 分配设备内存。CPU 将填充输入矩阵 **A** 和 **B** 的值。这些数据已从主机内存复制到设备内存，因为实际计算是由 GPU 完成的。CUDA 内核将对结果矩阵 **C** 进行计算。**C** 重新被复制到主机内存，以便 CPU 访问结果。

由于主机内存和设备内存是分别寻址的，因此要在主机和设备之间传输数据，总是需要使用显式内存复制。这也适用于与 CPU 共享同一物理 DRAM 的 iGPU。由于主机内存是在 CUDA 之外使用 `malloc()` 分配的，因此操作系统可以将其分页到硬盘。故此，在使用此类型的主机内存时，无法保证数据一定在 DRAM 中。这又称为\_可分页的主机内存\_。



请执行以下命令，以使用常规 CUDA 内存运行矩阵乘法的示例。使用常规内存的矩阵乘法的实现可在 `matMul()` 函数（位于 `matrixMulMem.cu` 文件中）中找到。终端上的示例输出如下所示。

```
$ cd ~/CUDA_on_DriveAGX/MatMul
$ make clean
$ make
$ ./matrixMulMem -hA=640 -wA=12800 -hB=12800 -wB=640 -device=1
```

```
nvidia@tegra-ubuntu:~/MatMul$ ./matrixMulMem -hA=640 -wA=12800 -hB=12800 -wB=640 -device=1
[Matrix Multiply Using CUDA] - Starting...
GPU Device 1: "Xavier" with compute capability 7.2
MatrixA(12800,640), MatrixB(640,12800)

Test#1: Global/Conventional Memory Test:
cudaMemcpy (64000.00 KB): 9.68 msec
Performance= 155.16 GFlop/s, Kernel Execution Time= 67.582 msec
Checking computed result for correctness: Result = PASS
```

上述命令执行的是大小为  $w_A \times h_A$  和  $w_B \times h_B$  的矩阵乘法运算。为 dGPU 选择 `device=0`，为 iGPU 选择 `device=1`。使用 iGPU 时，内存数据传输的总时间约为 11 毫秒，乘法运算时间为 64 毫秒。由于 dGPU 比 iGPU 功能更强大，因此 dGPU 乘法运算速度更快。此处的运行时间可用作参考，以便与使用其他内存类型的矩阵乘法的性能作比较。

## 2.2.2 固定内存

固定内存是指 SoC DRAM 上分配的不可分页的主机内存。它可以由设备直接访问，并且无需在主机和设备之间显式复制内存即可进行数据传输。因此，这种类型的内存也称为“零复制”内存。它使用 CUDA API `cudaMallocHost()` ([http://docs.nvidia.com/cuda/cuda-runtime-api/group\\_CUDART\\_MEMORY.html#group\\_CUDART\\_MEMORY\\_1gab84100ae1fa1b12eaca660](http://docs.nvidia.com/cuda/cuda-runtime-api/group_CUDART_MEMORY.html#group_CUDART_MEMORY_1gab84100ae1fa1b12eaca660)) 进行分配。主机和设备仍然使用不同的内存空间，因为主机内存获取的内存地址需要转换为有效的设备内存地址，才能通过设备进行访问。这一操作需要使用 `cudaHostGetDevicePointer()` ([http://docs.nvidia.com/cuda/cuda-runtime-api/group\\_CUDART\\_MEMORY.html#group\\_CUDART\\_MEMORY\\_1gc00502b44e5f1bdc0b42448](http://docs.nvidia.com/cuda/cuda-runtime-api/group_CUDART_MEMORY.html#group_CUDART_MEMORY_1gc00502b44e5f1bdc0b42448)) 调用完成。在支持统一虚拟寻址 (UVA) 的系统 (如 DRIVE AGX) 上，我们可以直接使用指针。因此，不需要使用 `cudaHostGetDevicePointer()` 调用。

DRIVE AGX 平台与 DRIVE PX2 的一个主要区别在于，由于具备 I/O 一致性，固定显存可在 CPU 上进行缓存。按照如下指示修改代码，以便在矩阵乘法示例中使用固定内存而不是常规内存。

注意：要通过终端编辑文件，请使用 `vi matrixMulMem.cu`。您可能会发现，使用 `:set number` 命令显示这些练习的行号很有帮助。要了解有关此编辑器的更多帮助，请参阅 [Vi Reference Card](http://web.mit.edu/merolish/Public/vi-ref.pdf) (<http://web.mit.edu/merolish/Public/vi-ref.pdf>) 或互联网上提供的许多其他参考之一。

```
$ make clean
$ make
$ ./matrixMulMem -hA=640 -wA=12800 -hB=12800 -wB=640 -device=1 -type=1 //Runs on iGPU
```

请注意，执行时间略有增加。这可能要归因于 GPU 上固定内存的非缓存行为。矩阵乘法运算要求多次访问相同的数据。对于每次访问，总执行时间都会略有增加。由于不涉及显式复制内存，因此内存访问方面增加的时间会在执行时间上显示出来。

这个问题的一个解决方案是使用共享内存。共享内存是 SM (流多处理器) 的所有核心(Cores)共用的更快的片上内存。这可以视为用户定义的缓存。可通过将小型数据块引入共享内存来计算输出的子矩阵，以此执行矩阵乘法运算。其原理是将子矩阵所需的数据 (由单个线程块计算) 引入共享内存，以加快重复数据访问的速度。这是通用的优化步骤，通常可以提高性能，即使使用其他类型的内存也是如此。启用以下宏，即可启用共享内存。

```
$ ./matrixMulMem -hA=640 -wA=12800 -hB=12800 -wB=640 -device=1 -type=1 -
enableSharedMemory=1 // Runs on iGPU
$ ./matrixMulMem -hA=640 -wA=12800 -hB=12800 -wB=640 -device=0 -type=1 -
enableSharedMemory=1 // Runs on dGPU
```

下表总结了不同矩阵大小的执行时间及其使用共享内存的数据传输速率。请注意，由于缓存旁路，固定内存更适用于未重用的小块数据。



- For iGPU:

Matrix dim (A and B)	Data transfer (MB)	Pinned Memory – Total time (ms)	Conventional Memory – Total time (memory transfer + exec) (ms)
12800x640	65.53	34.49	43.79 (9.29+34.5)
1280x64	0.65	0.12	0.31 (0.19 + 0.12)
128x64	0.06	0.05	0.22 (0.17+.05)

- For dGPU:

Matrix dim (A and B)	Data transfer (MB)	Pinned Memory – Total time (ms)	Conventional Memory – Total time (memory transfer + exec) (ms)
12800x640	65.53	74.3	19.43 (8.7+10.73)
1280x64	0.65	0.15	0.36 (0.23 + 0.13)
128x64	0.06	0.05	0.19 (0.15+.04)

Note: Numbers are taken by using shared memory

### 2.2.3 统一内存

我们还记得，在常规内存中，设备和主机使用不同的内存空间。在 Tegra 这样的系统上，CPU 和 iGPU 共有物理 DRAM，但内存模型却强制使用冗余内存拷贝，以在 CPU 和 iGPU 之间共享数据。在某种程度上，使用固定内存虽可以避免这种情况，但是它缺少缓存，仍会影响执行时间。

统一内存使用统一的内存空间，这意味着某个内存地址的指针对 CPU 和 GPU 均有效。这种内存不需要显式的内存复制，因为数据迁移是由 CUDA 驱动程序管理的。Tegra 上的 CUDA 驱动程序可避免在 CPU 和 iGPU 之间交换数据时使用冗余内存复制。

在 CPU 和 iGPU 上均可缓存统一内存。在 Tegra 上，若在应用程序中使用统一内存，需要在内核启动、同步和预取提示调用期间进行额外的一致性和缓存维护操作。因此，只有在 CPU 和 iGPU 上对大缓冲区进行频繁且重复的访问时，统一内存优先于固定内存。在这种情况下，在重复访问方面获得的收益可抵消缓存维护成本。

**注意：**在需要低延迟和可预测性的应用程序中，不推荐使用统一内存。例如，为了维护 GPU 和 CPU 之间的一致性，驱动程序需要执行一些缓存操作，这会使命令流中出现意外延迟，而这种情况对行驶安全来说是不希望出现的。因此，对于汽车客户而言，不推荐在任何安全环境中都使用统一内存。请查看适用于 Tegra 的[移植注意事项 \(https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html\)](https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html)。

```
$ ./matrixMulMem -hA=640 -wA=12800 -hB=12800 -wB=640 -device=1 -type=2 -
enableSharedMemory=1 // Runs on iGPU
```

DRIVE AGX 平台的 dGPU 目前不支持统一内存。下表总结了 iGPU 上统一内存的使用结果。统一内存比常规内存速度更快。它花费的时间与内核执行时间大致相同，同时完全无需进行内存的冗余复制，即可在 CPU 和 iGPU 之间共享数据。

Matrix dim (A and B)	Data transfer (MB)	Unified Memory – Total time (ms)	Conventional Memory – Total time (memory transfer + exec) (ms)
12800x640	65.53	35.2	43.79 (9.29+34.5)
1280x64	0.65	0.66	0.31 (0.19 + 0.12)

Note: Numbers are taken by using shared memory

## 2.2.4 总结

下表根据应用程序的性质总结了使用哪种类型的内存来更大限度地提高性能的建议。

Memory Type	Recommended	Not Recommended
Conventional Memory	For large buffers with no or minimal data exchange between CPU and GPUs.	Frequent migration of data between CPU and GPU
Pinned Memory	For frequent data access from both CPU and GPU specifically for small buffers with less repetitive data access. Can also be used for large buffers when GPU only needs to access data once in a coalescing manner.	Repeated data access
Unified Memory	For frequent data access from both CPU and GPU with highly repetitive access (to offset the cost of cache coherency maintenance)	For applications with high predictability needs and low latency.

## 3. 优化

在数值计算中，精度和性能之间的折衷至关重要。许多科学计算都需要使用32位浮点数（FP32）和64位浮点数（FP64）计算进行高精度计算。通常，高精度计算会更复杂，并且导致与低精度整数计算相比，吞吐量较低。因此，重要的是要根据应用选择最佳的数据表示形式。

研究人员发现，由于反向传播算法，深度神经网络体系结构通常具有自然的抗错能力。对于典型的CNN网络，一旦对网络进行了训练，则其半精度（FP16）和8位整数精度（INT8）可以用于推理而不会造成很大的准确性损失。DRIVE AGX平台上的GPU提供了本机指令来利用DNN的这一特性，从而通过低精度执行计算来加快推理速度。此处使用的MNIST示例模型训练时使用的是FP32精度。我们考虑以较低的精度（FP16和INT8）进行推理，并对比了性能增益和准确度损失。

在本节中，我们将讨论CUDA混合精度内在函数对增强应用程序性能的影响。我们以矩阵乘法为例来演示WMMA内在函数，并以MNIST为例来演示FP16和INT8内在函数。我们还将讨论CUDA流和负载均衡以进一步提高性能。

### 3.1 WMMA内在函数

NVIDIA Volta / Turing GPU具有张量内核，可提供更好的深度学习性能。使用WMMA指令可使用张量内核实现高效的矩阵乘法运算。每个张量内核每个时钟执行64个浮点FMA混合精度运算。有关WMMA内在函数的更多详细信息，请查阅[Warp矩阵函数 \(https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma\)](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma)。请使用以下命令在矩阵乘法运算中启用WMMA内在函数。

```
$ ./matrixMulMem -hA=640 -wA=12800 -hB=12800 -wB=640 -device=1 -useWMMA=1 // to enable WMMA on iGPU
$ ./matrixMulMem -hA=640 -wA=12800 -hB=12800 -wB=640 -device=1 -useWMMA=0 // to disable WMMA on iGPU
$ ./matrixMulMem -hA=640 -wA=12800 -hB=12800 -wB=640 -device=0 -useWMMA=1 // to enable WMMA on dGPU
$ ./matrixMulMem -hA=640 -wA=12800 -hB=12800 -wB=640 -device=0 -useWMMA=0 // to disable WMMA on dGPU
```

可以注意到，在iGPU上，使用WMMA内在函数的执行时间约为18毫秒。如果我们将共享内存与WMMA内在函数一起使用，则可以进一步提高性能。下表总结了各种输入矩阵大小下的WMMA内在函数的影响。

- For iGPU:

Matrix dim (A and B)	Without WMMA(ms)	With WMMA (ms)
12800x640	67.2	18.2
1280x640	6.37	1.18
4096x4096	865.3	127.3

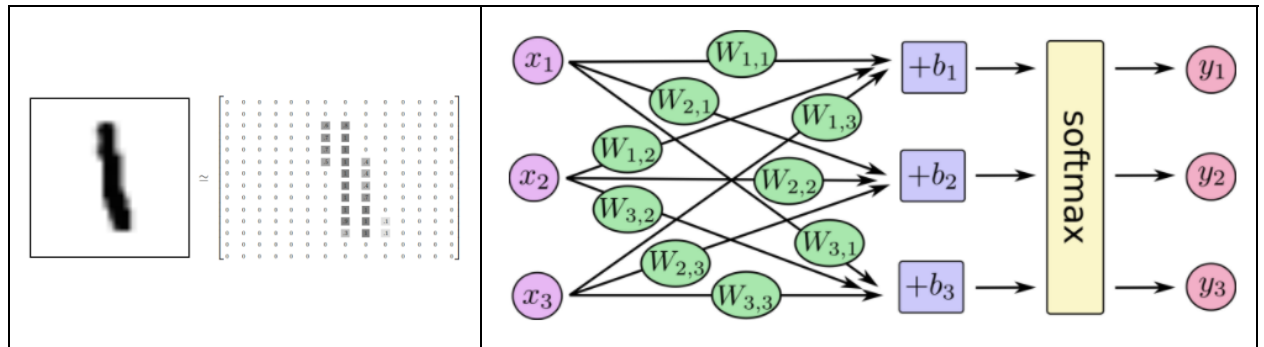
- For dGPU:

Matrix dim (A and B)	Without WMMA(ms)	With WMMA (ms)
12800x640	14.6	3.32
1280x640	1.39	0.44
4096x4096	179.04	36.85

Note: Considered only kernel execution time

## 3.2 MNIST示例

在本节中，我们考虑使用简单的 MNIST 推理示例，来演示 CUDA 中提供的一些可能的优化技术。MNIST 是一个大小为 28x28 的手写体数字图像数据库。我们使用单层全连接网络对数字进行分类。网络输入为 784 像素值向量（即 28x28 平面图像展平成一维向量）。



该网络已使用 50,000 张训练图像进行了训练。以下命令显示了单张图像的输入图像和网络输出。每次运行时，从 50,000 张图像中随机选择图像进行推理。多次运行以查看网络输出。

```
$ cd ~/CUDA_on_DriveAGX/MNIST_inference
$ make clean
$ make
$ ./inference
```

```
nvidia@tegra-ubuntu:~/CUDA_on_DriveAGX/MNIST_inference$ ./inference
Displaying randomly chosen input image:
```

```
Running on the CPU
CPU based time = 0.316000 ms
output of trained network
Probability of different classes
```

	0	1	2	3	4	5	6	7	8	9
NUMBER =	0	1	2	3	4	5	6	7	8	9
PROBABILITY =	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
PREDICTED NUMBER =	2									

在推理过程中，计算结果涉及权重矩阵 [784x10] 和展平后的输入向量 [1x784] 的矩阵乘法运算，如下图所示。

输出预测： $y = \text{softmax}(Wx + b)$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax} \left( \begin{bmatrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + b_3 \end{bmatrix} \right) \Rightarrow \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax} \left( \begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

由于使用优化技术来衡量运行时的改进情况相对比较简单，因此使用了批量大小为 50,000 的图像。以下命令对批量大小为 50,000 的图像运行相同的网络，并使用网络的运行时间来显示网络的准确度。

```
$ ./inference 50000
```

```
nvidia@tegra-ubuntu:~/CUDA_on_DriveAGX/MNIST_inference$ ./inference 50000
Running on the CPU
CPU based time = 1409.715942 ms
total matched images 45980 of 50000 with accuracy 0.919600
```

对于批量大小为 50,000 的图像，内核运行时间大约为 1409 毫秒。该网络能够正确预测 45,980 张图像，准确率为 91.96%。到目前为止，推理只使用 CPU 进行计算。基于 CPU 的推理在函数 `inferenceCPU()` 中得到实现。大型矩阵乘法等任务可大规模并行进行，因此是通过 GPU 实现加速的理想选择。简单的 CUDA 内核用于执行矩阵乘法运算。按照指示修改代码，以启用 GPU 加速的推理。

$$y = \text{softmax}(Wx + b)$$

Accelerated  
with CUDA

//待办事项 在 inference.cu 中，为第 841 行添加注释，并为第 842 行取消注释，以启用 CUDA 加速的函数 inferenceGPU()。

```
$ make clean
```

```
$ make
```

```
$ ./inference 50000
```

```
nvidia@tegra-ubuntu:~/CUDA_on_DriveAGX/MNIST_inference$ ./inference 50000
```

```
Using FP32 intrinsics and running on Device Graphics Device
```

```
kernel time is 5.284192 msec
```

```
h2d time is 20.097376 msec
```

```
d2h time is 0.640224 msec
```

```
Total time is 26.021793 msec
```

```
total matched images 45980 of 50000 with accuracy 0.919600
```

dGPU 上的内核运行时间大约是 5 毫秒。由于输入图像和权重矩阵需要传输到 GPU 内存，因此还打印了用于计算的从主机到设备 (h2d) 的数据传输以及计算结果的设备到主机 (d2h) 传输所用的时间。在这种情况下，内存传输大约需要 20 毫秒。

要切换 iGPU 和 dGPU 以进行计算，请修改宏 DEVICE\_NUM (dGPU 为 0, iGPU 为 1)。使用 dGPU 加速可实现超过 50 倍的加速！

### 3.2.1 FP16 算术基元

NVIDIA Volta/Turing 架构增加了对 FP16 运算的本地支持，可以在深度学习应用程序中加以利用。

hfma2() 指令可以在一个时钟周期内执行两次乘法累加运算。此指令的数据存储为 half2，它将两个 FP16 值打包在一个 32 位寄存器中。可使用 floats2half2\_rn() 将浮点数据转换为 half2 格式。

//待办事项 在 inference.cu 中，通过在第 24 行中将 DEVICE\_NUM 设置为 0/1，切换回 dGPU/iGPU。

//待办事项 在 Inference.cu 中，为第 842 行添加注释，并为第 843 行取消注释，以使用 inferenceGPU\_FP16()。然后重新编译。

```
$ make clean
```

```
$ make
```

```
$ ./inference 50000
```

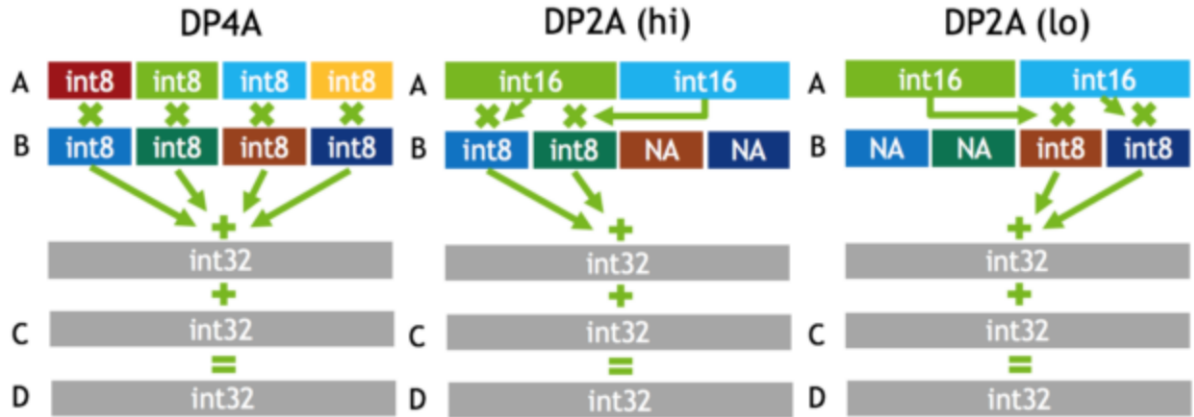
可以看到，运行时间有相当大的改进而不会造成准确度方面的任何损失。下表总结了 FP16 计算的结果。

Precision	iGPU (Kernel /Total) (ms)	d-GPU(Kernel/Total) (ms)	Accuracy (50000 images)
FP32	20.15/43.09	5.26/28.2	91.96
FP16	14.57/37.16	4.33/25.6	91.96

### 3.2.2 INT8 算术基元

Turing 架构为双路和四路整数点积运算提供了专门指令，非常适于加速深度学习推理工作负载。两个可用的内在函数是：

- **dp4a intrinsic** - 执行四个 8 位整数的点积运算和 32 位整数累加运算。
- **dp2a intrinsic** - 在一个向量中的两个 16 位整数与另一向量中的两个 8 位整数之间执行双元素点积运算，然后执行 32 位整数累加运算。



执行以下更改以启用基于 DP4A 的内核。理论上，使用 INT8 算法比 FP32 算法可以获得 4 倍的增益。应用程序中的近似增益是 2 倍，因为这是有内存限制的计算。准确度下降幅度也非常低 (0.09%)。dGPU 的输入数据传输 (h2d) 由约 20 毫秒降至约 5 毫秒。这是因为输入数据现在以 INT8 精度表示，而这只是同一数据 FP32 表示的大小的四分之一。

```
//待办事项 在 inference.cu 中，通过在第 24 行中将 DEVICE_NUM 设置为 0/1，切换回
dGPU/iGPU。
//待办事项 在 inference.cu 中，为第 843 行添加注释，并为第 844 行取消注释，以使用
inferenceGPU4A()
$ make clean
$ make
$ ./inference 50000
```

Precision	iGPU (Kernel /Total) (ms)	d-GPU(Kernel/Total) (ms)	Accuracy (50000 images)
FP32	20.15/43.09	5.26/28.2	91.96
INT8	13.37/16.57	3.92/9.47	91.87

到目前为止，通过 MNIST 示例，我们已了解 GPU 如何加速推理工作负载。在 GPU 上使用低精度算法可以进一步缩短运行时间。通过缩短 CUDA 内核本身的执行时间并缩短内存传输持续时间可缩短运行时间。

除了内在函数调用，CUDA 内核还包含其他运算。通过注释掉内核中的内在函数代码后再运行，可以计算出这些其他运算所花费的时间。在 CUDA 内核中注释掉 FP32、FP16 和 INT8 内部代码后，内核计算与删除之前非常相似。这表明，内核中出现的时间递减是由于依次采用 FP32->FP16->INT8 得到的，这可以测量我们在使用这些内部函数时的确切加速。



```
//待办事项 在 inference.cu 中，通过在第 24 行中将 DEVICE_NUM 设置为 0/1，切换回
dGPU/iGPU。
//待办事项 在 inference.cu 中，为第 108、337、484 行添加注释，并为第 842、843、844 行取消注释
$ make clean
$ make
$ ./inference 50000
```

```
nvidia@tegra-ubuntu:~/CUDA_on_DriveAGX/MNIST_inference$ ./inference 50000

Using FP32 intrinsics and running on Device Graphics Device
kernel time is 3.176064 msec
h2d time is 22.424671 msec
d2h time is 0.592544 msec
Total time is 26.193279 msec

Using FP16/half2 intrinsics and running on Device Graphics Device
kernel time is 3.072768 msec
h2d time is 22.320448 msec
d2h time is 0.360960 msec
Total time is 25.754175 msec

Using DP4A intrinsics and running on Device Graphics Device
kernel time is 3.160064 msec
h2d time is 5.818688 msec
d2h time is 0.139392 msec
Total time is 11.776768 msec
total matched images 4460 of 50000 with accuracy 0.089200
```

下表收集了 CUDA 内核在内在函数代码和非内在函数代码方面花费的时间。请注意，在所有情况下，非内在函数的代码都保持不变。使用 INT8 内在函数代码的速度比 FP32 快 2.7 倍。

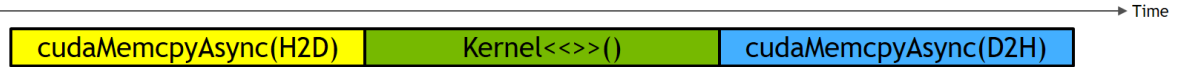
Precision	Intrinsics (ms)	Non intrinsics(ms)	Total
FP32	2.1	3.16	5.26
FP16	1.26	3.07	4.33
INT8	0.78	3.14	3.92

到目前为止，最佳运行时间是通过使用具有 INT8 推理的 dGPU 实现的，总运行时间为 ~9.47 毫秒（3.92 毫秒用于内核执行，5.5 毫秒用于数据传输）。在后续优化步骤中，我们会考虑这种情况，通过将内存传输与内核执行相重叠，进一步缩短总运行时间。

### 3.2.3 CUDA 流

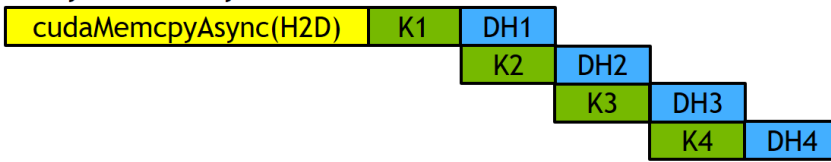
下图描述了 CUDA 内核的典型工作流程。GPU 上的并行计算所需的输入数据首先从主机内存传输到 GPU 设备内存。在所有数据都传输到设备内存后，系统便会启动内核来执行计算。然后将结果重新复制到主机内存，以供 CPU 使用。

## Serial

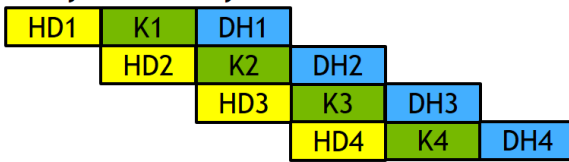


这种串行执行的方式并不是资源的最佳用法。在内存复制过程中，GPU 处于空闲状态，而在 GPU 上执行内核期间，复制引擎处于空闲状态。CUDA 流可用于内存复制和内核执行之间的重叠，以实现进一步优化。下图显示的是双路并发，其中计算结果以较小的批量重新复制到主机，而不是等到整个结果计算完毕。如下显示了类似的 3 路并发，同样将输入数据拆分成更小的数据块。通过将 50,000 张图像推理的任务拆分成较小的批量大小，可以进一步降低 MNIST 示例应用程序的运行时间。

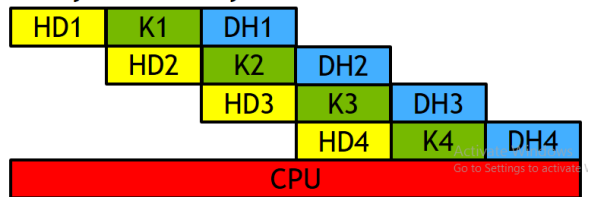
### 2-Way concurrency



### 3-Way concurrency



### 4-Way concurrency



这种优化可以通过使用 CUDA 流异步执行不同的子任务来实现。一个 CUDA 流即为一个设备作业队列。主机将作业置于队列中后立即继续后面的指令。当设备的资源空闲时，它将从各个流中调度作业。CUDA 运算（如内核启动和 CUDA 内存复制）可置于流中。同一流中的运算会顺序执行 (FIFO) 且不会重叠。不同流中的运算无需按顺序执行，故可能会重叠。

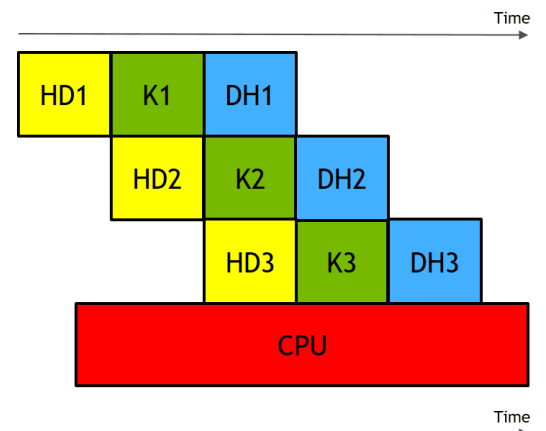
以下示例显示了如何在三个不同的流中实现三种不同的 CUDA 任务，以实现拷贝和计算的重叠。使用流不会缩短实际的计算时间或数据传输时间，它只有助于重叠这些任务。

```
cudaMemcpyAsync ( d_in2, in2, size, H2D, stream2 );
kernel <<< , , , stream1 >>> ( d_in1, d_out1 );
cudaMemcpyAsync ( out3, d_out3, size, D2H, stream3 );

cudaMemcpyAsync ( d_in3, in3, size, H2D, stream3 );
kernel <<< , , , stream2 >>> ( d_in2, d_out2 );
cudaMemcpyAsync ( out1, d_out1, size, D2H, stream1 );

cudaMemcpyAsync ( d_in1, in1, size, H2D, stream1 );
kernel <<< , , , stream3 >>> ( d_in3, d_out3 );
cudaMemcpyAsync ( out2, d_out2, size, D2H, stream2 );
```

asynchronousCPUMethod ( ... )



显然，将 50,000 张图像的推理任务拆分成较小的批量有助于通过 CUDA 流实现进一步的优化。但是，选择子批量大小并不简单。如果将任务拆分成非常少的子任务，则资源利用率很低。但是，如果将其拆分成过多的任务，则启动过多的内核和复制任务会导致运行时间增加。

尝试执行以下代码，使用 20 个 CUDA 流来重叠复制计算。可以更改流的数量，以查看对运行时间产生的影响。通常，运行时间最初随着流数量的增加而减少，但在某个时刻，运行时间会随着流数量的增加而增加。在 MNIST 示例应用程序中，将 50,000 张图像的推理任务拆分成 20 个批次会使运行时间达到最佳。回想一下，以前内核执行需要 3.92 毫秒，加上借助 dGPU 和 INT8 精度的数据传输需要的 5.5 毫秒。而现在，整个内核执行时间几乎与内存传输完全重叠，因而使用 20 个流使总执行时间只需 4.52 毫秒。

```
//待办事项 在 inference.cu 中，为第 108、337 和 484 行取消注释。  
//待办事项 在 inference.cu 中，在第 24 行中将 DEVICE_NUM 设置为 0（选择 dGPU）。  
//待办事项 在 inference.cu 中，为第 842、843、844 行添加注释，并为第 845 行取消注释，以  
使用 inferenceGPU_DP4A_stream()。  
//待办事项 在 inference.cu 中，在第 25 行更改 NUM_STREAMS 值。尝试 2、4、8、16、20、30、  
50、100。  
$ make clean  
$ make  
$ ./inference 50000
```

Number of Streams	Total time(msec) D-GPU with INT8
2	5.64
4	4.97
8	4.66
16	4.56
20	4.52
30	4.54
50	4.93
100	8.07

### 3.2.4 负载均衡

现在，我们将工作负载分配到 iGPU 和 dGPU 上。先把图像按批分配到 iGPU 和 dGPU 上，然后创建两个线程：一个在 iGPU 上的环境和一个在 dGPU 上的环境。在每个环境中，CUDA 流都被用来将数据传输与内核执行重叠。

```
//待办事项 在 inference.cu 中，为第 845 行添加注释，并为第 846 行取消注释，以使用  
inferenceGPU_LB()。  
$ make clean  
$ make  
$ ./inference 50000
```

```
nvidia@tegra-ubuntu:~/CUDA_on_DriveAGX/MNIST_inference$ ./inference 50000
Using load balancing
Using DP4A intrinsics with CUDA streams and running on Device Graphics Device
Using DP4A intrinsics with CUDA streams and running on Device Xavier
Total time is 3.251264 msec with 5 streams
Total time is 3.876352 msec with 15 streams
total matched images 45934 of 50000 with accuracy 0.918680
```

### 3.2.5 总结 – 优化

下表总结了以 CPU 实现 MNIST 推理示例为起点的优化步骤，这些示例的批量大小为 50,000 张图像。正如我们所预期的那样，GPU 加速实现了推理时间的大幅缩减，性能远超 CPU 推理。降低精度、CUDA 流和负载均衡等其他步骤还可以进一步显著缩短执行时间。

Optimization (Progressive)	Total runtime (ms)	Improvement
CPU implementation	1490.71	1x (reference)
GPU acceleration (d-GPU)	26.02	57x
Reduced precision - INT8	9.47	157x
20 CUDA Streams	4.52	329x
Load balancing	3.69	403x

### 参考资料

1. CUDA C 语言编程指南: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html> (<http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>).
2. CUDA C 语言最佳实践指南: <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html> (<http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html>).
3. 适用于 Tegra 的 CUDA: <https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html> (<https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html>).



DEEP  
LEARNING  
INSTITUTE

