2019 届本科生学士学位论文　　学校代码：10269

華東師範大學

East China Normal University

本科生毕业论文

Nvidia 新架构 GPU 为机器学习应用带来的性能提升的研究与评估

Research on performance of ML applications using Nvidia new GPUs

姓　　名：刘子汉

学　　号：10152130243

学　　院：计算机科学与软件工程学院

专　　业：计算机科学与技术

指导教师：钱莹

职　　称：副教授

2019　年　5　月

目　录

Nvidia 新架构 GPU 为机器学习应用带来的性能提升的研究与评估

摘要

本文主要针对 Nvidia 新架构的 GPU（图灵架构）为机器学习应用带来的性能提升进行研究，由于目前实际使用中的应用很难达到 Nvidia 官方宣传的性能提升幅度，故本文将从问题类型、代码结构结合硬件、指令特征对这一现象进行研究，并提出相应的建议。本文主要采用定量方法，通过不同世代的硬件和 SDK 进行横向比较，以及同一世代硬件、SDK 和不同类型应用进行纵向比较；并总结出特征。在研究中较为重要的部分为新硬件中加入的张量核心（Tensor Core）以及对应的线性代数库 CUTLASS，文章将通过混合矩阵运算、矩阵乘法、卷积运算等对其进行评估；其他还涉及了传统的矩阵运算库 CUBLAS、模型优化器 TensorRT 以及最为基本的浮点计算、内存种类等。

根据实验结果，新架构硬件中张量核心对于机器学习应用的类型、计算类型、超参数等条件敏感；要达到期望的性能，输入数据规模、形状、运算占比等方面有较为严苛的需求；在矩阵较为稀疏、输入规模较小时 CUSPARSE 稀疏矩阵库和基于纹理内存的方法能取得更高性能；而计算输入较为规律、符合硬件形状时张量核心能带来显著提升。至于网络推理阶段，TensorRT 在各种情况下均能带来明显的提升。在实际应用中，训练阶段应根据任务特征合理选择硬件、SDK 和内存系统使用；而在推理阶段应利用 Tensor Core 提升吞吐量。

关键词： Tensor Core，TensorRT，通用矩阵乘法，图灵架构

Research on performance of ML applications using Nvidia new GPUs

Abstract

This paper is focusing on the performance improvement in Machine Learning application brougnt by Nvidia’s new architecture (Turing architecture) GPU. Since currently the Machine Learning application actually in used can hardly get as much improvement as mentioned in Nvidia’s official White Paper, so, this paper will research this situation through the type of the application, the structure of the source code combining with feature of the hardware and instructions, thus give corresponding recommendation about coding. This paper uses quantitative methods, doing both horizontal comparation with hardware and SDK of different generations and vertical comparation with different types of problem running on the same generation of hardware and SDK, through which the pattern and feature can be extracted. Among all the new features, the most important is Tensor Core and corresponding library CUTLASS (CUDA Template Linear Algebra Subroutine), this paper evaluate this unit through GEMM, Matrix Multiple, Convolution, etc. Also, traditional matrix library CUBLAS, optimizer TensorRT, Float Point and GRAM are also mentioned.

In the conclusion, Tensor Core in the new architecture GPU is very sensitive to the type of applications, type of calculations, meta parameter, etc., to achieve expected performance, the scale of the data, shape of the data and type of calculations should be well fit to the hardware. Moreover, in some situation including the input matrixs are sparse and the scale of the input data is small, library oriented to sparse matrix (CUSPARSE) and methods based on texture memory will gain much higher performance, and in situation that the input fit the hardware well, the Tensor Core can bring the application a significant improvement in performance. When it comes to the inference stage, TensorRT can bring a significant improvement in almost all the situation.

So, in the training stage of actual application, the usage of hardware, SDK, memory, etc. should be chosen appropriate based on the feature of the applications, and in the inference stage, do not hesitate to use TensorRT!

Keywords： Tensor Core, TensorRT, GEMM, Turing Architecture

一、 引言

（一） 研究背景

近年来，人工智能在全球无论是否是计算机相关行业中，都掀起了一股热潮，尤其是深度学习更是赚足了眼球。作为深度学习应用中计算能力支撑的并行计算硬件与软件更是迅猛发展，而英伟达 (Nvidia) 更是在并行硬件领域独占鳌头。

在 2017 年，英伟达发布了一款基于伏特架构 (Volta) 的面向深度学习的 GPU，Tesla V100[1]，其中搭载了一些实验性的新技术；之后在 2018 年第三季度，英伟达发布了新一代图灵架构 (Turing)，在该架构中，正式引入了许多革命性的新技术，同时也对原有技术做了很大的改进。有面向深度神经网络应用的张量核心 (Tensor Core)[2]，能够大幅加速在神经网络训练、推理中的混合精度矩阵计算，该核心最先实验性地搭载于 Tesla V100，在图灵架构中上至面向深度学习推理的 Tesla T4，下至面向游戏玩家的 RTX 2080Ti 都搭载了这款核心；用于更高效搭建分布式计算平台的第二代端对端互联总线 (NV Link 2.0)[3]，相对于原有的 QPI 等总线，该总线能够直接互连 GPU，且提供远高于原先 SLI 技术所能提供的带宽；以及针对游戏玩家推出的实时光线追踪技术 (RTX)，该技术不在本文的讨论范围内。同时，由于 GPU 中 CUDA 计算单元架构包括流多处理器，纹理单元等的优化，在性能大幅提升的同时，热设计功耗 (TDP) 仍然维持在了上一代硬件的 250W。

在并行软件方面，与硬件一起，英伟达将其面向并行程序开发的 SDK CUDA 的版本更新到了 10.0，在游戏应用、通用计算方面针对新架构的特性进行了优化；同时发布了基于 CUDA 10.0 的进行线性代数计算的模板库 CUTLASS(CUDA Template Linear Algebra Subroutines)[4] 以利用其张量核心进行高效的代数运算。

然而，官方文档给出的性能提升仅仅包括单一模块的理论性能提升，如传统 CUDA 核心的浮点数值计算的理论峰值，新加入的张量核心的混合矩阵计算 (GEMM) 的理论峰值；NV Link 2.0 的理论峰值带宽等。在实际使用中，用户反映在网络推理方面以及基于支持新硬件的相关框架开发的机器学习应用中，提升并没有官方白皮书给出的的 9 倍之多 [5]，且同类型不同规模应用的性能提升幅度并不一致，性能提升对神经网络中参数数量、网络层数等因素较为敏感。实际上，官方给出的文档中的提升也仅为绝对计算性能的提升，没有考虑应用类型、平台构建等条件。且目前关于新架构 GPU 的研究主要集中在大型计算节点的扩展效率 [6]，基于 GPGPU-Sim 模拟的性能考察等 [7]，这些研究或是停留在表征性能层面、没有深入到代码或是中间代码层面；或是使用模拟技术、在 PC 机上进行模拟，尽管目前对于硬件的模拟运行的匹配度能够达到较高的水准，但是仍然有一定偏差，目前 GPGPU-Sim 的稳定版支持的最高的 CUDA SDK 版本为 4.0，开发版本支持的最高的 CUDA SDK 版本为 9.2。本文将直接针对真实的，单一的，图灵架构的 GPU：RTX 2080Ti 进行深入，结合版本最新的 CUDA SDK 10.0 以及对应的软件库包含 CUTLASS，CUBLAS 等，从架构、PTX 中间代码层面、SASS 机器码层面对 GPU 在使用 GPU 加速的机器学习应用中的性能以及性能提升进行研究和评估；根据研究和评估结果以及分析得到的原因对现有 CUDA 代码进行优化；且将结合目前对于新老架构的对比研究 [8]，将新架构与麦克斯韦架构的 GPU：GTX Titan X 与帕斯卡架构的 GPU：GTX 1080Ti 进行横向对比，从实际替换成本、环境搭建成本、维护成本、性能/功耗比等角度对新架构进行评估与进一步设想。

在最近刚结束的 GTC 2019 会议中，Nvidia 发布了若干面向机器学习的硬件、软件。包括专为张量计算

设计的 Turing Tensor Core GPU，嵌入式平台的 Jetson Nano[9]，将机器学习相关计算库整合起来的 CUDA X[10]，这些都将在后文提到，但由于这些本质上都基于目前的 Turing 架构，故不会单独进行详细地说明。

（二） 我们的工作

近年来，机器学习尤其是深度学习发展迅猛，各种方便程序员搭建模型的框架层出不穷。考虑到机器学习应用的计算量要求日益攀升，这些框架都陆续推出了基于 GPU 的版本。为方便程序员搭建模型，框架本身对硬件的操作进行了抽象。然而，正是因为这一层抽象，忽略了许多硬件层面的细节，使得框架无法完全利用硬件的性能。这也导致了许多用户反映在实际应用中，新架构的性能提升并没有官方给出的文档数值、硬件参数 (包括流处理器、纹理/光栅单元)、甚至价格上涨幅度那么多。

为了尽可能在实际应用场景中提升硬件性能的利用率，本文将从如下层面对基于 CUDA 以及相关框架的机器学习应用进行研究与评估；挖掘理论与实际不符的原因；并做出适当的修改和建议。

- CUDA 源码
- CUDA 源码编译出的 PTX 中间代码
- 基于 CUDA 的框架的应用源码

因 CUDA SDK 10.0 发布不久，目前许多框架还未对该 SDK 进行相关优化；一些既存的 CUDA 应用仍是基于 CUDA SDK 9 甚至 CUDA SDK 8 进行编译的。所以本文将结合对于上述三个层面的分析结果，结合新硬件、新架构、新 SDK 的特征，在源码层面进行调整并给出一些编写相关程序时的建议；力图尽可能多地发掘新硬件、新架构的潜力。

（三） 本文的组织结构

本文在第 2 章中介绍了该研究的背景和相关的工作。首先介绍了基于 GPU 的机器学习应用与 CUDA 的相关背景知识。由介绍基于 GPU 的机器学习应用引出 CUDA 的相关介绍，包括 CUDA 应用的编程模型、编译过程、调用/执行方式。然后介绍了目前对于评估、模拟 GPU，尤其是 CUDA 应用性能开展的相关工作；由超微半导体 (AMD) 开发的 GPU 也具有通用计算功能，然而目前市面上还没有基于 AMD 开发的 GPU 的相关 SDK 或框架，故本文不做讨论。最后介绍了基于 CUDA 的可执行程序的汇编代码结构和使用 CUDA 源码编译得到的 PTX 中间代码的结构，供之后的分析使用。

本文在第 3 章中首先简要介绍实验动机、实验步骤以及实验结果。然后介绍了实验所需的工具、环境以及搭建方式等。接着详细介绍了我们的主要工作，包括基于单一功能、测试用的应用的 Benchmark、基于 CUDA 源码的机器学习应用的研究过程、针对汇编代码与 PTX 中间代码的研究过程、针对基于 CUDA 的相关框架的机器学习应用的研究过程以及根据分析得出的结果给出的修改、建议等。最后给出了各项实验的结果和对比，并进一步分析原因。

本文最后在第 4 章进行总结，并给出之后改进与深入工作的设想和预期。

二、 背景及相关工作

（一） 基于 GPU 的机器学习应用与 CUDA

1. 基于 GPU 的机器学习应用

随着当今机器学习，尤其是深度学习应用中数据量、网络结构复杂度的增长，该类应用对于硬件计算能力的要求也迅速增长。而在这类应用中，有许多密集的计算互相之间是没有数据/控制依赖的，也就是可以并行执行的；比如神经网络前向、反向传播中的权重矩阵计算，这些权重在同一轮计算中不存在耦合性；随机森林 (Random Forest) 中不同分类器的训练，这一特征可以利用到 GPU 处理中流这一特征；一系列聚类算法，包括 DBSCAN、K-Means 等；而图形处理单元 (GPU) 的设计初衷正是大规模并行计算，也因为 GPU 的计算能力，深度学习自上世纪末至今迅猛发展，同时 GPU 的运算性能以及相应的软件的发展也非常迅速。目前，GPU 更多代表了通用处理单元 (General Purpose)。

当然，GPU 上的编程模型与 CPU 上的模型有较大差别，为了方便程序员搭建模型，目前市面上的许多框架包括 TensorFlow, PyTorch, PyChain 等都更新了对 GPU 的支持。然而，这些框架方便了程序员的程序编写，抽象了底层硬件的细节，比如在 CUDA 中，线程块、线程束的调度以及相应寄存器文件的分配会对程序性能造成极大影响，然而这些特征都被框架抽象这就导致了硬件性能无法得到完全的发挥。且目前大部分框架都是基于老架构与老 SDK 编译，没有对新架构与新 SDK 做出优化。本文的目的也是在于挖掘出新架构的硬件以及对应的新的 SDK 中的代码翻译、指令执行等部分的特征以及相较老架构和老 SDK 的变化；根据分析得出的结论修改已有 GPU 程序的源码，尝试修改某些框架的源码，并给出实际的修改、编程时的建议。

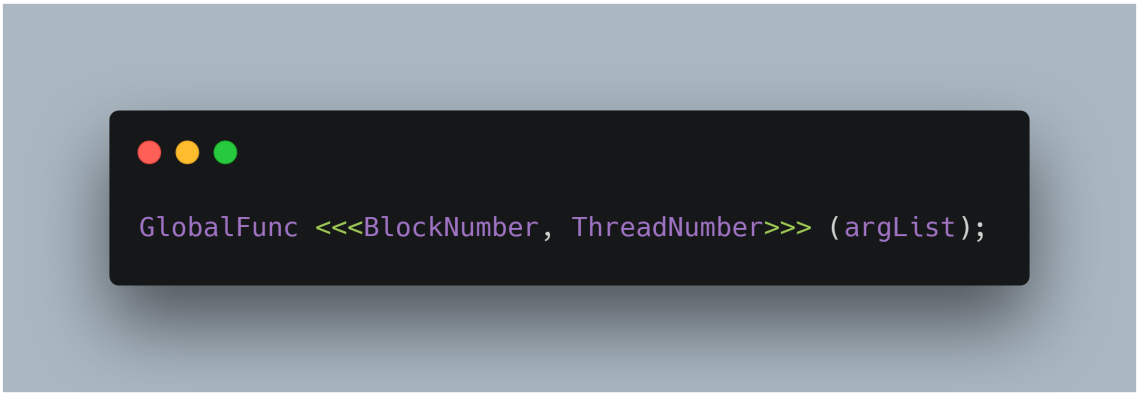
2. CUDA

CUDA (Compute Unified Device Architecture) 是由英伟达 (Nvidia) 针对图形处理单元开发的并行计算平台及对应的编程模型。在编写 CUDA 程序时，程序员通过在一些较为热门的语言包括 C/C++、Python、MATLAB、Fortran 中以关键字的形式加入扩展来描述并行行为 [11]。下面将介绍 CUDA 程序的编程模型、编译过程与调用/执行方式。

（1） 编程模型 在介绍编程模型前，需要简要介绍一下 Nvidia GPU 的硬件结构。自顶向下的结构为：一块 GPU 芯片有若干流多处理器单元 (Stream Multiprocessor，SM)，这些流多处理器单元被一个线程块调度器管理，所有流多处理器单元通过全局内存总线和常量内存总线经过 L2 缓存共享全局内存与常量内存，这部分内存自费米架构以来有 GDDR4、GDDR5、GDDR5X、GDDR6X、HBM、HBM2 等类型。每个流多处理器单元中有若干个流处理器 (Stream Processor，SP)，因 CUDA 程序为 SIMT(单指令多线程) 并行方式，所以这些流处理器共享一个指令缓存，每个流处理器拥有自己的线程束调度器与寄存器文件；流处理器中包含若干种执行单元，有浮点单元，整数单元，在新架构中还加入了张量单元 (Tensor Core)，在 RTX 2080TI 上具体的参数为：一个 SM 包含 64 个单精度浮点算术单元，32 个双精度浮点算术单元，64 个 32 位整形算术单元，8 个混合精度张量单元，4 个线程束调度器和 16 个特殊功能单元；所有流处理器通过显存纵横矩阵 (CrossBar) 访问共享内存，或被称为 L1 缓存 [12]，所有流多处理器共享 L2 缓存。

```
void __global__ hostCallDeviceFunction(...) {
    // __global__ means this function is called by host(CPU) and run by device(GPU).
}
int __device__ deviceCallDeviceFunction(...) {
    // __device__ means this function is called by device(GPU) and run by device(GPU).
}
int __host__ hostFunction(...) {
    // __host__ means this function is called by host(CPU) and run by host(CPU).
}
```

图 1: CUDA 中的三种函数



```
GlobalFunc <<<BlockNumber, ThreadNumber>>> (argList);
```

图 2: __global__ 函数调用方式

因本文的工作主要基于 C/C++ 完成，故只介绍 CUDA C/C++ 的编程模型。在保证相关环境配置完成后，向 C/C++ 源码中加入 CUDA 相关工具是十分方便的。首先需要确定哪些任务需要在 CPU 上执行，哪些在目标硬件 GPU) 上执行；选择的标准一般是考察其数据/控制依赖，依赖性小、计算密集的可以考虑在 GPU 上执行。确定完毕后编写 CPU 端和 GPU 端的函数，有如图3中所示的三种函数。

以上代码段加粗部分为 CUDA 关键字，对于函数有三种修饰：__global__, __device__, __host__. 分别代表被 CPU 调用运行于 GPU、被 GPU 调用运行于 GPU 和被 CPU 调用运行于 CPU 的函数。其中被 CPU 调用运行于 GPU 的函数只能拥有 void 返回值，且所有运行于 GPU 的函数都不支持可变参数列表 [13]。__device__ 和 __host__ 关键字修饰的函数的调用方式与传统函数别无二致，__global__ 关键字修饰的函数调用方式如图2所示。BlockNumber 和 ThreadNumber 分别代表要启动的线程块数目和每个线程块中线程的数目，这一部分取值对程序性能影响较大，之后会详细说明。

之前提到了 GPU 端的缓存，CUDA 程序的另一个重点是存储系统的管理。传统 CPU 编程模型中，寄存器、缓存等资源都是由 CPU 自行管理，而不开放给程序员。其原因在于 CPU 拥有的寄存器、缓存资源较为紧缺，为提高指令级并行能力，需要采用多队列乱序发射与寄存器重命名等技术。相对得，GPU 有较为充足的物理寄存器、缓存资源，程序员也对这部分资源掌握有一定的控制权 [14]。CUDA 中的存储设备如表所示。

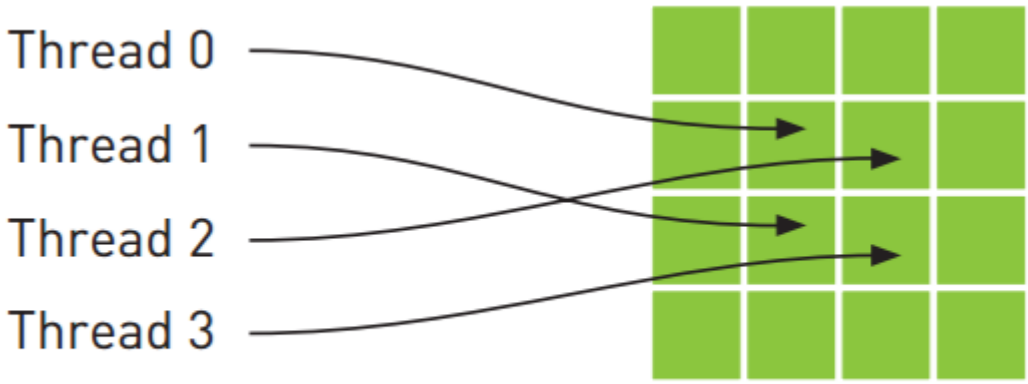


Figure 7.1 A mapping of threads into a two-dimensional region of memory

图 3: 纹理内存访存方式

项目	大小	延迟 (时钟周期)	访问权限
寄存器文件	8KB-64KB/SM	10^0	GPU 端
共享内存 (L1,L2)	16KB-128KB/SM	10^1	GPU 端
常量内存	N/A	N/A	N/A
纹理内存	N/A	N/A	N/A
全局内存	-GB	10^2	CPU 端/GPU 端

需要注意的是，常量内存与纹理内存都是全局内存的一种虚拟地址形式。和常量内存一样，纹理内存也是一种只读内存；但是在缓存加载的行为方面，常量内存与传统方式一样，加载所访问数据单元的所在行的一部分单元，而纹理内存则加载所访问数据周围一个范围内的单元 [15]。这样做的原因是在 GPU 进行图形运算时，处理某一像素点需要用到周围一个范围内所有像素点的信息比如进行抗锯齿作业时，而非只有一行，如图3所示。采用这种结构能改善某些访问模式情况下程序的性能。

这些内存的使用方式如图4所示。寄存器和共享内存的使用方式很好理解，关于常量内存和纹理内存，由于他们是全局内存中的虚拟地址，故声明常量内存的语句也就是用了全局内存；纹理内存的使用则需要借助相应 API。而一般的全局内存的读写权限同时开放给 CPU 和 GPU，故需要使用特定的 API 进行访问。

（2） 编译方式 CUDA 程序的编译较为简单，只需使用 *nvcc* 对源文件进行编译生成可执行文件。首先将从 C/C++/CUDA 文件编译生成 PTX 中间代码，再从 PTX 中间代码生成 SASS 机器代码。本实验中由于需要观察、研究具体 GPU 代码的生成方式、特征，故在编译时加入 $--keep$ 保留编译产生的 PTX 中间代码文件。而 SASS 机器代码则在 NVIDIA 的支持下通过捕捉 CUDA 应用的指令流获得。

（3） 运行模式 关于如何从 CPU 端 (host) 启动 GPU 端 (device) 的函数将在下一节通过 CPU 端应用程序的反汇编代码详细描述，这一节仅介绍 GPU 相关的部分。

根据弗林分类法 [16]，计算机系统可以分为 SIMD, MIMD, SISD, MISD 等类型，目前多核心 CPU 系

```
__constant__ static const u32   dataWithGlobalMemory_Constant;

void __global__ MemoryDemoGlobal(int *arrayOnDevice) {
                u32     dataWithRegister;
        __shared__  u32     dataWithSharedMemory;
        // Do Something
}

void __host__   MemoryDemoHost(int *arrayOnHost) {
        int *arrayOnDevice;
        int  arraySize;
        cudaMalloc((void**)&arrayOnDevice, arraySize * sizeof(int));
        cudaMemcpy(arrayOnDevice, arrayOnHost, arraySize * sizeof(int), cudaMemcpyHostToDevice);
        MemoryDemoGlobal <<<blockNum, threadNum >>> (arrayOnDevice);
        // Do Something
        cudaMemcpy(arrayOnHost, arrayOnDevice, arraySize * sizeof(int), cudaMemcpyDeviceToHost);
        // Now array with data proceed by the GPU is accessable
}
```

图 4: 不同存储系统使用方式

统就是 MIMD 系统。而 NVIDIA 的 GPU 系统被称为 SIMT(单指令多线程)，与 SIMD(单指令多数据) 有所不同。在这种模型中，一条指令并非仅仅代表一个固定的功能，而是代表执行这一指令的类型、使用的管道/流水线。线程需要执行的具体操作需要编写相关内核代码。直观的特征便是再 PTX 中间代码和 SASS 机器代码中，所有的逻辑操作指令都是 $lop/lop3$，通过后缀 $.and/.or, .sync/.async$ 指明具体运算和调度特征。所以，在 SIMT 模型中，内核程序读入统一的数据，程序代码根据需要进行不同操作；实际调度时不同操作通过重复指令流按顺序发射，只不过运算单元会屏蔽无关线程。

首先需要介绍一下计算能力，此处计算能力指的是 GPU 中流多处理器 (SM) 支持的运算的等级，分为 Major 和 Minor 等级，可以等价为流多处理器的代号，所支持的运算不同。其中 Major 代号代表架构的更新，这也会带来许多新的硬件支持的运算，而 Minor 代号则代表同一架构下不同定位的流多处理器产品。如伏特架构的计算能力为 7.2，图灵架构的计算能力为 7.5，Major 代号一样就代表这两种架构其实并无太大修改，而 Minor 代号则代表伏特架构中流多处理器的类型是 Heavy，图灵架构中流多处理器的类型是 Lite。Lite 和 Heavy 一般用于区分消费级/工作站级 GPU，分别对应 GeForce 和 Tesla 代号。

基于 GPU 的应用与传统的基于 CPU 的应用在运行方式上有较大差别，主要有以下几点。

- GPU 中由大量的物理寄存器，达几十 几百 KB，且都能在 1 个时钟周期内访问，而 CPU 中物理寄存器资源极为有限。故在进行上下文切换时，GPU 只需更改寄存器文件指针来切换，而 CPU 需要使用堆栈保存上下文。

- CPU 仅仅支持数十个硬件线程，而 GPU 则支持数千个硬件线程。在 GPU 上开启过少的硬件线程会极大降低硬件使用率，进而导致性能降低。具体开启线程数量取决于硬件 SM 最大并发线程数、最大并发线程束数和最大并发块数等。表3显示了在不同计算能力上开启不同数量的线程时设备的利用率以及所能开启包含该数量线程的线程块的数量。

	1.0	1.2	2.0	2.1	3.0
64	67%, 8	50%, 8	33%, 8	33%, 8	50%, 16
96	100%, 8	75%, 8	50%, 8	50%, 8	75%, 12
128	100%, 6	100%, 8	67%, 8	67%, 8	100%, 10
256	100%, 3	100%, 4	100%, 6	100%, 6	100%, 8
512	67%, 1	100%, 2	100%, 2	100%, 3	100%, 4
1024	N/A, N/A	N/A, 1	67%, 1	67%, 1	100%, 2

可见，随着计算能力的增长，一个流多处理器上所能容纳的线程束月俩月多。在充分利用寄存器文件和共享内存 (L1 缓存) 的情况下开启线程数越多，设备利用率越高。然而过多的线程会导致资源紧缺，在实际使用中应当根据硬件参数，问题规模做出调整。

在 CUDA 程序中，32 个线程被组织成一个线程束 (warp)，作为基本的调度单元，具有各自的物理寄存器，也就是说线程束中 32 个线程一般情况下都会执行相同指令流，为 SIMD 模式。在目前的图灵架构中，线程束仍然是同步的单位，即线程束之间可以保证同步，线程束内部线程无法保证同步。在下一代安培架构 (Ampere) 则引入 $Arrive - Wait$ 模式以实现线程级别同步，以提高 CUDA 程序的灵活性。

若干个线程束被组织为一个线程块 (block)，在下一代安培架构中将添加一个线程束组的层级 (warp group)，由四个线程束组成，但该层级仅为大规模通用矩阵乘法运算所用 (Ultra MMA)，且本代架构还未应用，这里不做讨论。线程块中的线程束可以通过 __syncthread() 进行同步，线程块中的线程能够访问共享内存进行数据交换。一个线程块被分配在一个流多处理器上执行，一个流多处理器能分配多个线程块。

若干个线程块被组织成一个线程网格 (grid)，线程网格可被看作一个分配给 GPU 的任务。

表3详细说明了 CUDA 应用中不同不同粒度的线程组织形式。

粒度	调度者	分配给
warp	流多处理器内部调度器	流处理器 (Stream Processor)
block(CTA)	TPC 调度器 (MPC)	流多处理器 (Stream Multi-processor)
grid	GPC 调度器 (GPM)	TPC(Texture Processing Cluster)∗
kernel	CPU, PCIe	GPC(Graphics Processing Cluster)

∗ 在本世代图灵架构及以前，TPC 与 SM 可以等价，因为一个 TPC 上仅包含一个 SM，然而自下一代安培架构开始，一个 TPC 中将会有若干个 SM。虽然本文研究的图灵架构的硬件在逻辑上 SM 与 TPC 等价，但在硬件上还是会做区分，故在表中详细写出。

（二） 目前展开的工作

（三） CUDA 应用的汇编代码与 PTX 中间代码的结构

1. CUDA 应用汇编代码结构

2. PTX 中间代码结构

三、 评估 NVIDIA 新架构 GPU 的机器学习应用性能

（一） 实验动机、过程与结果

（二） 实验工具与环境

（三） 实验详细过程

1. 基于测试样例的 Benchmark

为了为接下来的实验设定基准，这一步先使用用途单一的测试样例测试绝对性能以及相应的提升，因不同架构的硬件各项参数 (包括流处理器数量、显存容量等) 不尽相同，所以直接对比不同架构硬件的性能是没有意义的，这里选择对比不同架构硬件在不同 SDK 下性能提升的比例。此处选用了 CUDA 10.0，CUDA 9.2，CUDA 9.0 三种 SDK，同时选用 9.2 与 9.0 的原因是因为 9.2 版本是为了图灵架构的 GPU Tesla V100 发布的 [17]，也在本文的研究范围内。

因为本文主要讨论新架构 GPU 在机器学习应用中带来的性能提升，故选用的评测样例大部分都与机器学习应用相关；主要从以下角度进行评估：通用矩阵乘法 (GEMM, General Matrix Multiply)、矩阵乘法运算性能、卷积运算性能、神经网络推理性能以及结合框架的综合性能。在评估这些性能时也会包含单/双精度浮点计算性能。

（1） 通用矩阵乘法 (GEMM, General Matrix Multiply) 待评测项目中最为重要的是通用矩阵乘法 (GEMM, General Matrix Multiply)，新架构对该运算进行了硬件、指令级别的优化，是与老架构最鲜明的区别所在。其混合体现在：运算中同时有加法和乘法，且精度同时涉及半精度浮点、单精度浮点和 8 位整数。与矩阵乘法相比，通用矩阵乘法被定义为：

$$C \leftarrow \alpha AB + \beta C$$

若将 β 置为 0，则该运算变为矩阵乘法运算。通用矩阵乘法这一运算在神经网络训练、推理中十分常见，根据官方文档，目前 Tensor Core 仅能用在 CNN/RNN 等特定结构的神经网络上，且只能用于前馈和反馈两部分。这个范围看起来很宅，然而在深度学习中占到了非常高的比重。式中操作数分别代表输入、权重和偏置，下文将简写为矩阵乘加。NVIDIA 在新的伏特架构与图灵架构中加入的张量核心 (Tensor Core) 正是专门加速这种运算的硬件；对应新硬件，在 PTX 中间代码层面新增了 $wmma$ 指令，在 SASS 机器代码层面则增加了对应的 $hmma$ 指令。该指令的作用为以指定的精度计算两个输入矩阵的乘积并将计算结

果累加到指定的精度的矩阵中；指令进行的具体操作、操作数的数据精度、形状、存储方式 (行主元素/列主元素) 等通过指令中特定的字段指定。

在底层的实现中，张量核心以 4×4 的矩阵作为最小的计算单元，被称为 $tile$，任何输入都会被划分为 $tile$ 进行分块运算。在伏特架构以前 (Volta) 的帕斯卡架构 (Pascal)，一次 4×4 矩阵乘加需要首先调用 16 次整数点积运算 (若硬件支持 $idp/idp4a$ 指令)，再将结果累加到乘加矩阵中。而使用 Tensor Core 则仅通过 $hmma$ 指令直接完成。根据官方文档给出的描述，这种机制能使伏特架构相比帕斯卡架构再 FP16, INT8, INT4 京都中分别提供 8 倍、16 倍、32 倍的吞吐量提升。实际测试中，Tesla V100 再 FP16 精度下的 $m = 2048, k = 2048, n = 2048$ 规模的矩阵乘加中比 Tesla P100 块 9.3 倍 [5]，这也是上文提到的官方宣称的 9 倍。本节将在各种规模、精度、形状的情况下考察 Tensor Core 实际能够带来的性能提升并探究相应原因。

1） **实验结果** 根据开发者社区的反映，新架构硬件性能的差别主要体现在问题规模、问题类型等方面 (张量维度、形状，训练/推理任务等)，而 NVIDIA 官方仅给出一种规模的结果，所以本节使用了自行编写的一系列测试用例，辅以深度学习测试套件 DeepBench，在开启和关闭新架构中张量核心的情况下进行测试。实验性能使用 TFlops/s 统计，方法为简单的运算数除以运算时间，运算时间的统计采用 CUDA 内置的 $cudaEvent$ 记录。

2） 结果分析

（2） 矩阵乘法运算

1） 实验结果

2） 结果分析

（3） 卷积运算

1） 实验结果

2） 结果分析

（4） 神经网络推理

1） 实验结果

2） 结果分析

2. 基于 CUDA 源码的应用

（1） 卷积神经网络

1） 实验结果

2） 结果分析

（2） 并行支持向量机

1） 实验结果

2） 结果分析

3. 基于 TensorFlow 框架的应用

1） 实验结果

2） 结果分析

四、 表与图

这节用来展示表格与图片的插入。

（一） 表格

本来 LaTeX 里表格的变化是非常多的，但鉴于学校要求用三线式，问题反而简单了。以下是一个例子：如果你有使用更复杂的表格的需求，请自行查资料完成。

表 1: 示例表格
Example Table

◦◦	◦◦	◦◦	◦◦	◦◦	◦◦	◦◦
◦◦	◦◦	◦◦	◦◦	◦◦	◦◦	◦◦
◦◦	◦◦	◦◦	◦◦	◦◦	◦◦	◦◦
◦◦	◦◦	◦◦	◦◦	◦◦	◦◦	◦◦
◦◦	◦◦	◦◦	◦◦	◦◦	◦◦	◦◦
◦◦	◦◦	◦◦	◦◦	◦◦	◦◦	◦◦

（二） 插图

由于这份模板不考虑多栏排版，所以格式要求中所述的半栏图大小要求我们不作演示。以下是一个通栏图的演示：

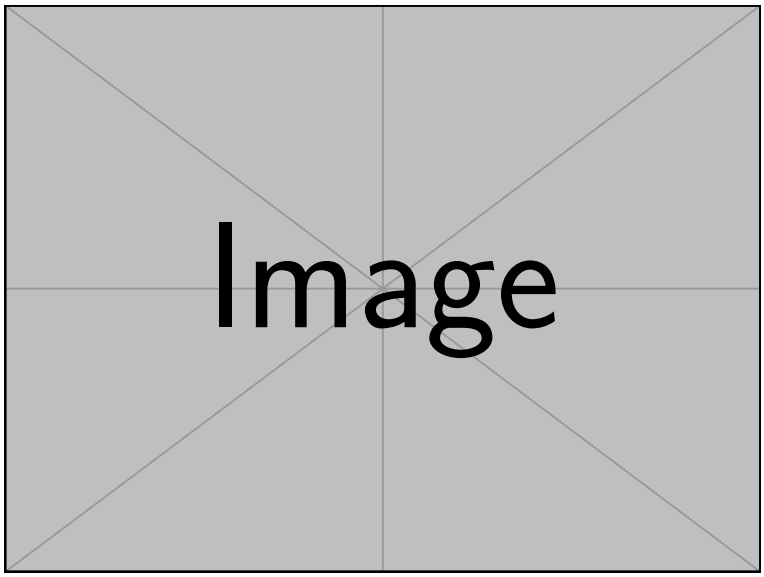


图 5: 图片测试（最小宽度）
Image test (Minimal width)

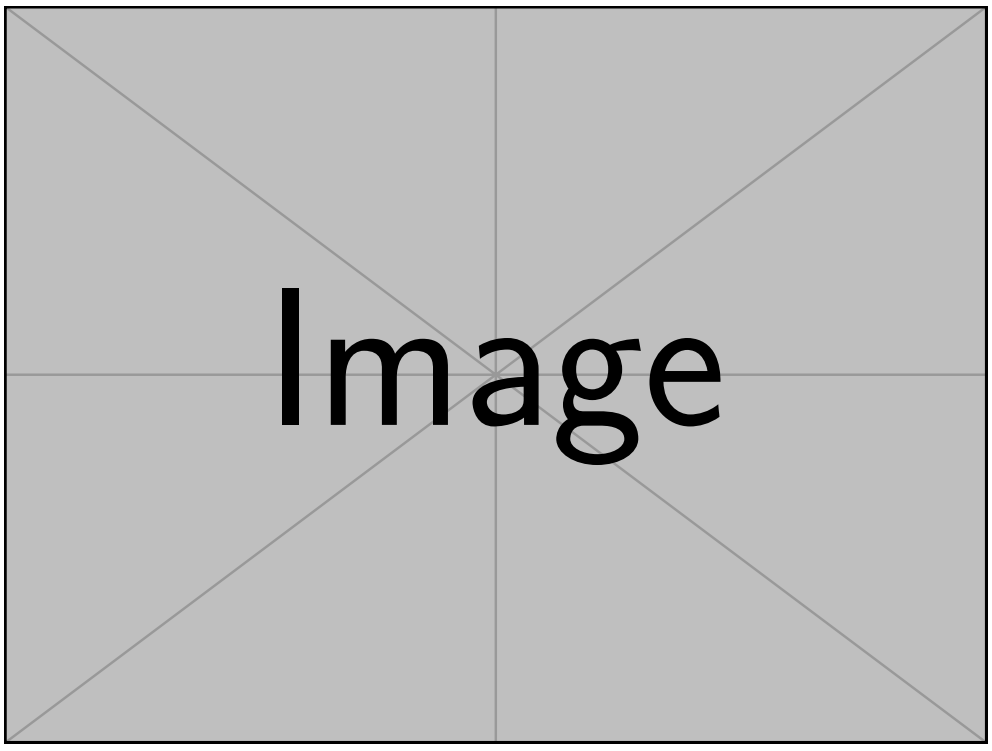


图 6: 图片测试（最大宽度）
Image test (Maximal width)

注意：这里为了减少图片上下的空白，使用了 float 宏包。

五、 注释与引用

这节用来展示注释与引用。

（一） 注释——脚注与尾注

1. 脚注

这里是脚注测试[1]这里是脚注测试这里是脚注测试这里是脚注测试[2]这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试[3]这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试这里是脚注测试

注意！正如这份演示中所出现的情况，若该页（也就是本文档中的前一页）剩余空间不大，不足以显示足够多的文档与脚注，那么该段文字就会被移至下一页而留下空白。目前我们尚未找到解决的方法，所以如果遇到了这个问题，请修改排版，以留下足够大的空间。

2. 尾注

这里是尾注测试[尾注 1] 这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试[尾注 2] 这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试[尾注 3] 这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试这里是尾注测试

注意！endnotes 宏包并不支持 hyperref，也就是无法通过点击文中尾注标号以跳转到尾注。当然，这在打印出来的文档中并不会造成任何影响。

提示：尾注出现在全文最后。为了区分脚注与尾注的编号，我们在尾注编号前加上了“尾注”二字。

（二） 交叉引用

本模板使用 cleveref 宏包来进行交叉引用。使用的指令为 \cref{label}。例子如下：

由**??**我们可以知道 XXXXXXXX。

由**??**我们可以知道 XXXXXXXX。

请注意，label 是需要手工设置的，一般将 label 放在你需要引用的环境内即可（具体可见 SectionB.tex）。

（三） 文献引用的演示

本模板使用 biblatex 进行文献管理，这是一套相对较新的系统。另外，使用了 hushidong 制作的符合 gb7714-2015 标准的 biblatex 样式。在此对他的工作表示感谢，要完成这样的样式非常不容易。本模板中 gb7714-2015.bbx 与 gb7714-2015.cbx 即为他的作品，在这里打包发布以便使用。

[1] 1111111111

[2] 2222222222

[3] 3333333333

默认的 bib 文件位于 /reference/thesis-ref.bib，内容是由 Wang Tianshu 制作，在此仅作演示之用。关于 bib 文件的编写与管理请自行查找相关教程。

下方的演示已经给出了正文中引用文献的基本方法，这与传统的 cite 命令是类似的。如有更多需求，请至`https://github.com/hushidong/biblatex-gb7714-2015`查找相关资料。

本模板使用 parencite 而不是 cite 命令，因为这样能与脚注所产生编号进行区分。当然，如果你没有脚注或尾注，那么 cite 命令也是推荐使用的。

尾注

1. 伴随着互联网的发展以及新的网络应用的出现, 互联网用户由单纯的“读”网页, 向“读、写”网页, 共同建设互联网发展, 由此网上产生了大量带有用户主观感情的数据, 从这些带...

2. 尾注测试 2

3. 尾注测试 3

参考文献

[1] NVIDIA. NVIDIA TESLA V100 TENSOR CORE GPU[A]. 2019.

[2] NVIDIA. NVIDIA TENSOR CORES, The Next Generation of Deep Learning[A]. 2019.

[3] NVIDIA. NVLINK FABRIC[A]. Advancing Multi-GPU Processing. 2019.

[4] KERR A, MERRILL D, DEMOUTH J, et al. CUTLASS: Fast Linear Algebra in CUDA C++[A]. 2019.

[5] NVIDIA. NVIDIA TESLA V100 GPU ARCHITECTURE[R]. NVIDIA Corp., 2017: 14–15.

[6] KURTH T, TREICHLER S, ROMERO J, et al. Exascale Deep Learning for Climate Analytics[C]// Super Computing Conference. [S.l.]: [s.n.], 2018.

[7] RAIHAN M A, GOLI N, AAMODT T M. Modeling Deep Learning Accelerator Enabled GPUs[J/OL]. CoRR, 2018, abs/1811.08309. http://arxiv.org/abs/1811.08309.

[8] MIKI Y. Gravitational octree code performance evaluation on Volta GPU[J/OL]. CoRR, 2018, abs/1811.02761. http://arxiv.org/abs/1811.02761.

[9] NVIDIA. JETSON NANO, Bringing the Power of Modern AI to Millions of Devices[A]. 2019.

[10] NVIDIA. NVIDIA CUDA-X AI, NVIDIA GPU-Acceleration Libraries for Data Science and AI[A]. 2019.

[11] NVIDIA. CUDA Zone[A]. 2019.

[12] KHAIRY M, JAIN A, AAMODT T M, et al. Exploring Modern GPU Memory System Design Challenges through Accurate Modeling[J/OL]. CoRR, 2018, abs/1810.07269. http://arxiv.org/abs/1810.07269.

[13] HARRIS M. An Even Easier Introduction to CUDA[A]. 2017.

[14] COOK S. CUDA Programming: A Developer's Guide to Parallel Computing with GPUs[M]. [S.l.]: Morgan Kaufmann, 2012: 99–102.

[15] HAKURA Z S, GUPTA A. The Design and Analysis of a Cache Architecture for Texture Mapping[C]// Proceedings of the 24th International Symposium on Computer Architecture, Denver, Colorado, USA, June 2-4, 1997. [S.l.]: [s.n.], 1997: 108–120.

[16] FLYNN M J. Some Computer Organizations and Their Effectiveness[J]. IEEE Trans. Computers, 1972, 21(9): 948–960. DOI: 10.1109/TC.1972.5009071.

[17] NVIDIA. CUDA Toolkit Documentation v9.2.148[A]. 2018.

附录 一、 实验数据

233333333333333333333333333333333333333333

附录 二、 调查结果

233333333333333333333333333333333333333333

致谢

感谢天，感谢地，感谢阳光照耀了大地