



华东师范大学
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

摘要	I
Abstract	II
第一章 引言	1
1.1 研究背景	1
1.2 目前展开的工作	2
1.3 我们的工作	3
1.4 本文的组织结构	3
第二章 背景及相关工作	5
2.1 基于 GPU 的机器学习应用	5
2.2 NVIDIA GPU 硬件结构与硬件机制	5
2.2.1 GPU 芯片总体结构	5
2.2.2 流多处理器单元 (SM)	5
2.2.3 存储模型与管理	6
2.2.4 纹理内存 (Texture Memory)	6
2.2.5 硬件机制	7
2.3 伏特/图灵架构新硬件	9
2.3.1 在流多处理器单元层面的差异	9
2.3.2 张量核心 (Tensor Core)	9
2.4 软件	11
2.4.1 机器学习框架 (Tensor Flow)	11
2.4.2 CUDA	11
2.4.3 硬件代码 (SASS) 与中间代码 (PTX)	12
2.4.4 相关工具	12
第三章 评估 NVIDIA 新架构 GPU 的机器学习应用性能	15
3.1 实验工具与环境	15
3.1.1 实验环境	15
3.1.2 实验工具	15
3.2 实验详细过程	16
3.2.1 基于测试样例的 Benchmark	16
3.2.2 基于 CUDA 源码的应用	25
3.2.3 基于 TensorFlow 框架的应用	26
第四章 总结与展望	27
参考文献	29
致谢	32

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：张量核心，纹理内存，通用矩阵乘法，图灵架构

Abstract

This paper is focusing on the performance improvement in Machine Learning application brought 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, Texture Memory, GEMM, Turing Architecture

第一章 引言

1.1 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。

1.2 目前展开的工作

本文的研究同时涉及硬件和软件：Nvidia 新架构的 GPU 与使用 GPU 加速计算的机器学习应用，注意这里的机器学习应用并不只是现在大行其道的深度学习，还包括传统的基于概率模型的方法等。

在近年来深度学习迅猛发展之前，关于使用 GPU 并行化机器学习算法的研究就已有很多，甚至可以说正是基于 GPU 的强大并行计算能力，深度学习才能发展如此迅速。早在 2005 年，D. Steinkraus 等人便对在 GPU 上实现机器学习算法进行了研究，当时实现的算法是 OCR，如今 OCR 能够非常快速、方便地通过各种框架、语言实现。然而这一研究奠定了使用 GPU 加速机器学习应用的基础 [11]。之后的几年中，除去 OCR 外，基于 GPU 的各种并行机器学习算法包括 kNN[12]，支持向量机 [13] 都慢慢成熟。由于贝叶斯网络的精确推理是个 NP 难的问题，其性能受限于硬件水平，然而根据 Md Vasimuddin 等人的研究 [14]，通过并行方法将延迟降低了许多。由于传统机器学习方法有着延迟低、模型小、训练时间短、可解释性强等优点，在深度学习迅猛发展的今天，传统机器学习方法仍然占有很大的市场，故评估 GPU 对传统机器学习加速的性能是十分有必要的。本文中在评估传统机器学习的部分便采用了并行的支持向量机算法 (SMO-SVM)，如今，不管是在工业生产领域还是学术研究领域，该算法仍占有一席之地。

之后不久，深度学习、神经网络便迎来了爆炸式的发展。实际上在二十世纪末，便已经有完整的神经网络算法的理论基础；在 1979 年，日本学者福岛邦彦提出了 Neocognition 模型，其中使用多层网络以及神经元对图像特征进行提取和筛选被认为是启发了卷积神经网络的开创性研究 [15]。1989 年，LeCun 首次在论文中提出了“卷积”一次，卷积神经网络因此得名 [16]。在 1993 年，贝尔实验室对 LeCun 的工作进行了代码实现，并大量部署于手写支票识别系统，然而限于当时计算能力低下，基于神经网络的研究也停滞在了理论阶段，其主要原因便是网络中需要训练的参数太多，网络结构复杂，在当时没有芯片能满足如此高的性能要求 [17]。然而，随着高性能 GPU 芯片的出现，基于神经网络的方法正如日中天发展。从 TinyCNN 等较轻量级、功能单一的库，到 TensorFlow、PyTorch、Caffe、Chain 等完整、易于使用的框架，这些工具或是本身就是基于 GPU 编写的，或是慢慢更新对 GPU 的支持；目前市面上的绝大部分该类产品均支持使用 GPU 运行，单单使用 CPU 进行深度学习训练已经成为历史。在本文中，卷积神经网络由于它的广泛性、高性能、典型性，在本文中被选为深度学习部分评估的主要载体。

而准确地对 GPU 以及机器学习的性能进行评估也尤为重要。且为了深入研究 GPU 对于机器学习应用的性能提升的幅度以及不同提升幅度的不同原因，单单是对训练时间进行统计、评估是不够的。因本文涉及 Nvidia 新架构 GPU 中新加入的硬件以及对应 CUDA 中新加入的 API 等，故指令、运行流级别的分析是有必要的。Ali Bakhoda 等人曾设计实现了一种在软件层面对 CUDA 执行流进行指令级别的模拟和仿真的系统 GPGPU-SIM，该系统是基于 Kepler 架构的硬件以及 CUDA 3.0 版本，在缓存命中率、分支、指令乱序执行等方面能达到 90-95% 与真实硬件的吻合程度 [18]。在之后 Mahmoud Khairy 等人对该系统进行了改进，使其支持伏特架构 (Volta) 的 GPU 以及对应的 CUDA 9.0 [19]。然而，目前 GPU 硬件已经更新到图灵架构 (Turing)，基于安培架构 (Ampere) 的硬件也即将发布；对应的 CUDA 版本已经更新到了 10.1，在指令执行、调度方式等方面都发生了很多的改变；且 Mahmoud Khairy 等人的工作主要着重于 GPU 的内存等方面。故能够准确评估新硬件的系统非常必要。本文中选用了 nvprof、NSight 等公开的工具，这些

工具能从指令运行时间、访存、缓存命中率等方面对 CUDA 应用程序进行评估 [20]。

在新架构的 GPU 中，最为重要的便是新加入的计算单元：张量核心 (Tensor Core)，该运算单元能为深度神经网络中大量存在的张量计算带来明显的提升 [5]，然而这些数据是 Nvidia 官方白皮书中给出的数据，开发者社区中反映很少有情况能获得如 Nvidia 官方宣传所能得到的性能提升；而关于 Tensor Core 的研究少之又少，故本文并非旨在填补这方面研究的空白，姑且在新的方向进行一些稚嫩的尝试；且由于在 Nvidia 进行实习工作，有机会接触到许多内部资料，本文是一个很好的契机。

当然，仅有理论计算性能的研究是无力的，最终本文还是会回归实际，使用实际应用中的模型，如各种结构的神经网络、广泛使用的支持向量机并行库对新架构的 GPU 进行评估。从广为各大厂商使用的深度学习性能评估工具 DeepBench[21]，到使用 cudnn 从 C++ 源码实现的卷积神经网络，再使用 Tensor Flow 框架实现的各种结构的网络，包括 LeNet-5[16]，ResNet[22]，MobileNet[23] 等，本文将由下而上对新架构硬件再浮点精度计算、张量计算、卷积计算、矩阵计算等方面进行评估，不求全面，只求能给出启发。

1.3 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。

1.4 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。

第二章 背景及相关工作

2.1 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旧的架构和旧 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，尝试修改选定框架 (Tensor Flow) 的源码，并给出实际的修改、编程时的建议。

2.2 NVIDIA GPU 硬件结构与硬件机制

2.2.1 GPU 芯片总体结构

在介绍新老架构区别之前，本节首先自顶向下简要介绍一下 NVIDIA GPU 芯片的结构。一块 GPU 芯片拥有若干图形处理器簇 (Graphics Processing Cluster, GPC)，由外围总线进行调度管理；一个图形处理器簇上有若干纹理处理器簇 (Texture Processing Cluster, TPC)；需要注意的是以上两种结构在编写 CUDA 程序时并不暴露。一个纹理处理器簇上有若干流多处理器单元 (Stream Multiprocessor, SM)，也是本文关注的重点。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经过 L2 缓存共享全局内存。每个流多处理器单元中由若干流处理器 (Stream Processpr, SP)，然而这一概念随着流多处理器单元中运算单元种类的增加而被弱化了。在一个流多处理器单元内部的流处理器共享一个指令缓存，每个流处理器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 [24]。

2.2.2 流多处理器单元 (SM)

上文提到过，流多处理器单元 (SM) 是本文关注的重点，其原因是每一次 NVIDIA GPU 芯片更新都会伴随着其计算能力 (Compute Capability) 的更新，计算能力指的是流多处理器单元 (SM) 支持的运算的

表 2-1 CUDA 存储系统层级

Table 2-1 CUDA storage system hierarchy

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

等级，分为 Major 和 Minor。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。

流多处理器单元中的不同类型的流处理器对应了不同的流水线，不同流水线的延迟、吞吐量均不同。且有些流水线指令发射是互斥的，即互斥流水线的指令不能同一时间发射，这将在下文介绍新硬件架构时提到，本文中主要关注流多处理器上的计算指令流水线 (M-Pipe)，光栅等其他计算涉及的流水线不做介绍。

2.2.3 存储模型与管理

随着 GPU 计算性能的增长，传统的、由硬件全权管理的内存的访问模型由于无法最大程度利用时间/空间局部性逐渐成为限制性能的瓶颈 [25]。NVIDIA GPU 的存储模型与其存储管理系统也是另一个重点。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 [26]。CUDA 中的存储设备如表2-1 所示。

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；但是在访存、缓存加载方式上与其余存储系统存在较大差异，而这种差异会在某些应用中极大提高性能，在本文的实验中大量利用了纹理内存的特性，故将在下一节详细介绍纹理内存。

2.2.4 纹理内存 (Texture Memory)

分层存储系统中都会采用缓存，利用程序的空间局部性与时间局部性来加速程序访问、写回数据的速度。在 GPU 中，纹理内存的缓存加载方式与其余存储系统方式不同。纹理内存加载所访问数据周围一个范围内的数据 [27] 供其余线程访问，而其余存储系统仍然采用加载所访问数据所在行中别的数据供其余线程访问，如图2-1 所示，左侧为线程束中线程访问纹理内存的形式，右侧为线程束中线程访问其他存储系统的形式。

使用纹理内存的原因是由于 GPU 需要执行大量图形运算，处理某一像素点时其周围像素点也有很大可能一并处理，如抗锯齿作业，采用这种加载一个二维区域内的单元的方式能改善这种情况下的访存性

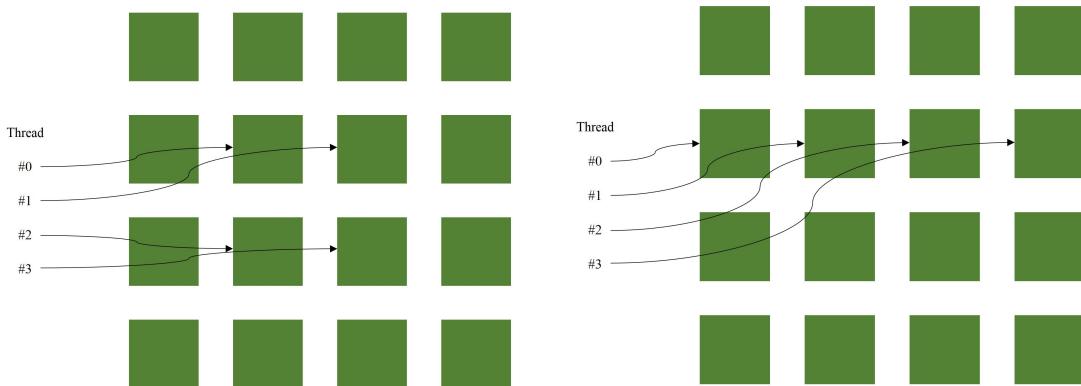


图 2-1 纹理内存和其余存储系统

Figure 2-1 Texture memory and other memory system

能。而在卷积神经网络中，卷积核也需要对一个二维区域内的单元进行处理，故本文中将尝试使用纹理内存优化卷积神经网络中的卷积运算。

2.2.5 硬件机制

2.2.5.1 线程组织形式 本节将介绍 GPU 上任务的调用机制。根据弗林分类法 [28]，计算机系统可以分为 SIMD, MIMD, SISD, MISD 等类型。目前多核心 CPU 系统就是 MIMD 系统，而 NVIDIA 的 GPU 系统被称为 SIMT(Single Instruction Multiple Thread) 即单指令多线程，与 SIMD(Single Instruction Multiple Data) 即单指令多数据不同；在这种模型中，一种指令并非仅仅代表一个固定的功能，而是代表这一指令使用的流水线类别，即指令的类型，线程需要执行的具体操作需要编写相关内核代码。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。

在 GPU 程序中，32 个线程被组织为一个线程束 (warp)，作为基本的调度单元，拥有各自的物理寄存器。也就是说线程束中的 32 个线程一般情况下会执行相同的指令流访问不同的数据，即 SIMD 模式。且线程束目前仍然作为同步的最小粒度，即线程束和线程束可以保证同步，而线程束内部的线程无法保证同步。在下一代安培架构 (Ampere) 中则将引入 *Arrive – Wait* 模式以实现线程级同步，以提高 GPU 程序的灵活性。

若干个线程束被组织为一个线程块 (block)，线程块之间通过共享内存进行数据交换，线程块中的线程束可以通过 `_syncthread()` 进行同步。在下一代架构中将添加线程束组的层级 (warp group)，由四个线程束组成，然而该层级仅为大规模通用矩阵乘法指令所用，这里不做讨论。

若干个线程块被组织为一个线程网格 (grid)，线程网格可被看作一个分配给 GPU 的任务，故线程网格的虚拟内存空间是相互独立的。

表2-2 详细说明了 GPU 中不同粒度的线程的组织形式以及相应调度者。

2.2.5.2 调度方式 以上内容介绍了 GPU 线程的组织形式，这里将介绍 GPU 线程的调用方式，在介绍调用方式之前，需要先明确 GPU 与 CPU 的硬件上的差别，主要有如下几点。

表 2-2 线程组织形式

Table 2-2 Type of organizing threads

粒度	调度者	分配给
warp	线程束调度器	流处理器 (Stream Processor)
block(CTA)	TPC 调度器 (MPC, M-Pipe Controller)	流多处理器 (Stream Multi-processor)
grid	GPC 调度器 (GPM, Graphics Pipe Manager)	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 [29]。

表 2-3 可分配线程数与占用率的关系

Table 2-3 Available thread number and corresponding occupancy rate

	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

- GPU 中存在大量的物理寄存器, 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2-3 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。

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数量越多。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。

- 发生分支时, CPU 采用分支预测-预测错误则清空流水线的机制进行, 而 GPU 则引入了线程束分化概念, 这一概念将在下文详细介绍。

GPU 程序首先由内核 (kernel) 开始, 一个内核代表一个需要在 GPU 上运行的程序, 由 CPU 管理、分配给 GPU 上的图形处理簇 (GPC)。一个大任务会被拆分为若干小任务, 这些小任务被抽象为线程网格 (grid) 由图形处理簇上的调度器分配给纹理处理簇 (TPC)。在纹理处理簇上的调度器将以线程块 (block) 为单位将线程分配给流多处理器单元 (SM)。最后, 具体的运算任务将由流多处理器单元内部的线程束调度器分配给不同流处理器 (SP)。

在调度时, 线程束作为最小的调度单元在一般情况下将会运行同一指令流, 这一情况在发生分支时会有不同。线程束被分发到分支指令时, 若线程束内部线程发生分化 (即一部分执行分支, 一部分不执行分

支), 则两个分支都会被执行, 分支结束时只有执行了正确分支目标的线程会被置未激活状态, 执行了错误分支目标的线程被设置为未激活状态。由于线程束调度器每次只能为一个线程束取到一条指令, 这意味着线程束内部发生线程分化时, 未激活状态的线程会被阻塞, 导致利用率、性能下降。自麦克斯韦架构开始, 为了尽可能减小线程束内部线程分化对性能造成的影响, 指令是以半个线程束即十六个线程进行分发的。之后, GPU 引入了 CBU 单元 (Convergence Barrier Unit) 以保持通过同一分支的线程被组织在一起的方式尽可能减少分支带来的性能下降 [30]。在实际编程中, GPU 提供非常细致的线程、线程块、线程网格 ID, 故根据这些 ID 控制分支使得线程束内部线程分化尽可能减少是可行的 [31]。

最后将介绍 GPU 中的流以及相应的锁页内存的概念。GPU 中的流可以被看做一个 GPU 操作队列, 该队列中的操作将会按顺序执行。这一特点使得在实际编写程序的过程中可以通过调整不同操作的顺序, 如访存、密集计算、写回等来隐藏数据依赖和控制依赖带来的延迟 [32], 通过异步操作 (ASYNC) 提高任务级并行度。CUDA 流需要在支持设备重叠的 GPU 上运行, 设备重叠使得 GPU 能够在执行一个核函数时在设备和主机间进行数据交互。而为了支持流, 所有涉及流的内存需要在锁页内存上进行分配。锁页内存正如其字面含义, 是分配在显存上、不允许被移动或被换页到磁盘上的设备内存。其好处是允许 GPU 上的 DMA 控制器绕过 CPU 直接请求主机传输数据, 在延迟更低的基础上支持设备重叠 [33]。若不使用锁页内存, DMA 控制器在定位内存页面时会造成困难。本文的部分实验中使用了流这一特性, 将在具体实验章节详细介绍。

2.3 伏特/图灵架构新硬件

2.3.1 在流多处理器单元层面的差异

图2-2 展示了伏特/图灵架构 (Volta/Turing) 与帕斯卡架构 (Pascal) 在流多处理器单元层面的异同, 左侧为伏特/图灵架构、右侧为帕斯卡架构。

在调度方面, 指令缓存、线程束调度器与寄存器并无太大差别, 而指令分发器 (Dispatch Unit) 由帕斯卡架构中的每 SM 两个变为每 SM 一个, 这是因为在处理分支时, 如上文提到的在伏特架构以前为了减少线程束内部线程分化带来的性能下降, 指令以十六个线程进行分发, 故存在两个指令分发器; 而后则采用 CBU 单元处理分支指令, 故仅存在一个指令分发器。

在运算单元部分, 帕斯卡架构简单的分为计算单元 (Core)、访存单元 (LD/ST, Load/Store) 和特殊功能单元 (SFU, Special Function Unit) 三个部分, 其中计算单元统一为单精度浮点计算单元, 访存单元负责一组流处理器依赖数据的请求和结果的写回, 特殊功能单元则负责逻辑判断、分支等操作。在新架构中, 计算单元根据精度被分为双精度浮点计算单元 (FP64)、单精度浮点计算单元 (FP32)、整数计算单元 (INT) 以及最新加入的张量计算单元 (Tensor Core), 也是本文的重点。在访存单元和特殊功能单元方面的变化本文并不做详细讨论。在新架构中, L1 数据缓存的概念也与共享内存的概念统一了。

2.3.2 张量核心 (Tensor Core)

作为本文的重点, 张量核心 (Tensor Core) 将在本文详细介绍。张量核心是一种专为通用矩阵乘法运算 (GEMM, GEneral Matrix Multiple) 设计的运算单元, 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。

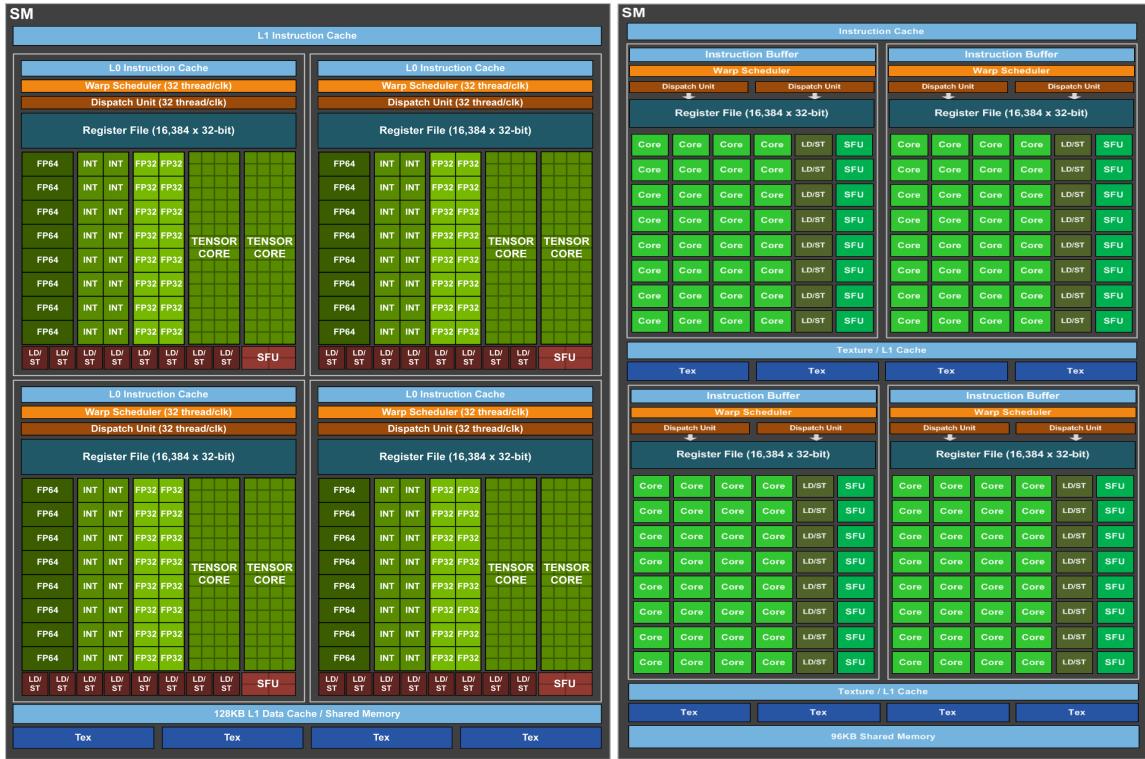


图 2-2 伏特架构与帕斯卡架构的流多处理器示意图

Figure 2-2 SM diagram of Volta and Pascal architecture

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，如图2-3 所示 [4]。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，再将结果累加到乘加矩阵中。而使用张量核心则仅通过一条指令直接完成。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在 FP16, INT8, INT4 精度中分别提供 8 倍、16 倍、32 倍的吞吐量提升。实际测试中，Tesla V100 在 FP16 精度下的 $m = 2048, k = 2048, n = 2048$ 规模的矩阵乘加中比 Tesla P100 快 9.3 倍 [5]，这也是上文提到的官方宣称的 9 倍。在基于 Pascal 架构以及之前的架构的 GPU 上，一次 4×4 矩阵乘法需要调用 16 次向量点积加和运算，在有 *idp4a* 指令支持的 GPU 上在 16 个周期内运算出乘积矩阵，若没有指令支持则需要 64 个周期，而运算得出的乘积矩阵还需要累加；而张量则在更少的周期内完成乘积和加和的运算，具体周期数由 *hmma* 指令的操作数决定，对于被分割为 $m = 8, n = 8, k = 4$ 的子矩阵的运算，张量核心仅需要八个周期完成所有乘积和加和运算 [34]。本节将在各种规模、精度、形状的情况下考察张量实际能够带来的性能提升并探究相应原因。

值得注意的是，在流处理器单元中，张量核心是与其他运算单元包括浮点单元、整数单元共用调度器，调度器每个时钟向分支单元、张量核心阵列、数学分派单元或访存单元发出一个线程束调度指令，这就表明一个周期内硬件不能同时执行张量核心负责的混合矩阵运算和其他运算，单一周期只能执行单一类型的运算 [35]。张量核心有一定程度的可编程性，但是 4×4 矩阵仍然是操作的最小单元，在 LLVM IR 的级别观察其指令仍是如此。尽管被描述为最小 4×4 为单元的矩阵的混合运算，但实际上张量核心进行的运算总是使用 16×16 矩阵，并且跨两个 Tensor Core 进行处理 [2]，这与其最小调度单元为线程束，也就是 32 个线程有关。

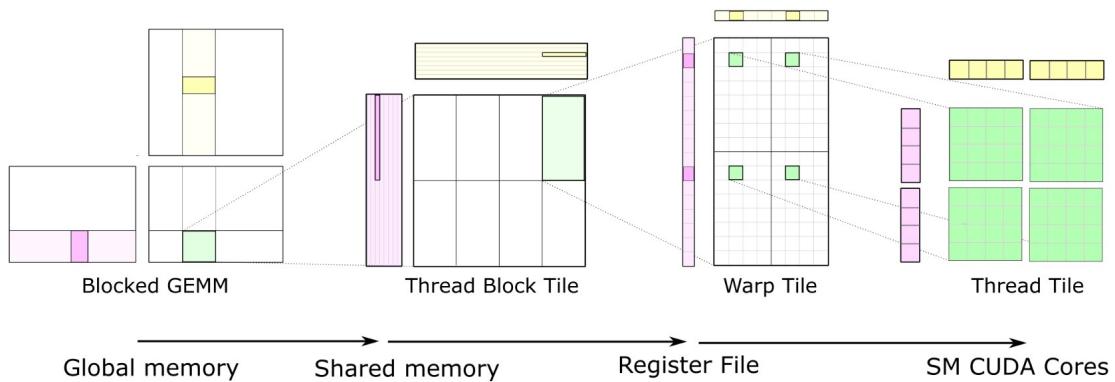


图 2-3 使用张量核心进行通用矩阵乘法计算的层级

Figure 2-3 Hierarchy of GEMM using Tensor Core

2.4 软件

本节将自顶向下介绍 NVIDIA GPU 硬件对应的不同层级的编程软件及相关工具。

2.4.1 机器学习框架 (Tensor Flow)

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。本文将挑选目前非常流行的 Tensor Flow 框架作为最上层应用的研究载体，考察如何根据硬件特性对 Tensor Flow 级别的源码以及 Tensor Flow 本身的源码（基于 C/C++）进行优化以达到更快的速度、更高的吞吐量。为达到这一目的，需要对 Tensor Flow 框架从源码重新编译，具体步骤此处不再赘述 [36]。

2.4.2 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编程语言包括 C/C++、Python、Matlab、Fortran 中以关键字的形式加入扩展来描述并行行为 [37]。本文选用 C/C++ 进行 CUDA 的扩展，基础的核函数编写、存储系统调用 [38] 此处不再说明。

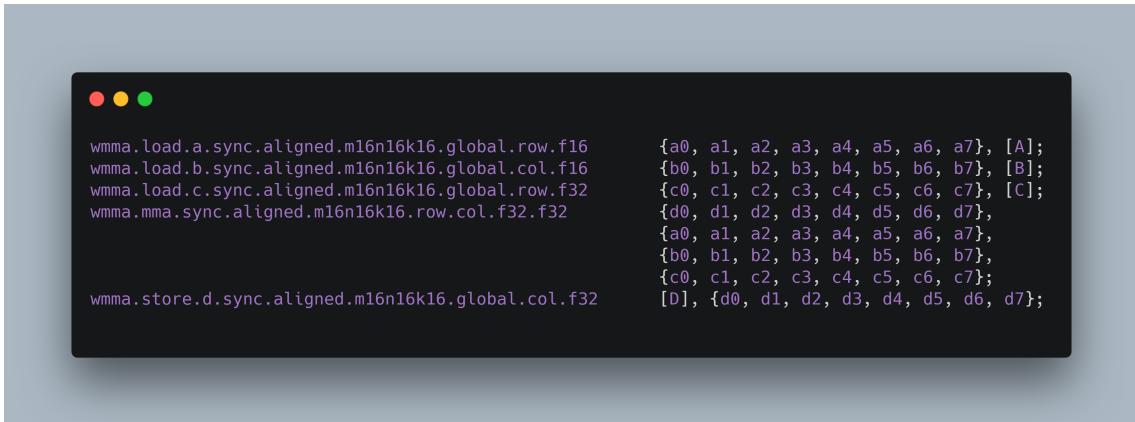
在本文中的实验进行时，CUDA SDK 最新版本为 10.0，主要在 CUDA SDK 9.2 的基础上对图灵架构进行了支持；对更多分割尺寸的通用矩阵乘法运算进行了支持 [39]；以及对多 GPU、不同操作系统、兼容性等方面进行了更好的支持。而 CUDA SDK 9.2 相对于 CUDA SDK 9.0 则更新了对伏特架构以及对应张量核心 (Tensor Core) 的支持，属于一次较大的更新 [40]。在文章撰写之际，NVIDIA 推出了 CUDA SDK 10.1，在 CUDA SDK 10.0 的基础上更新了轻量级矩阵乘法库；更新了对应的构建、测试工具等 [41]。CUDA SDK 10.1 的更新对于本文的实验结果并无决定性的影响，故本文仍然采用 CUDA SDK 10.0 进行实验。

CUDA 程序经过一系列工具如 nvcc 等被编译为中间代码 (PTX)，再从中间代码编译为能直接运行于

Instruction	Specified Operations								Operands Register, Immediate
	wmma	.load	.ident	.sync	.aligned	.shape	.memory	.row/.col	
		.store	.mma					.row/.col	
								.dtype(s)	

图 2-4 中间代码 (PTX) 格式示例

Figure 2-4 An example of PTX code



```

wmma.load.a.sync.aligned.m16n16k16.global.row.f16      {a0, a1, a2, a3, a4, a5, a6, a7}, [A];
wmma.load.b.sync.aligned.m16n16k16.global.col.f16     {b0, b1, b2, b3, b4, b5, b6, b7}, [B];
wmma.load.c.sync.aligned.m16n16k16.global.row.f32     {c0, c1, c2, c3, c4, c5, c6, c7}, [C];
wmma.mma.sync.aligned.m16n16k16.row.col.f32.f32       {d0, d1, d2, d3, d4, d5, d6, d7},
                                                       {a0, a1, a2, a3, a4, a5, a6, a7},
                                                       {b0, b1, b2, b3, b4, b5, b6, b7},
                                                       {c0, c1, c2, c3, c4, c5, c6, c7};
wmma.store.d.sync.aligned.m16n16k16.global.col.f32     [D], {d0, d1, d2, d3, d4, d5, d6, d7};

```

图 2-5 一段具体的中间代码 (PTX)

Figure 2-5 Part of actual PTX code

目标硬件的硬件代码 (SASS)。

2.4.3 硬件代码 (SASS) 与中间代码 (PTX)

中间代码 (PTX, Parallel Thread eXecution)[42] 是一种并行线程执行虚拟机代码 (IR)，在实际运行时，该中间代码会被编译为硬件代码 (SASS)。本实验中为了详细观察硬件层面的操作，将在编译上层应用程序时添加 *--keep* 保留编译时产生的中间文件。同时，本节将简单介绍中间代码的语法供后文使用。

中间代码中由两种语句：Directive Statement 和 Instruction Statement，前者起到声明作用，后者则是具体运算动作，起到声明作用的语句在此不再赘述，详细考察具体运算动作的指令，以本文中的重点：线程束级矩阵乘加指令 (wmma, warp-level matrix multiply Accumulate)，也就是张量核心使用的指令为例，如图2-4 所示。中间代码主要分三个部分：指令、具体操作和操作数与传统的 SIMD 模型中的指令、操作数不同，在 SIMD 模型中，其“多线程”的特点正是对应了中间代码中的具体操作这一部分。在具体操作这一部分中，中间代码会指明存/取、是否同步、操作数数据分布 (行/列主元素)、数据精度等项目。图2-5 是一段具体的涉及 wmma 指令的中间代码，代码中操作数形状、数据类型等将在实验中具体介绍。

硬件代码 (SASS) 与中间代码 (PTX) 类似，分为指令、具体操作和操作数三个部分，其区别是硬件代码会开放涉及浮点数存储方式、分支预测操作等底层的、直接对硬件进行的操作，本文中不再详细介绍，仅会在涉及张量核心的实验中提到对应中间代码中 wmma 指令的硬件代码中的 hmma 指令。

2.4.4 相关工具

在实验中仍有一些工具在上文没有提到，将统一在本节介绍。

2.4.4.1 性能评估 (Profiling) 工具 为了详细地考察硬件底层的运行机理，实验中采用了若干不同层级的性能评估工具。

- **Nsight:** 该工具用于在后台监听 CUDA 程序的执行流，结合 Visual Studio 中的工具可以通过可视化的方式直观地观察 CPU/GPU 调用、上下文切换等信息 [20]。
- **nvprof:** 通过该工具分析某一 CUDA 程序，可以获得程序执行时的分支效率、访存效率、各 API 调用占比等信息 [43]。
- **GPGPU-SIM:** 该工具支持从中间代码 (PTX) 层级对 CUDA 程序的运行进行模拟、仿真，该工具能够提供缓存是否命中、分支是否执行、指令执行周期数等信息 [18]。与 GPGPU-SIM 搭配的还有 NVIDIA 内部使用的硬件代码 (SASS) 层级的仿真工具 SMart，该工具能够更为详细地提供指令运行时硬件的各种信息。由于涉及企业机密本文将不详细说明 SMart 工具的细节。

2.4.4.2 神经网络推理工具 在实际的机器学习应用中，模型的训练只是第一步，在模型完成训练后需要对模型进行部署。当模型构建完毕，实际部署到目标机器、芯片、平台上并发挥作用是，推理就成为了主要工作。与训练的过程和目的不一样，推理首先没有了训练过程中的反向迭代，同时对于吞吐率、延迟、功耗等有着更高的要求；且现有框架在实际部署时因其复杂度高，对部署造成了很大困难，TensorRT 由此而生。

TensorRT 是一个高性能深度学习推理的平台，包含深度学习推理优化器和运行时应用来提供低延迟、高吞吐量的深度学习推理。本节将考察 TensorRT 相对于传统神经网络推理的加速性能 [44]。因 TensorRT 原生支持 TensorFlow，这就意味着可以直接使用 TensorRT 自带的模型导入器对训练好的模型进行导入，故本文通过这种方法来考察 TensorRT 的实际提升。

在实际使用 TensorRT 之前，将先对 TensorRT 的优化以及部署流程进行介绍。首先需要明确的是，TensorRT 作为一个 GPU Inference Engine(GIE)，它是在项目部署阶段发挥作用的。TensorRT 的部署分为两个部分：优化训练好的模型并生成计算流图和使用 TensorRT Runtime 部署计算流图。其最重要的便是优化过程，如下图2-6 所示 [45]。

优化过程的第一部分，也是较为重要的部分便是网络层和张量的合成，这一过程将会在不改变底层计算内容的情况下重构计算图来获得更高效的计算方式。DL 框架在进行推理时会执行多个子过程，这些过程都需要 GPU Kernel 来运行，这就不可避免地带来会更多 Kernel Launch，如前文所说，Kernel Launch 所带来的上下文切换仍然是很大的开销。TensorRT 主要从 Kernel 纵向融合 (如卷积、偏置、激活层的 Kernel 可以融合为一个 Kernel)；Kernel 横向融合 (挖掘输入数据形状相同但权重不同的层，使用一个 Kernel 而非原来的若干个 Kernel 提高效率) 和消除 Concatenation 层 (通过预分配输出缓存、不同的写入方式来避免转换)。

第二部分，即能鲜明体现新硬件的部分：FP16 和 INT8 精度校准。大部分网络在训练是为了精度，会使用 FP32，因此最后输出模型也是基于 FP32 精度。然而完成训练也就意味着参数达到最优，在推理过程中由于没有反向传播故采用更低的精度能在不牺牲过多准确率的情况下极大提高网络计算吞吐率。

第三部分，动态分配的张量显存能在使用期间减少显存占用率，增加显存复用率，从而避免过度开销以提高推理性能。

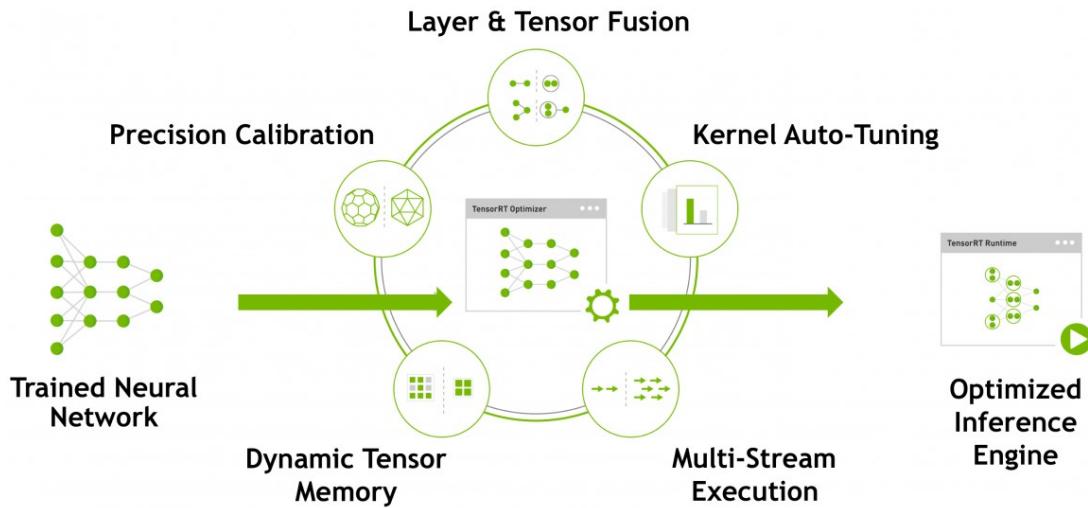


图 2-6 TensorRT 中的模块

Figure 2-6 Modules in TensorRT

最后便是 Kernel 自动调整和多流执行，TensorRT 会针对输入数据大小，filter 大小，张量数据分布，batch 大小针对目标 GPU 进行选择和优化，这一部分与传统编译器后端处理，即及其相关优化较为相似。同时 TensorRT 也会借助 CUDA 流来提高任务级并行度，这点也在上文提到过。

由于经过 TensorRT 优化后的模型只能运行于特定硬件平台 (TESLA T4, TESLA V100, Jetson 系列)[45]，本文将采用由 NVIDIA 提供的 Jetson TX2 开发套件进行 TensorRT 部分的实验，NVIDIA 的 Jetson 系列是专为嵌入式机器学习应用设计的芯片。在 GTC 2019 会议中，NVIDIA 发布了基于最新技术的 Jetson Xavier，然而由于成本原因，本文不使用该硬件。

表 3-1 实验环境

Table 3-1 Environment

项目	内容
CPU	AMD Ryzen ThreadRipper 2990WX 32C64T @ 3.0GHz
主板	MSI X399
内存	CORSAIR DDR4 3200 @ 16-15-15-34-1T 128GB
GPU	NVIDIA Geforce RTX 2080TI (Turing)
硬盘	INTEL750 NVMe PCIe 1.2TB * 2 @ RAID 0
系统	Windows 10 64-bit build 17763
CUDA	10.1, 10.0, 9.2, 9.0
其他	Jetson TX2 *

* 该硬件由 NVIDIA 提供。

表 3-2 实验工具

Table 3-2 Tools

项目	内容
Python 3.6	用于进行数据统计、编写 TensorFlow 应用
Conda 4.5.12	用于创建、管理、隔离 Python 环境
TensorFlow	1.12.0 和 1.13.0 版本的源码，用于对比、研究、调整
Bazel 0.16.0	用于从源码构建 TensorFlow
Msys2	用于从源码构建 TensorFlow
CMake 3.1.0	用于构建 CUTLASS
Nsight 6.0	用于后台监听 CUDA 应用，捕捉 Trace
nvprof	用于分析 CUDA 程序的 API 调用、分支效率等
git	版本控制
Perforce	版本控制
Ubuntu 16.04 Physical	用于执行 GPGPU-SIM 应用
GPGPU-SIM 3.2	用于从指令级别模拟 CUDA 程序
Visual Studio 2017	搭配 10.0.17763.0 版本的 SDK

第三章 评估 NVIDIA 新架构 GPU 的机器学习应用性能

3.1 实验工具与环境

3.1.1 实验环境

表3-1 中列出实验环境。

3.1.2 实验工具

实验中使用到了若干软件工具，如表3-2 列出。

3.2 实验详细过程

3.2.1 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 的原因是由于 9.2 版本是为了图灵架构的 GPU Tesla V100 发布的 [46]，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。

3.2.1.1 通用矩阵乘法 (GEMM, General Matrix Multiply) 作为新硬件最为鲜明的特点，本节将对张量核心以及相应的通用矩阵乘法运算进行评估，通用矩阵乘法运算又被称为混合矩阵运算，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，目前张量核心仅能用在 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。这个范围看起来很窄，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。

I 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[21]，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。

首先评估的是在不同问题规模下，开启和关闭新架构中的张量核心所能达到的性能，如图3-1 所示。随着问题规模的上升，总体加速比呈上升趋势，在大规模数据时半精度通用矩阵运算性能的加速比能达到 3 到 3.5 倍、单精度通用矩阵运算性能的加速比能达到 2 倍。然而，单纯考察数据规模发现加速比差距非常大，甚至是在大规模数据中仍然存在开启张量核心后性能不如不开启张量核心的情况，结合文档在通用矩阵运算一章中在指令中需要指明运算的最小单元这一点中 [42]；可以推测出输入矩阵的“形状”对张量核心的性能由较大影响。

为了研究输入矩阵“形状”对于加速比的影响，由于两个输入矩阵涉及三个维度，故采用控制变量法，控制 m, n, k 中某一维度考察另外两个维度对于加速比的影响。由于测试数据中存在部分离群值 ($N \geq$

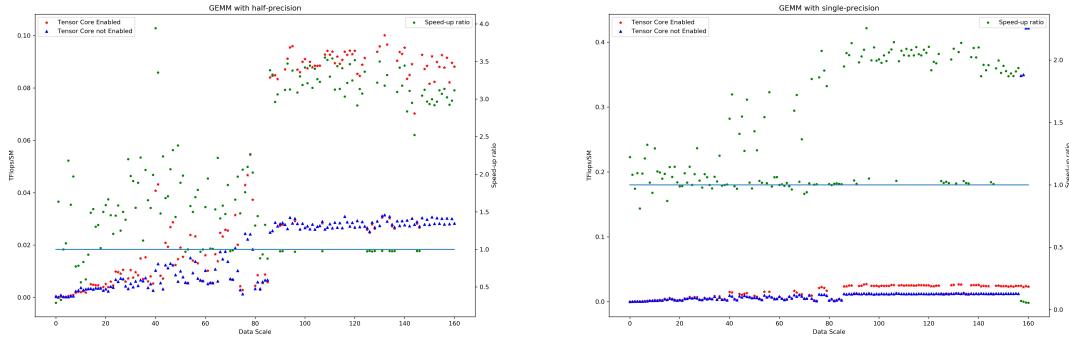


图 3-1 半精度/单精度 GEMM 性能

Figure 3-1 Performance of GEMM at Half and Single

500000), 这会对作图精度产生极大影响, 故先予以剔除。实验结果如图3-2 所示。可见在两个输入矩阵的三个维度中, 两矩阵共享的维度 K 对于性能的影响最为显著。

关于矩阵的形状、维度对于性能的影响, 其原因将在后文结果分析中详细说明。以上实验数据以及性能旨在考查开启和关闭张量核心时性能提升幅度, 故开启的线程块数量和线程数量较小, 相应的 GPU 占用率也较小, 导致所得性能并非 GPU 峰值性能。为测量峰值性能, 这里还是用 NVIDIA 官方发布的用于线性代数计算的模板库 CUTLASS(CUDA Template Linear Algebra Subroutines), 该模板库根据新架构硬件特性编写, 提供了许多测试样例供参考, 这里使用 CUTLASS 测试得到的性能作为峰值性能基准。

CUTLASS 库中的 GEMM 运算有多种精度可供选择: HGEMM、SGEMM、DGEMM、CGEMM、ZGEMM 和 IGEMM, 分别代表半精度浮点、单精度浮点、双精度浮点、单精度复数、双精度复数和八位整数。鉴于之前的测试并不涉及复数, 此处也不选用复数精度作为基准。需要注意的是测试样例后缀中存在 $[n/t][n/t]$ 分别代表运算中输入矩阵的数据存储、分布方式, 即行列是否转置(如上文提到的, CUDA 中矩阵存储分行主元素和列主元素存储)。图3-3 为测得的性能基准。

II 结果分析

首先, 本实验通过 Nsight 和 nvprof 的搭配对应用程序中包括 API 调用、核函数运行时间等运行时细节进行研究。图3-4 和图3-5 是开启和关闭张量核心的情况下, 使用 nvprof 运行通用矩阵乘法应用所得到的报告, 该报告主要侧重于 API 的调用、运行。由于本文关心的重点在 GPU 硬件, 故仅截取硬件相关的 API 调用 (API Call) 部分。图中由左至右边分别代表 API 调用占比、运行总时长、调用次数、平均运行时长、最短运行时长、最长运行时长和 API 名称。

根据 nvprof 生成的报告, 可以看出在开启张量核心的情况下, 用于设备上线程束同步的 API: cudaDeviceSynchronize(), 无论是在运行总时长还是调用占比中都显著小于不开启张量核心的情况。另外用于开启内核运行和异步访存的 API 的调用次数在开启张量核心时都明显小于不开启张量核心的情况。可见开启张量核心后, 由于原先需要多条点积指令的矩阵乘法运算被合并为仅用一条 wmma 指令替代, 其计算更加密集、设备同步更少, 故性能提升明显。表3-3 显示了根据 GPGPU-SIM 测得的张量核心相关指令与一般点积运算指令所需要的运行周期数, 以及相应开启内核 (Launch Kernel) 指令所需的运行周期数, 根据指令运行周期级别的数据可以看出使用张量核心相关指令 (wmma) 不仅能极大减少计算所花费的时间, 用于开启内核、上下文切换的时间也会显著减少。

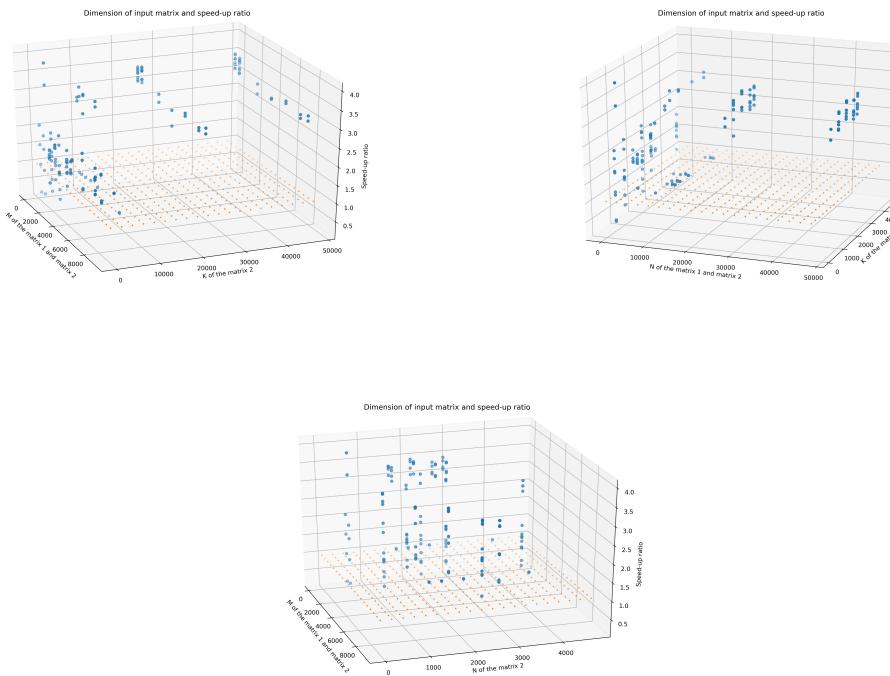


图 3-2 输入矩阵维度与加速比的关系

Figure 3-2 Relationship of input matrix dimension and speed-up ratio

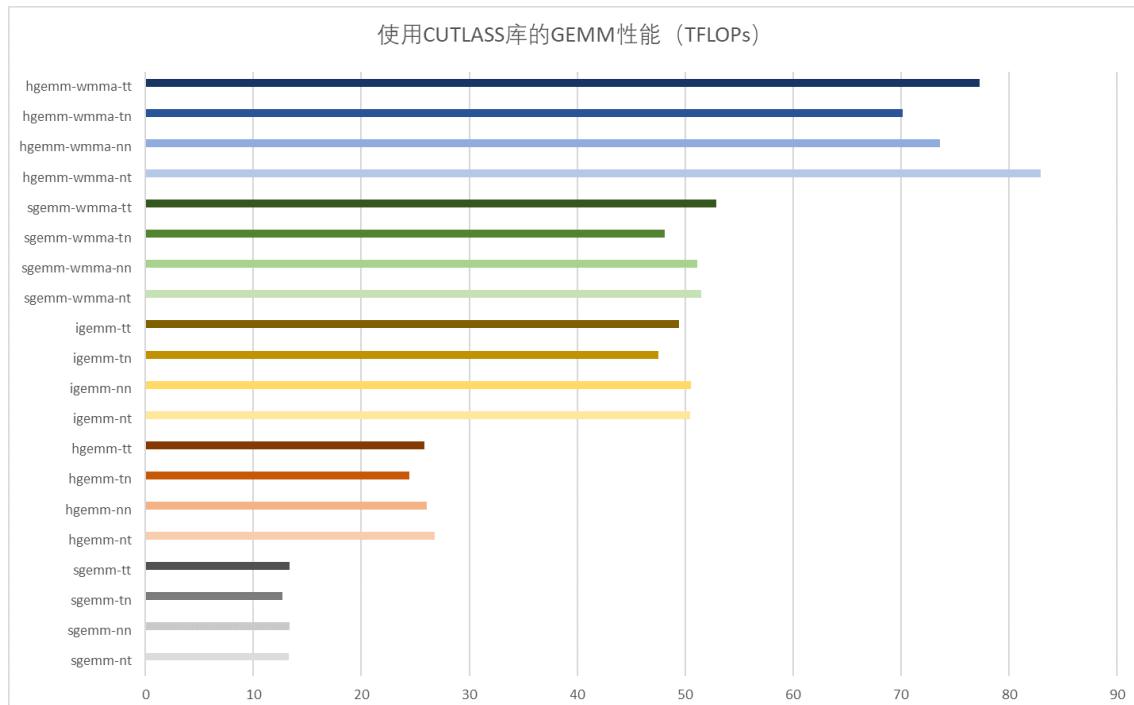


图 3-3 使用模板库测得的 GEMM 性能

Figure 3-3 GEMM Performance with CUTLASS

图3-6 和图3-7 则是开启和关闭张量核心的情况下，使用 Nsight 运行通用矩阵乘法应用所得到的报告，该报告主要侧重于上下文切换、线程调度等信息。在这些信息中本实验重点关注线程就绪 (Ready Thread) 和上下文切换 (Context Switch) 两部分。

根据 Nsight 生成的报告，可以看出无论是否开启张量核心，其上下文切换仍然是一笔较大的开销。正如上文提到的，在 GPU 上进行上下文切换，其寄存器可以简单地通过更改寄存器文件指针保存/还原，然而上下文切换所需要做的清空流水线等工作还是无法被省略，造成较大的开销。相比较而言开启张量核心的应用在上下文切换中的开销下降了 10% 左右，而因为就算不使用张量核心， 4×4 规模的混合矩阵运算也能使用点积指令在 16 个周期内完成，故应用中运算开销相对于上下文切换、加载内核映像的工作占用资源较小。且使用张量核心的应用中硬件中断 (IRQ) 占比也较小，说明使用张量核心不仅在运算速度上有极大提升，在与外围设备数据交换，缓存读取、命中等方面都有较大优势。当然，因计算优势在缓存读取、命中等方面体现，故输入数据的“形状”能否符合张量的硬件特性变得尤为重要。

在上文的实验中，可以看出矩阵乘加运算中，两输入矩阵共享的维度 K 对性能影响较为显著。在 $M \times N$ 的结果矩阵中，每个单元格都是由 $1 \times K$ 的矩阵与 $K \times 1$ 矩阵相乘，这个运算将被拆分并分发给张量核心进行处理，那么在这一步，若 K 无法被张量核心正好拆分，则会引发一个数据缺失，进而造成硬件中断，旨在从共享/全局内存获取新的数据以拼成完整的可以交予张量核心进行处理的单元。若没有这种数据则调用传统运算核心进行运算。那么张量核心硬件上是以 4×4 作为计算单元，调度时以 16×16 作为调度单元，那么隐含的就要求 K 能被恰好拆分为 16 或 4，到此，我们根据官方文档、硬件架构做出了猜想，根据下面两张图中的实验数据，我们将对这个猜想进行验证。图3-8 左侧为按照开启与关闭张量核心情况下的半精度混合矩阵运算的加速比进行排序，并记录下实验编号的顺序，然后使用记录下的实验编号的顺序对单精度混合矩阵运算的实验结果进行排序得到右侧图表 (右侧图表并不是按照单精度混合矩阵运算实验的加速比排序，而是根据基于加速比排序后的半精度混合矩阵运算实验得到的实验编号的顺序排序)。两张图中的加速比可分为三段：加速比低于 1、加速比介于 1-2.5 之间和加速比大于 3 的部分，且两图中突变坐标吻合，故可以推测与精度无关、在硬件上张量核心对于输入操作数的形状较为敏感。

图3-9 中根据实验中输入矩阵共享的维度 k 对加速比进行着色。可见无法被 8 整除的测试样例的加速比较低，接近 1；而能够被 8 整除的测试样例大部分都能被张量核心有效地加速；而随着 K 值的上升，加速比也呈现上升趋势。结合 GPU 实际在线程调度中的特征，以及中间代码 (PTX) 中给出的子矩阵分割形状参数，如图3-10 所示 (图中 wmma 指令为跨线程束矩阵乘加，而 mma 则非跨线程束)，在实际使用中应尽量确保输入矩阵共享的维度 K 能够被 8 整除，且最好为 32 的倍数；且尽量不要使用太“扁”或太“长”的矩阵作为输入。

由于以上评估进行时 GPU 均未到达满载，故使用 CUTLASS 库评估了 GPU 满载时的矩阵乘加性能，如图3-3 所示，可见在 CUTLASS 中以半精度浮点进行运算，开启张量核心 (hgemm-wmma) 后性能相比未开启张量核心 (hgemm) 提升了约 3 倍，与前文实验中相符；而其输入矩阵数据分布方式对性能有一定影响，总体来看两矩阵均以行主元素存储的情况下性能较强 (nt 代表矩阵 A 不进行转置，而矩阵 B 进行转置，然而由于矩阵乘法运算的特征，矩阵 B 转置表示其本身存储方式仍是行主元素存储；而 tn 则代表 A 和 B 都以列主元素存储)，而两矩阵均以列主元素存储时性能较弱。在实验平台中的 NVIDIA GeForce RTX 2080TI 搭载的 TU102 核心中有 68 个流多处理器单元 [47]，结合之前单个流处理器单元测试的性能，大约

API calls:	79.29%	186.156s	322	578.12ms	5.6000us	12.6023s	cudaDeviceSynchronize
	19.41%	45.5659s	811	56.185ms	500ns	2.51149s	cudaFree
	0.78%	1.83506s	805	2.2796ms	4.8000us	44.113ms	cudaMalloc
	0.24%	557.25ms	64961	8.5780us	6.1000us	120.00us	cudaLaunchKernel
	0.18%	433.69ms	27268	15.904us	5.3000us	101.20us	cudaEventQuery
	0.06%	130.15ms	27268	4.7730us	4.2000us	82.800us	cudaMemsetAsync
	0.01%	29.273ms	29273	1.0000us	800ns	7.4000us	cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags
	0.01%	28.891ms	27268	1.0590us	900ns	83.700us	cudaEventRecord
	0.01%	24.068ms	1	24.068ms	24.068ms	24.068ms	cuModuleUnload
	0.01%	15.551ms	64810	239ns	100ns	70.800us	cudaGetLastError
	0.00%	2.6583ms	480	5.5380us	3.7000us	14.900us	cudaFuncGetAttributes
	0.00%	749.70us	282	2.6580us	100ns	138.20us	cuDeviceGetAttribute
	0.00%	674.00us	482	1.3980us	600ns	8.1000us	cudaGetDevice
	0.00%	569.20us	491	1.1590us	400ns	6.9000us	cudaDeviceGetAttribute
	0.00%	326.40us	960	340ns	100ns	5.9000us	cudaPeekAtLastError
	0.00%	153.70us	1	153.70us	153.70us	153.70us	cudaGetDeviceProperties
	0.00%	70.900us	76	932ns	600ns	4.3000us	cudaFuncSetAttribute
	0.00%	58.900us	3	19.633us	17.800us	22.600us	cuDeviceTotalMem
	0.00%	27.300us	1	27.300us	27.300us	27.300us	cudaMemcpy
	0.00%	15.800us	16	987ns	700ns	4.0000us	cudaEventCreateWithFlags
	0.00%	15.400us	1	15.400us	15.400us	15.400us	cudaThreadSynchronize
	0.00%	13.600us	16	850ns	600ns	2.9000us	cudaEventDestroy
	0.00%	10.500us	5	2.1000us	300ns	7.5000us	cuDeviceGetCount
	0.00%	3.4000us	4	850ns	400ns	1.7000us	cuDeviceGet
	0.00%	2.5000us	3	833ns	600ns	1.1000us	cuDeviceGetName
	0.00%	1.9000us	2	950ns	900ns	1.0000us	cuInit
	0.00%	1.7000us	2	850ns	800ns	900ns	cuDriverGetVersion
	0.00%	1.1000us	3	366ns	200ns	600ns	cuDeviceGetLuid
	0.00%	1.0000us	3	333ns	200ns	500ns	cuDeviceGetUuid
	0.00%	700ns	1	700ns	700ns	700ns	cuDevicePrimaryCtxRelease

图 3-4 开启张量核心下通用矩阵乘法运算的性能分析 (API Calls)

Figure 3-4 Performance analysis of GEMM with Tensor Core On (API Calls)

表 3-3 GPGPU-SIM 测得的各指令运行所需时钟周期

Table 3-3 Clock cycle the instructions will cost based on GPGPU-SIM

指令	周期

是 60 倍，这与硬件特征亦相符合，该结果将作为基准性能。从该试验得到的结果以及 NVIDIA 官方文档的资料，正如前文提到过，张量核心的计算最小粒度是 4×4 然而在调度时仍然是以 16×16 为单位，且两个线程束也被组织到一起进行执行；我们可以提出合理地猜想，在下一代架构的硬件中（安培架构），将会出现规模更大、跨越更多个线程束的矩阵乘加的指令集，以提高并行度进而进一步增加计算密集型任务的性能；当然，这样的指令也会在线程束之间的同步带来挑战。

3.2.1.2 矩阵乘法运算 在新架构以矩阵乘加运算进行深度学习性能评估之前，大部分评估都采用单纯的矩阵乘法运算进行。自 CUDA 6.0 开始，NVIDIA 推出了优化线性代数运算的 cuBLAS 库，而每次硬件架构更新，cuBLAS 库均会针对新硬件做出改进、优化。最新的 CUDA SDK 10.1 则推出了 cuBLAS 轻量级库，在生成的应用程序大小方面做出了明显优化，性能方面则并无太多优化；由于本节旨在评估不使用张量核心的情况下，使用和不使用 cuBLAS 库的性能，而最新的 cuBLAS 库均针对张量核心进行优化，故本节中使用的 cuBLAS 库是基于 CUDA SDK 8.0。本节将考察使用和不使用 CUDA SDK 8.0 中的 cuBLAS 库的情况下使用 GPU 进行矩阵乘法运算的性能。

I 实验结果 图3-11 显示了使用和不使用 CUDA SDK 8.0 中 cuBLAS 库进行矩阵乘法运算的性能。随着数据规模的增长，基于 cuBLAS 库的应用的单精度浮点矩阵乘法运算性能稳定在 13TFlops 左右；而不适用 cuBLAS 库的应用尽管不明显，其单精度浮点矩阵运算性能仍随着数据规模的增长而增长，最终稳定在 0.13TFlops 左右。实验中使用的数据集并未如上一节中测试张量核心时的数据，分为能否被 8 整除，而是全部随机。实验结果发现在张量核心以及相关指令尚未出现时，使用 cuBLAS 库时矩阵形状并不

API calls:	91.40%	543.509s	322	1.68792s	5.9000us	47.8398s	cudaDeviceSynchronize
	7.98%	47.4476s	811	58.505ms	500ns	2.53135s	cudaFree
	0.31%	1.83833s	805	2.2836ms	4.8000us	44.561ms	cudaMalloc
	0.14%	803.25ms	40501	19.832us	5.3000us	87.000us	cudaEventQuery
	0.12%	693.12ms	64961	10.669us	6.0000us	140.00us	cudaLaunchKernel
	0.04%	230.83ms	40501	5.6990us	4.1000us	101.40us	cudaMemsetAsync
	0.01%	56.140ms	40501	1.3860us	1.0000us	94.700us	cudaEventRecord
	0.01%	36.334ms	30877	1.1760us	700ns	73.000us	cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags
	0.00%	24.075ms	1	24.075ms	24.075ms	24.075ms	cuModuleUnload
	0.00%	17.964ms	64810	2.277ns	100ns	72.800us	cudaGetLastError
	0.00%	2.9162ms	480	6.0750us	3.7000us	13.600us	cudaFuncGetAttributes
	0.00%	680.80us	482	1.4120us	600ns	6.1000us	cudaGetDevice
	0.00%	645.20us	282	2.2870us	100ns	115.60us	cuDeviceGetAttribute
	0.00%	597.20us	491	1.2160us	400ns	5.5000us	cudaDeviceGetAttribute
	0.00%	323.10us	960	336ns	100ns	4.8000us	cudaPeekAtLastError
	0.00%	168.80us	1	168.80us	168.80us	168.80us	cudaGetDeviceProperties
	0.00%	99.500us	3	33.166us	13.300us	69.800us	cuDeviceTotalMem
	0.00%	87.300us	76	1.1480us	600ns	18.300us	cudaFuncSetAttribute
	0.00%	28.700us	1	28.700us	28.700us	28.700us	cudaMemcpy
	0.00%	16.300us	16	1.0180us	700ns	3.9000us	cudaEventCreateWithFlags
	0.00%	13.800us	1	13.800us	13.800us	13.800us	cudaThreadSynchronize
	0.00%	13.600us	16	850ns	600ns	2.8000us	cudaEventDestroy
	0.00%	6.7000us	5	1.3400us	300ns	5.0000us	cuDeviceGetCount
	0.00%	2.1000us	2	1.0500us	1.0000us	1.1000us	cuDriverGetVersion
	0.00%	2.0000us	3	666ns	600ns	800ns	cuDeviceGetName
	0.00%	1.9000us	4	475ns	300ns	800ns	cuDeviceGet
	0.00%	1.8000us	2	900ns	800ns	1.0000us	cuInit
	0.00%	1.0000us	3	333ns	300ns	400ns	cuDeviceGetUuid
	0.00%	800ns	3	266ns	200ns	400ns	cuDeviceGetLuid
	0.00%	700ns	1	700ns	700ns	700ns	cuDevicePrimaryCtxRelease

图 3-5 关闭张量核心下通用矩阵乘法运算的性能分析 (API Calls)

Figure 3-5 Performance analysis of GEMM with Tensor Core Off (API Calls)

Statistics for ETW Events							
Provider ID	Opcode	Version	# Events	Total Size	Average Size Per Event	Percentage of Total Events Sizes (%)	
Event Trace	Info	2	1	576	576.0	0.00	
Event Trace	Extension	2	3	172	57.3	0.00	
Event Trace	80	2	1	80	80.0	0.00	
Event Trace	32	2	2	104	52.0	0.00	
Process	DoStart	4	227	77083	339.6	0.45	
Thread	DoStart	3	3797	417388	109.9	2.45	
Image	DoStart	3	14492	2851176	196.7	16.76	
Thread	Ready Thread	2	98208	2356992	24.0	13.85	
Thread	Context Switch	4	188786	7551620	40.0	44.39	
Thread	Start	3	90	9540	106.0	0.06	
Event Trace	8	2	2	32	16.0	0.00	
Image	Load Image	3	276	57690	209.7	0.34	
Image	End	3	370	75750	204.7	0.45	
Thread	66	2	71	2272	32.0	0.01	
Thread	67	2	71	2272	32.0	0.01	
Thread	68	2	2273	72736	32.0	0.43	
Process	Start	4	5	1659	331.8	0.01	
Thread	End	3	154	16324	108.0	0.10	
Process	11	2	9	324	36.0	0.00	
Process	End	4	9	3277	364.1	0.02	
Hw Config	Video Configuration	2	4	10464	2818.0	0.06	
Hw Config	28	2	1	40	40.0	0.00	
Hw Config	MIC Configuration	2	7	1956	279.4	0.01	
Hw Config	Physical Disk Configuration	2	3	1800	600.0	0.01	
Hw Config	Logical Disk Configuration	2	5	720	144.0	0.00	
Hw Config	31	2	5	1300	260.0	0.01	
Hw Config	FNU Device Info	5	307	69376	291.1	0.53	
Hw Config	IRQ	3	38	6402	168.5	0.04	
Hw Config	Active Services	3	261	40644	155.7	0.24	
Hw Config	CPU Configuration	3	1	860	860.0	0.01	
Hw Config	35	2	56	25760	460.0	0.15	
Hw Config	ACPI Configuration	2	1	40	40.0	0.00	
Hw Config	Platform	2	1	156	156.0	0.00	
Hw Config	33	2	1	48	48.0	0.00	
Hw Config	34	2	1	56	56.0	0.00	
Hw Config	29	2	1	36	36.0	0.00	
Hw Config	30	2	1	48	48.0	0.00	
Thread	DoEnd	3	3732	410498	110.0	2.41	
Process	DoEnd	4	223	75466	338.4	0.44	
Image	Kernel Base	2	1	24	24.0	0.00	
Image	DoEnd	3	14396	2834418	196.9	16.66	
Process	Performance Defunct	5	105	15313	145.8	0.09	
TOTAL			328000	17012591	51.9		

图 3-6 开启张量核心下通用矩阵乘法运算的性能分析 (上下文切换)

Figure 3-6 Performance analysis of GEMM with Tensor Core On (Context Switch)

Statistics for ETW Events						
Provider ID	Opcode	Version	# Events	Total Size	Average Size Per Event	Percentage of Total Events Sizes (%)
Event Trace	Info	2	1	576	576.0	0.00
Event Trace	Extension	2	3	172	57.3	0.00
Event Trace	80	2	1	80	80.0	0.00
Event Trace	32	2	2	104	52.0	0.00
Process	DoStart	4	216	66629	308.5	0.30
Thread	DoStart	3	3811	411802	108.1	1.88
Thread	Ready Thread	2	150585	3614040	24.0	16.49
Thread	Context Switch	4	287743	11509720	40.0	52.52
Image	DoStart	3	13537	2677168	197.8	12.22
Event Trace	8	2	2	32	16.0	0.00
Thread	Start	3	143	15158	106.0	0.07
Image	Load Image	3	400	82200	205.5	0.38
Image	End	3	494	100332	203.1	0.46
Thread	66	2	116	3712	32.0	0.02
Thread	67	2	116	3712	32.0	0.02
Thread	68	2	2451	78432	32.0	0.36
Process	Start	4	8	2400	300.0	0.01
Thread	End	3	299	31694	106.0	0.14
Process	11	2	10	360	36.0	0.00
Process	End	4	10	2814	281.4	0.01
Hw Config	Video Configuration	2	4	10464	2616.0	0.05
Hw Config	28	2	1	40	40.0	0.00
Hw Config	NIC Configuration	2	7	1956	279.4	0.01
Hw Config	Physical Disk Configuration	2	3	1800	600.0	0.01
Hw Config	Logical Disk Configuration	2	5	720	144.0	0.00
Hw Config	31	2	5	1300	260.0	0.01
Hw Config	FDO Device Info	5	307	69376	291.1	0.41
Hw Config	IRQ	3	38	6402	168.5	0.03
Hw Config	Active Services	3	261	40666	155.8	0.19
Hw Config	CPU Configuration	3	1	660	660.0	0.00
Hw Config	35	2	56	25760	460.0	0.12
Hw Config	ACPI Configuration	2	1	40	40.0	0.00
Hw Config	Platform	2	1	156	156.0	0.00
Hw Config	33	2	1	48	48.0	0.00
Hw Config	34	2	1	56	56.0	0.00
Hw Config	29	2	1	36	36.0	0.00
Hw Config	30	2	1	48	48.0	0.00
Thread	DoEnd	3	3655	395266	108.1	1.80
Process	DoEnd	4	214	66183	309.3	0.30
Image	Kernel Base	2	1	24	24.0	0.00
Image	DoEnd	3	13441	2660138	197.9	12.14
Process	Performance Defunct	5	87	12835	147.5	0.06
TOTAL			478040	21915311	45.8	

图 3-7 关闭张量核心下通用矩阵乘法运算的性能分析 (上下文切换)

Figure 3-7 Performance analysis of GEMM with Tensor Core Off (Context Switch)

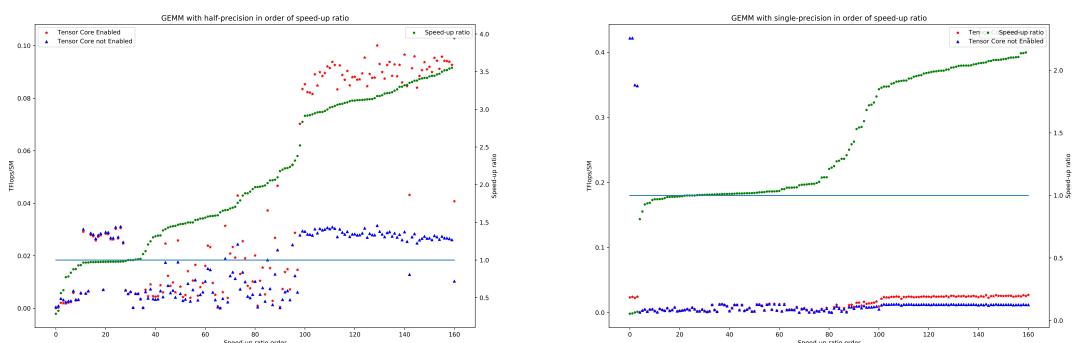


图 3-8 半精度/单精度 GEMM 性能 (按加速比排序)

Figure 3-8 Performance of GEMM at Half and Single (Sorted by speed-up ratio)

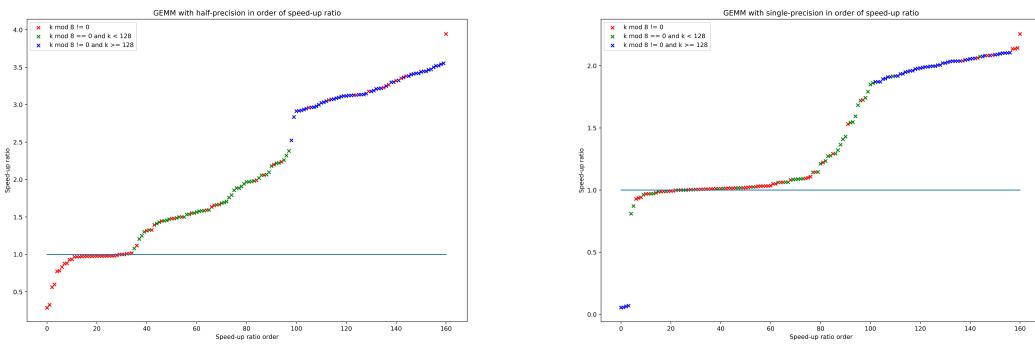


图 3-9 半精度/单精度 GEMM 性能 (按加速比排序, 根据维度 K 特征着色)

Figure 3-9 Performance of GEMM at Half and Single (Sorted by speed-up ratio, colored with feature of K)

Instruction	Shape	Data-type	PTX Version
wmma	.m16n16k16	INT8/FP16/FP32	6.0 or later
wmma	.m8n32k16/.m32n8k16	INT8/FP16/FP32	6.1 or later
wmma	.m8n8k32	INT4	6.3 or later
wmma	.m8n8k128	One bit	6.3 or later
mma	.m8n8k4	FP16/FP32	6.4 or later

图 3-10 官方文档给出的 wmma 指令支持的分割尺寸 [42]

Figure 3-10 Matrix size supported in wmma instruction based on official documentation

会对性能造成显著影响。

II 结果分析 需要注意的是, 在实验中不使用 cuBLAS 库的矩阵乘法是基于 GPU 直接进行数值计算的, 性能较差的同时支持的数据量也不够理想(因系统、硬件和驱动的原因, CUDA 程序的一个内核在 Windows 10 操作系统下运行超过一定时间即会抛出超时异常, 而不使用 cuBLAS、采用直接计算的方法计算矩阵乘积速度较慢, 导致大数据量下程序超时)。且由于内存分配模式的原因, 在直接计算中, 无法被 32 整除的维数需要被填充为 32 的倍数, 在某些情况下会造成大量的内存浪费。故考虑到性能、程序健壮性等因素, 在不支持张量核心的设备上进行矩阵乘法运算应尽可能使用 cuBLAS 库。

3.2.1.3 卷积运算 在深度学习中, 卷积作为提取特征、过滤图像噪声的一种方式, 无疑是从业者最为熟悉、存在于最多应用中的计算。且在图像处理中, 卷积运算也是非常重要的一环。计算卷积有若干方法, 本节将使用 NVIDIA 官方发布的 Benchmark 样例评估这些不同方法在不同情况下的性能。当然, 由于还存在解卷积运算, 故精度也在本实验考虑范围之内, 以满足不同对速度和精度的取舍。

I 实验结果 本节的实验涉及如下几种卷积计算方法: 基于快速傅里叶变换的卷积; 基于通用矩阵乘法的卷积和基于传统的直接计算的卷积 (cudnn 中将这一方法命名为 Conv_Direct), 即分别计算输出图像每一个单元的值。而这三种方法在 CUDA 的实现中也有差异, 如表3-4 所示。

其中, 访问纹理内存时使用传统的 OpenCL/GL 中的实现方式。因纹理内存的访存、缓存方式与一般存储系统不同, 在测试时分为行卷积和列卷积。实验中卷积核步进、图像填充 (padding) 均不变, 性能采

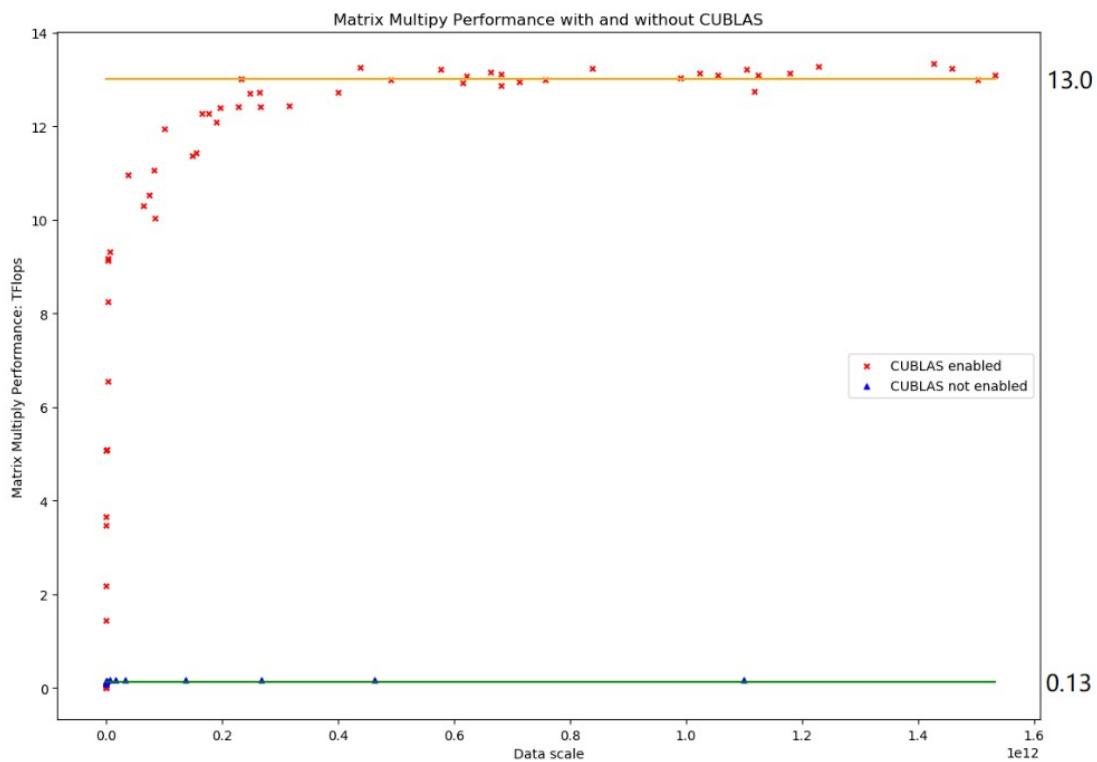


图 3-11 使用和不使用 cuBLAS 库时的矩阵乘法运算性能

Figure 3-11 Performance of Matrix Multiply with and without cuBLAS library

表 3-4 实验中的几种卷积计算方式

Table 3-4 Methods to calculate convolution

卷积计算方式	描述
基于快速傅里叶变换 (FFT) 的 CUDA 内建的卷积	使用 CUDA 提供的基于 FFT 的卷积库
基于快速傅里叶变换 (FFT) 的自定义卷积	使用 CUDA 提供的 FFT 库实现卷积
基于通用矩阵乘法 (GEMM) 的 CUDA 内建的卷积	使用 CUDA 提供的利用张量核心的 API
直接卷积 (全局内存, Global Memory)	直接计算每一个单元的值, 基于全局内存
直接卷积 (纹理内存, Texture Memory)	直接计算每一个单元的值, 基于纹理内存

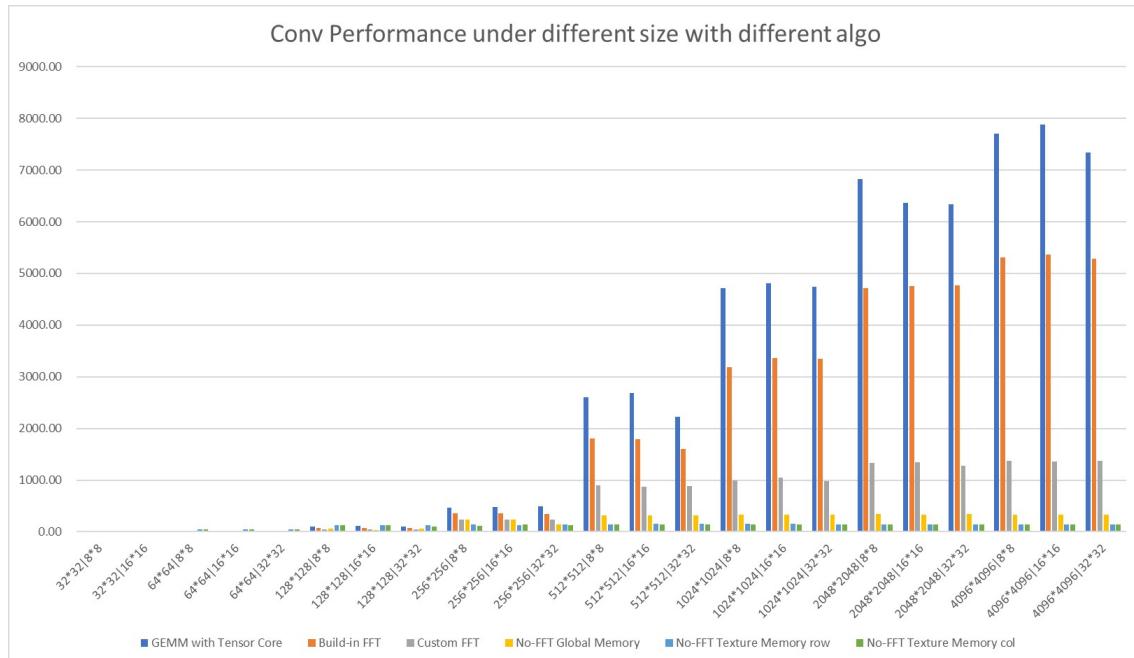


图 3-12 使用不同计算方法的卷积性能

Figure 3-12 Performance of convolution based on different algorithm

用兆像素每秒 (MPix/s) 计算。

II 结果分析

3.2.1.4 神经网络推理

I 实验结果

II 结果分析

3.2.2 基于 CUDA 源码的应用

3.2.2.1 卷积神经网络

I 实验结果

II 结果分析

3.2.2.2 并行支持向量机

I 实验结果

II 结果分析

3.2.3 基于 TensorFlow 框架的应用

I 实验结果

II 结果分析

第四章 总结与展望

本文主要通过自底向上的方式对 Nvidia 最近发布的新架构硬件(伏特、图灵架构)在机器学习应用中的性能提升进行了评估，其中在新架构中新加人的张量核心(Tensor Core)是本文考察的重点，并根据评估结果给出了编程建议，以及对于下一代硬件的一些合理设想。

在正式评估之前，本文对基于 CUDA 的 GPU 编程模型进行了简要介绍，其中包含 CUDA 应用程序的编写步骤、调用方式、内存模型等，同时还对中间层的 PTX 代码进行了简要介绍，这些知识对于后文性能分析部分有较大的帮助。接下来本文便从三个层级对基于 CUDA 的机器学习应用在新架构硬件上的性能提升幅度进行了评估。

首先，本文涉及的最低层面便是用途单一、专为评估绝对性能设计的简单应用进行基准测试，这些应用大部分是 Nvidia 官方发布的测试用例。这些测试用例涵盖混合矩阵运算(GEMM)、矩阵乘法、卷积核神经网络推理。根据实验得到的结果，在混合矩阵运算(GEMM)方面，新架构硬件能在操作数形状、尺寸与硬件参数、调用特征较为匹配的情况下取得大幅度的性能提升，而这些显著的性能提升是采用“用精度换速度”的策略，计算时数据精度均为 FP16/INT8 等低于传统的精度；然而在不匹配的情况下，性能下降极为明显。在矩阵乘法方面，我们评估了 cuBLAS 在新架构上的性能提升，结果是在所有情况下使用 cuBLAS 进行矩阵乘法运算优势都极为明显，且不依赖于操作数的形状、尺寸。在卷积运算方面，基于混合矩阵运算的卷积计算在新架构上相对于原有的基于快速傅里叶变换的计算方法在大规模输入时提升明显，且精度更高；然而在输入规模较小时，使用纹理内存进行直接计算占绝对优势。考虑到目前许多神经网络中的卷积计算的图像规模多为 100-1000 数量级，在该数量级上使用纹理内存进行直接计算的方法性能较强，实际应用中应考虑这种方法。以上三种大多是在网络训练阶段涉及的计算，而在网络推理阶段，本文尝试了一种新的模式，即使用 TensorRT 对训练好的网络结构进行优化并在目标硬件上进行推理。尽管 TensorRT 目前仅能运行于特定硬件，但是根据本文的实验，使用 TensorRT 能为网络推理带来极大的吞吐量提升。

在完成基准测试之后，本文移步基于 CUDA 源码构建的机器学习应用。在该部分中本文分为深度学习应用和传统机器学习应用。深度学习应用选用了结构较为简单的卷积神经网络，而传统机器学习应用选择了支持向量机。在卷积神经网络部分，本文将计算分为前向传播，反向传播更新连接参数，反向传播更新卷积核参数三个部分，分别考察新架构对于这三个部分的提升幅度。实验结果令人意外，除去前向传播中新架构能带来 30%-50% 的提升外，另两个部分中开启新架构甚至会降低性能。其原因为反向传播部分多为梯度计算，能使用混合矩阵计算(GEMM)从而利用到 Tensor Core 的部分较少，而开启 Tensor Core 又会对调度、同步、访存和其他指令的发射带来影响，故反向部分会造成性能下降。而前向传播部分由于卷积核、步进、填充的存在，无法保证每一层操作数的形状都能适用于 Tensor Core，故提升幅度极为有限。值得注意的是，通过将卷积操作更换为第二章中提到的纹理内存方式，总体性能得到了一定的提升。在支持向量机部分，本文则根据支持向量机输入矩阵较为稀疏的特征，分别考察了使用专为稀疏矩阵设计的 API 和使用 Tensor Core 的 API 进行评估，实验结果表明在数据量较大时，特征矩阵会愈稀疏，专为稀疏矩阵设计的 API 性能较好，而数据量较小时，仍然是使用 Tensor Core 的 API 性能较好。

之后，本文对基于 TensorFlow-GPU 框架的应用进行评估。在这一部分我们搭建了一个简单的基于 TensorFlow-GPU 的卷积神经网络，目的在于考察在最贴近真实应用场景时如何尽可能利用新硬件提升性

能。本文从神经网络的超参数、网络结构、卷积计算方式、数据精度和推理等方面进行考察；结果发现增加网络的批大小能带来显著的训练速度提升，但是过大的批会导致网络准确度下降，实际应用中应权衡这两点；而由于输入的图片尺寸较小，本文通过修改 Tensor Flow 源代码，将内建的卷积计算方式更换为使用纹理内存的直接方式后，取得了较为明显的提升且网络准确度仍然维持在较高的水平，然而由于这种方式局限大，且更改源码需要重新编译、安装，这个过程极为麻烦，故实际应用中不推荐对源码进行更改；在数据精度方面，使用 FP16/INT8 代替 FP32 并不会对网络总体准确度带来太大的下降，而在训练速度上提升很明显，实际应用中在准确度要求不高的情况下可以考虑用低精度数据替换；网络结构方面，将卷积核大小改为适合 Tensor Core 计算的形状能给训练速度带来一定提升，但是会极大降低网络准确度，实际应用中不推荐使用。最后，本文使用 TensorRT 对训练好的模型进行推理，在延迟方面有 40

通过以上实验，可以总结出新架构的确能在特定情况下为机器学习中大量存在的矩阵混合运算带来明显的性能提升，从而提升总体机器学习应用的性能，然而目前为止，硬件仍然对输入、结构等较为敏感。且有些情况下仍然有性能更高的传统方式。所以在实际编码时，应根据问题规模、算法、结构、数据分布等方面合适选择不同方法，而不是一味使用新硬件提供的方法。

最后，根据实验中发现的一些问题，本文做出了合理地设想，如计算规模更大、跨越多个线程束执行混合矩阵运算的新指令，以单个线程为粒度的同步机制等；这些设想是否会应验，或者在一定程度上实现，只能交给时间去判断，这里仅仅提出我们的设想供启发。

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]. [S.l.]: NVIDIA Corp., 2017: 14-15.
- [6] KURTH T, TREICHLER S, ROMERO J, et al. Exascale Deep Learning for Climate Analytics[C]. in: 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] STEINKRAU D, SIMARD P Y, BUCK I. Using GPUs for Machine Learning Algorithms[C]. in: Eighth International Conference on Document Analysis and Recognition (ICDAR 2005), 29 August - 1 September 2005, Seoul, Korea. [S.l. : s.n.], 2005: 1115-1119. DOI: 10.1109/ICDAR.2005.251.
- [12] KOMAROV I, DASHTI A, D'SOUZA R. Fast k-NNG construction with GPU-based quick multi-select[J]. CoRR, 2013, abs/1309.5478.
- [13] KEERTHI S S, SHEVADE S K, BHATTACHARYYA C, et al. Improvements to Platt's SMO Algorithm for SVM Classifier Design[J]. Neural Computation, 2001, 13(3): 637-649. DOI: 10.1162/089976601300014493.
- [14] VASIMUDDIN M, CHOCKALINGAM S P, ALURU S. A Parallel Algorithm for Bayesian Network Inference Using Arithmetic Circuits[C]. in: 2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018. [S.l. : s.n.], 2018: 34-43. DOI: 10.1109/IPDPS.2018.00014.
- [15] 邦彦 福. 位置ずれに影響されないパターン認識機構の神経回路モデル — ネオコグニトロン —[J]. 電子情報通信学会論文誌 A, 1979, J62-A(10): 658-665.
- [16] BENGIO Y, LECUN Y, HENDERSON D. Globally Trained Handwritten Word Recognizer Using Spatial Representation, Convolutional Neural Networks, and Hidden Markov Models[C]. in: Advances in Neural Information Processing Systems 6, [7th NIPS Conference, Denver, Colorado, USA, 1993]. [S.l. : s.n.], 1993: 937-944.
- [17] FARHAT N H. Photonic Neural Networks and Learning Machines[J]. IEEE Expert, 1992, 7(5): 63-72. DOI: 10.1109/64.163674.

- [18] BAKHODA A, YUAN G L, FUNG W W L, et al. Analyzing CUDA workloads using a detailed GPU simulator[C]. in: IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009, April 26-28, 2009, Boston, Massachusetts, USA, Proceedings. [S.l. : s.n.], 2009: 163-174. DOI: 10.1109/ISPASS.2009.4919648.
- [19] KHAIRY M, JAIN A, AAMODT T M, et al. Exploring Modern GPU Memory System Design Challenges through Accurate Modeling[J]. CoRR, 2018, abs/1810.07269.
- [20] NVIDIA. NVIDIA Nsight Systems[A]. 2019.
- [21] NARANG S, BAIDU. DeepBench[A]. 2016.
- [22] HE K, ZHANG X, REN S, et al. Deep Residual Learning for Image Recognition[C]. in: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016. [S.l. : s.n.], 2016: 770-778. DOI: 10.1109/CVPR.2016.90.
- [23] HOWARD A G, ZHU M, CHEN B, et al. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications[J]. CoRR, 2017, abs/1704.04861.
- [24] 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>.
- [25] RHU M, SULLIVAN M B, LENG J, et al. A locality-aware memory hierarchy for energy-efficient GPU architectures[C]. in: The 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, Davis, CA, USA, December 7-11, 2013. [S.l. : s.n.], 2013: 86-98. DOI: 10.1145/2540708.2540717.
- [26] COOK S. CUDA Programming: A Developer's Guide to Parallel Computing with GPUs[M]. [S.l.]: Morgan Kaufmann, 2012: 99-102.
- [27] HAKURA Z S, GUPTA A. The Design and Analysis of a Cache Architecture for Texture Mapping[C]. in: Proceedings of the 24th International Symposium on Computer Architecture, Denver, Colorado, USA, June 2-4, 1997. [S.l. : s.n.], 1997: 108-120.
- [28] FLYNN M J. Some Computer Organizations and Their Effectiveness[J]. IEEE Trans. Computers, 1972, 21(9): 948-960. DOI: 10.1109/TC.1972.5009071.
- [29] NVIDIA. Ampere Block Diagram[A]. 2019.
- [30] TIRUMALA A, GIROUS O, NELSON P, et al. Threads-are threads Functional Description, SM Branch ISA and Convergence Barrier Unit[A]. 2015.
- [31] SARKAR S, MITRA S. A Profile Guided Approach to Optimize Branch Divergence While Transforming Applications for GPUs[C]. in: Proceedings of the 8th India Software Engineering Conference, ISEC 2015, Bangalore, India, February 18-20, 2015. [S.l. : s.n.], 2015: 176-185. DOI: 10.1145/2723742.2723760.
- [32] LI H, KUMAR A, TU Y. Performance modeling in CUDA streams - A means for high-throughput data processing[C]. in: 2014 IEEE International Conference on Big Data, Big Data 2014, Washington, DC, USA, October 27-30, 2014. [S.l. : s.n.], 2014: 301-310. DOI: 10.1109/BigData.2014.7004245.

- [33] MUKHERJI S, RAMI A. Speeding Up the Computation of Cross-Correlation Using the FHT and FFT by Page-locking[C]. in: IEEE International Geoscience & Remote Sensing Symposium, IGARSS 2006, July 31 - August 4, 2006, Denver, Colorado, USA, Proceedings. [S.l. : s.n.], 2006: 992-994. DOI: 10.1109/IGARS.2006.255.
- [34] MCCORMARK J. Volta HMMA.884 instruction(s) and GEMM[A]. 2019.
- [35] BEZDANY M, KUMAR N. Ampere SM SCH POR[A]. 2019.
- [36] TensorFlow. Building TensorFlow from source[A]. 2019.
- [37] NVIDIA. CUDA Zone[A]. 2019.
- [38] HARRIS M. An Even Easier Introduction to CUDA[A]. 2017.
- [39] NVIDIA. NVIDIA CUDA Toolkit v10.0.130 Release Notes[A/OL]. 2018. <https://docs.nvidia.com/cuda/archive/10.0/cuda-toolkit-release-notes/index.html>.
- [40] NVIDIA. NVIDIA CUDA Toolkit v9.2.148 Release Notes[A/OL]. 2018. <https://docs.nvidia.com/cuda/archive/9.2/cuda-toolkit-release-notes/index.html>.
- [41] NVIDIA. NVIDIA CUDA Toolkit v10.1.105 Release Notes[A/OL]. 2019. <https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html>.
- [42] NVIDIA. Parallel Thread Execution ISA Version 6.4[A]. 2019.
- [43] NVIDIA. NVIDIA Profiler User's Guide[A/OL]. 2019. <https://docs.nvidia.com/cuda/profiler-users-guide/index.html>.
- [44] NVIDIA. NVIDIA TensorRT, Programmable Inference Accelerator[A/OL]. 2019. <https://developer.nvidia.com/tensorrt>.
- [45] NVIDIA. Deep Learning SDK Documentation: What is TensorRT[A/OL]. 2018. <https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html>.
- [46] NVIDIA. CUDA Toolkit Documentation v9.2.148[A]. 2018.
- [47] TECHPOWERUP. NVIDIA Geforce RTX 2080TI Specs[A/OL]. 2019. <https://www.techpowerup.com/gpu-specs/geforce-rtx-2080-ti.c3305>.

致谢

距离 2015 年 09 月，已经过去了将近四年。

四年前刚刚踏入大学校园的种种仿佛还就在眼前。回忆起曾经高中学业的紧张、择校选专业时的忐忑，再看现在未来已经基本定型的情况，不由心生感慨：能找到自己真正的兴趣并在目前的实习、之后的进一步学习和工作中予以实践，着实是一件幸运的事。

首先，感谢家人的陪伴，感谢父母对于我学业无论是在经济上还是在生活上全力的支持，没有你们的帮助，我甚至没有能力承担学业。有时我会学习到深夜，在大家都已经进入梦乡的时间，你们还会为我煮上一碗面。正是因为你们，我才能完成论文、完成学业。

在大学的前几年，我也迷茫过，在别的同学做项目、打比赛的时候我也焦虑过；但是幸好我被给予了一个延续我高中以来的梦想：研究硬件、系统结构的机会，钱老师的并行计算课程、魏老师的计算机组成课程、王老师的嵌入式原理课程、肖老师的算法课程、陆老师的 C++ 课程、等等……这些课程可谓是我打开了一个新世界的大门，我得以系统地学习我曾感兴趣的知识；也正是因为这门课程，我也确定了本文的主题，确定了研究生的学习方向，确定了目前的实习，以及将来的职业目标。

说到职业，这里不得不提到在英伟达 (Nvidia) 实习时，公司以及同事对我的帮助，正是因为这份实习，让我有机会接触到无数的涉及 GPU 底层架构的文档，让我有机会深入到 GPU 级别的汇编代码进行编程，这些经验、资料对本文的写作带来了极大的帮助，当然，这一切都归功于 Edward 先生愿意给予我这次实习机会，并悉心指导我。

在论文的撰写中，我还得到了许多同学的协助；有同样对并行计算感兴趣的吕同学与我耐心的探讨，有姚同学给我提出的建议，有沈同学与我分享行业最新信息，还有各位一起娱乐的群友们为我带来的欢乐与放松……这些无一不让我在紧张的论文撰写中得以卸下一些压力。当然，不只是大学中的同学们，这里也感谢我自初中以来的同学，也是我的女友的 Vega 姜小姐九年以来的陪伴以及在身心上给予我的支持。

论文总有一天会完成上交，学生生涯总有一天会迎来结束。然而对新知识的探求正是支撑起我们计算机学子前进的基石。不求对世界做出什么改变，不求对人类做出什么贡献，只求在未来的道路里不忘初心、坚守道德、尽力而为、劳逸结合、保持童心、有始有终、乐观对待、做自己想做的事，并且无憾一生。

Arrivederci.