计算机科学
COMPUTER SCIENCE

矩阵乘法的GPU并行计算时耗模型与最优配置方法

雷超, 刘江, 宋佳文

引用本文

雷超, 刘江, 宋佳文. 矩阵乘法的GPU并行计算时耗模型与最优配置方法[J]. 计算机科学, 2024, 51(6A): 230300200-8.

LEI Chao, LIU Jiang, SONG Jiawen. Time Cost Model and Optimal Configuration Method for GPU Parallel Computation of Matrix Multiplication [J]. Computer Science, 2024, 51(6A): 230300200-8.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.230300200

矩阵乘法的GPU并行计算时耗模型与最优配置方法

雷 超[1,2] 刘 江[1,2] 宋佳文[3]
1 中国科学院重庆绿色智能技术研究院 重庆 400714
2 中国科学院大学重庆学院 重庆 400714
3 中南大学航空航天技术研究院 长沙 410017
(leichao@cigit.ac.cn)

摘 要 水平矩阵乘竖直矩阵是科学计算及工程领域中的基本计算之一,很大程度上影响了整个算法的计算效率。GPU并行计算是迄今主流的并行计算方式之一,其底层设计使得GPU非常契合于大规模矩阵计算。迄今已经有许多研究基于GPU并行计算框架,针对矩阵的结构设计、优化矩阵乘法,但尚未有针对水平矩阵乘竖直矩阵的GPU并行算法及优化。此外,GPU核函数配置直接影响计算效率,但迄今针对最优核函数配置的研究极为有限,通常需要研究人员针对具体算法的计算特点启发式地设置。基于GPU的线程、内存模型,设计了一种并行水平矩阵乘竖直矩阵乘法PHVM。数值实验结果表明,在左乘矩阵的水平维度远远大于竖直维度时,PHVM要显著优于NVIDIAcuBLAS库中的通用矩阵乘法。进一步,基于GPU的硬件参数,建立了PHVM运行时间的核函数配置最优化理论模型。数值实验结果表明,该理论模型较为准确地描述了PHVM算法运行时间随核函数配置(网格大小、线程块大小)变换的变化趋势,且模型得出的理论最优核函数配置与实际最优运行核函数配置相符。

关键词:矩阵乘法;GPU;CUDA;核函数配置
中图分类号 TP391.9

Time Cost Model and Optimal Configuration Method for GPU Parallel Computation of Matrix Multiplication

LEI Chao[1,2],LIU Jiang[1,2] and SONG Jiawen[3]
1 Chongqing Institute of Green and Intelligent Technology,Chinese Academy of Sciences,Chongqing 400714,China
2 Chongqing School,University of Chinese Academy of Sciences,Chongqing 400714,China
3 Research Institute of Aerospace Technology,Central South University,Changsha 410017,China

Abstract Horizontal matrix & vertical matrix multiplication(HVM) is one of the fundamental calculations in scientific computing and engineering,as it largely affects the computational efficiency of higher-levet algorithms. GPU parallel computing has become one of the mainstream parallel computing method,and its underlying design makes it highly suitable for large-scale multiplication calculations. Numerous studies have focused on designing matrix structures and optimizing matrix multiplication using GPU parallel computing frameworks. However,there has been a lack of GPU parallet algorithms and optimization methods specifically targeting HVM. Furthermore,the configuration of GPU kernel functions directly affects computational efficiency,but studies on the optimal configuration of kernel functions have been extremely limited,typically requiring researchers to heuristically set them based on the specific computational characteristics of the algorithm. This paper designs a parallel HVM algorithm,PHVM,based on the GPU's thread and memory model. The numerical experimental results show that when the horizontal dimension of the left matrix is much larger than the vertical dimension,PHVM significantly outperforms the general matrix multiplication in the NVIDIA cuBLAS library. Furthermore,this paper establishes an optimal theoretical model for kernel function configuration of PHVM runtime based on GPU hardware parameters. The numerical experimental results indicates that this theoretical model accurately reflects the trend of changes in PHVM algorithm runtime with kernel function configuration(grid size and thread block size) variations.

Keywords Matrix multiplication,GPU,CUDA,Kernel function configuration

---

基金项目:国家重点研发计划(2018YFC0116704);中国科学院科技服务网络计划区域重点项目(KFJ-STS-QYZD-2021-01-001);中南大学课题(大规模稀疏线性方程组并行加速求解研究);雷达资料同化关键技术及数值预报客观订正技术研究(E190600801)
This work was supported by the National Key R&D Program of China(2018YFC0116704),Science and Technology Service Network Initiative(KFJ-STS-QYZD-2021-01-001),Central South University Project(Research on Parallel and Accelerated Solution of Large-scale Sparse Linear Equations) and Rsearch on Key Techniques of Radar Data Assimilation and Objective Correction of Numerical Forecast(E190600801).
通信作者:刘江(liujiang@cigit.ac.cn)

1 概述

1.1 水平矩阵与竖直矩阵乘法概述

本文聚焦于水平矩阵-竖直矩阵乘法[1-3]。一般地，考虑水平矩阵乘竖直矩阵 $\boldsymbol{E}\times\boldsymbol{F}$，其中 $\boldsymbol{E}\in\mathbb{R}^{M\times K}$，$\boldsymbol{F}\in\mathbb{R}^{K\times N}$，$M\ll K$，$N\ll K$，如图 1 所示。

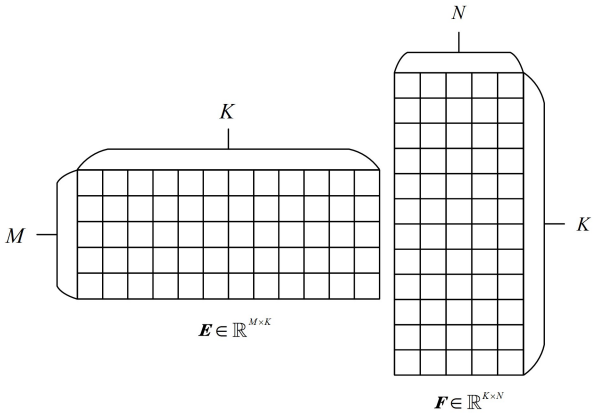

图 1 水平矩阵乘竖直矩阵

Fig. 1 Horizontal matrix multiply vertical matrix

水平矩阵与竖直矩阵乘法是科学计算及工程领域中的基本计算之一。在图像压缩领域中，此类乘法被用于离散余弦变换(Discrete Cosine Transform，DCT)算法压缩图像[1]。DCT 是一种傅里叶变换类型，将图像转换为一组频率以更有效地存储。其首先将图像分成小块，即水平矩阵，然后将每个块乘以竖直矩阵以生成 DCT 系数。在信号处理中，水平矩阵乘法用于信号处理应用，如音频和视频压缩[2]：信号被分成小块，然后每个块被乘以一个水平矩阵来生成离散余弦或正弦变换系数。此外，此类乘法也是机器学习及深度学习的神经网络计算中的重要步骤[3]：输入数据首先通过卷积层进行处理，该层使用此类乘法从输入图像或文本中提取特征。

针对通用矩阵乘法，Strassen[4]等提出了 Strassen 矩阵乘法，时间复杂度为 $O(n^{\log_2 7})$；Coppersmith[5]等提出了基于算术级数的矩阵乘法，将时间复杂度降至 $O(n^{2.376})$。进一步，利用相乘矩阵的结构性质，Davis[6]针对稀疏矩阵总结了稀疏矩阵向量乘算法。Casaent 等[7]设计了一种新的时空积分光学线性代数处理器以加速处理带状矩阵乘法。基于矩阵分块的策略，Challacombe 等[8]采用数据并行的消息传递方式提出了一种通用并行稀疏分块矩阵乘法。Rubensson 等[9]利用四叉树数据结构，提出了一种在分布式内存集群上进行并行稀疏分块矩阵乘法的方法。但是迄今未见针对水平矩阵-竖直矩阵乘法的研究。

1.2 GPU 并行计算概述

相较于中央处理器 CPU，GPU 专注于并行计算，适用于涉及大量数据并行性的计算任务，即对大量数据执行相同操作，例如矩阵乘法、图像处理和科学计算等。目前，在 AI 领域的大规模计算均使用 GPU 加速[10]。更重要的是，在能耗方面，同等计算效率的 GPU 比 CPU 更低[11]。NVIDIA 公司于 2006 年发布了统一计算架构 CUDA[12-13]。CUDA 是建立在 NVIDIA 的图形处理器 GPU(Graphics Processing Unit)上的一个通用并行计算平台和编程模型，基于 CUDA 编程可以利用 GPUs 的并行计算引擎更加高效地解决比较复杂的计算问题。因此，对矩阵乘法等算法在 GPU 上并行实现及优化的研究也备受国内外学者关注[12-14]。Fatahalian 等[14]解释了用 GPU 并行处理矩阵乘法的一般逻辑；Herault 等设计了基于 PaRSEC 的多 GPU 加速分布式内存平台[15]；NVIDIA 给出了性能优异的通用 GPU 矩阵操作算子库 cuBLAS[16]，囊括了 152 种基础线性代数算子的 GPU 优化实现。但是 cuBLAS 是闭源库，研究人员及开发人员无法根据实际需要修改 cuBLAS 的源码。

进一步，GPU 核函数配置(网格大小、线程块大小)直接影响计算效率。线程块是 CUDA 程序并行计算的基本单元，如果线程块过小，则可能无法充分利用 GPU 的并行计算能力，从而导致计算效率低下；如果线程块过大，则可能会导致 GPU 内存不足，从而影响计算性能。网格是由多个线程块组成的，因此它的大小可以影响到程序的并行度。如果网格过小，可能无法充分利用 GPU 的计算资源，导致计算性能低下。如果网格过大，可能会使 GPU 内存不足，从而影响程序的执行效率。目前有许多针对 GPU 的程序优化研究[15-17]。Wang 等[17]提出了一种基于 GPU 的并行计算性能分析模型，帮助程序员从指令流水线、共享存储器访存、全局存储器访存等方面优化程序。Yin 等[18]对 GPU 的存储层次结构进行了模拟，找出了 GPU 矩阵乘法算法中的流多处理器数量与存储控制器数量之间的最佳配置关系。DALTON 等[19]针对稀疏矩阵乘法的 GPU 实现提出了带宽优化策略。但迄今针对最优核函数配置的研究极为有限，通常需要研究人员针对具体算法的计算特点启发式地设置。

1.3 本文工作及贡献

针对上述章节中描述的两个问题，本文基于 GPU 设计了一种新的水平矩阵——竖直矩阵乘法算法 PHVM，并基于 GPU 的硬件参数给出了 PHVM 的核函数配置最优化理论模型。具体地，本文的主要贡献如下。

(1)基于 GPU 的线程、内存模型，针对水平矩阵乘竖直矩阵设计了并行矩阵乘法算法(Parallel Horizontal & Vertical Matrix Multiplication Method，PHVM)。数值实验表明，当左乘矩阵的水平维度远远大于竖直维度时，PHVM 要显著优于 NVIDIAcuBLAS 中的通用矩阵乘法 cublasSgemm。

(2)基于 GPU 的硬件参数，本文给出了 PHVM 的核函数配置最优化理论模型。模型以 GPU 的硬件参数为常数，矩阵尺寸、网格大小与线程块大小为参数，最小化算法理论运行时间为优化目标。数值实验表明，该理论模型较准确地反映了程序运行的实际状态，描述了算法运行时间随网格大小、线程块大小变化的趋势。当研究人员在已知参数的 GPU 上部署计算 PHVM 时，可以根据该理论模型自动计算出最优的核函数参数配置组合。该理论模型具有一般意义，研究人员可按照 PHVM 时耗理论模型的建立流程构建其他算法的 GPU 时耗理论模型，从而减少测试其 GPU 最优核函数配置的准备时间。

2 基于 GPU 的并行水平矩阵乘竖直矩阵乘法 PHVM

2.1 算法提出

一般地，考虑水平矩阵乘竖直矩阵 $\boldsymbol{E}\times\boldsymbol{F}$(见图 1)，其中

$E\in\mathbb{R}^{M\times K}$，$F\in\mathbb{R}^{K\times N}$，$M\ll K$，$N\ll K$。为了方便讨论，令 $M=N$。此类计算的特点是结果矩阵中元素数量少（M^2），计算每个元素所需的计算量大（计算长度为 K 的向量内积）。针对此类计算，我们考虑以下高效并行计算方式。

算法 1　基本并行水平矩阵乘竖直矩阵算法

输入：(**E**,**F**,M)

输出：**R**

1. for i←[1,M] in Parallel do：
2.　for j←[1,M] in Parallel do：
3.　**R**[i][j] ← DOT(**E**. row(i),**F**. COL(j))
4.　end for
5. end for

由于 $M\ll K$，所以大部分计算消耗在内积的计算上。并行内积的基本思想为分治法[20]：在当前计算步，将一半的元素并行加到另一半元素上，从而将问题规模减小一半。如果 GPU 线程数足够多，且不考虑 GPU 线程间的通信花费，上述算法可在 O(log$K$)的时间复杂度完成计算。但是 GPU 的可激活线程是有上限的，以 GPU A100-PCIE-40GB 为例，可激活线程总数为 221 184；并且 GPU 线程间通信的花费也不可忽略，尤其是线程块与线程块之间通过全局内存进行通信，相对耗时较长。为了尽量减少访问全局内存，基于分治思想与 CUDA 硬件配置，本章给出基本并行内积算法，其计算流程如图 2 所示。

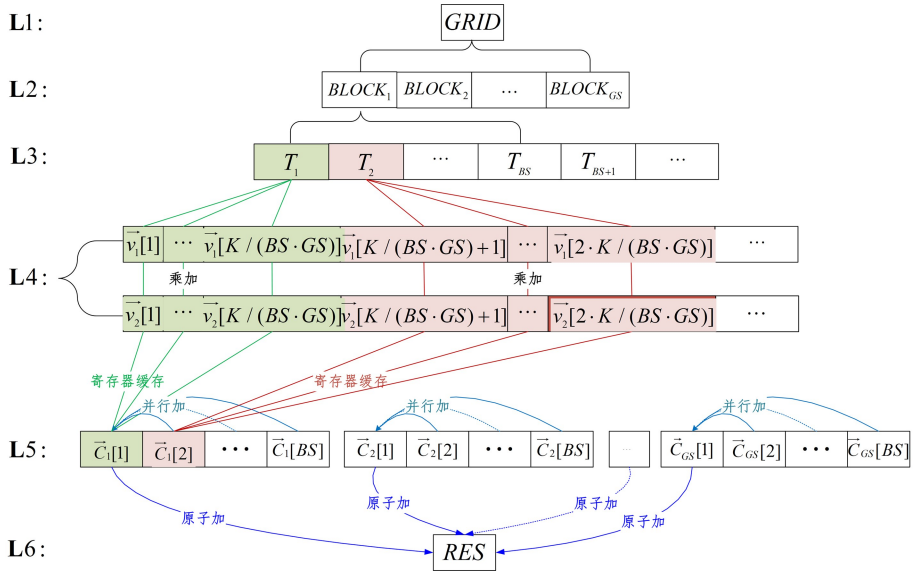

图 2　基本并行内积算法示意图

Fig. 2　Schematic diagram of basic parallel inner product algorithm

下面从上往下分析该算法的六层要素。

L1：*GRID* 为线程网格。

L2：$BLOC\,K_i$ 为第 i 个线程块，$i\in[1,GS]$，GS 为线程网格容量，即线程块的个数。

L3：T_i 为第 i 个线程，$i\in[1,BS\cdot GS]$，BS 为线程块容量，即单个线程块中线程的个数。T_i 为线程块 $BLOC\,K_{\frac{i}{BS}+1}$ 中的线程。

L4：$\vec{v}_1$ 与 $\vec{v}_2$ 为计算对象，存储于全局内存中。T_i 负责将 $\vec{v}_1$，$\vec{v}_2$ 的数据段 $\left[(i-1)\cdot\dfrac{K}{BS\cdot GS}+1:i\cdot\dfrac{K}{BS\cdot GS}\right]$ 相乘相加，累计在其寄存器中。

L5：在 GPU 中，同一个线程块中的线程通过共享内存进行通信，所以需要将上一步骤中寄存器的缓存结果放入共享内存中。$\vec{C}_i$ 存储于 $BLOCK_i$ 中的共享内存，其中 $i\in[1,\cdots,BS]$。此步首先将线程 T_i 的累积结果存入 $\vec{C}_{\frac{i}{BS}+1}[i\%BS+1]$ 中，其中 $i\in[1,BS\cdot GS]$，然后使用并行加算法将 $\vec{C}_i[j]$ 并入 $\vec{C}_i[1]$，其中 $i\in[1,\cdots,GS]$，$j\in[1,\cdots,BS]$。

L6：*RES* 为最终结果，其存储于全局内存中。此步将上步中所有线程块的首位元素原子累加进 *RES*，即 $RES+=\vec{C}_i[1]$，其中 $i\in[1,GS]$。

由于 CUDA 的执行方式是单指令多线程，最小单位是线程束（Warp），所以在该算法的 L4 步骤中，线程束内部的连续线程在同一访存指令的作用下会访问离散的全局内存地址，故该线程束需访问 $2\cdot WS$ 次全局内存，WS 为线程束中线程的数量。

基于 CUDA 程序的优化准则[21]，考虑合并上述算法 L4 步骤中一个线程束内部的全局内存读取，我们给出合并取内存并行内积算法，其计算流程图如图 3 所示。在该算法中，虽然单个线程负责的计算数据在全局内存中以步长 $Stride=GS\cdot BS$ 离散存储，但是线程束内部的连续线程在同一访存指令的作用下会访问连续的全局内存。CUDA 会将这些取存合并，故单个线程束仅需访问一次全局内存。具体地，更改步骤 L4 为：

$\hat{L4}$：T_i 负责将 $\vec{v}_1$ 与 $\vec{v}_2$ 中对应下标为 $k\cdot(GS\cdot BS)+i$ 的数据对进行先相乘后相加的操作，并将计算结果累计在其寄存器中，其中 $i\in[1,\cdots,GS\cdot BS]$，$k\in\left[1,\cdots,\left\lfloor\dfrac{N-i}{GS\cdot BS}\right\rfloor\right]$。

将基本并行水平矩阵乘竖直矩阵算法（算法 1）中的第 3 行替换为基本并行内积算法（见图 2），得到完整基本并行矩阵乘法算法 BasicMM。将基本并行水平矩阵乘竖直矩阵算法（算法 1）中的第 3 行替换为合并取存内积算法（见图 3），即得到优化后的并行水平矩阵乘竖直矩阵算法 PHVM。

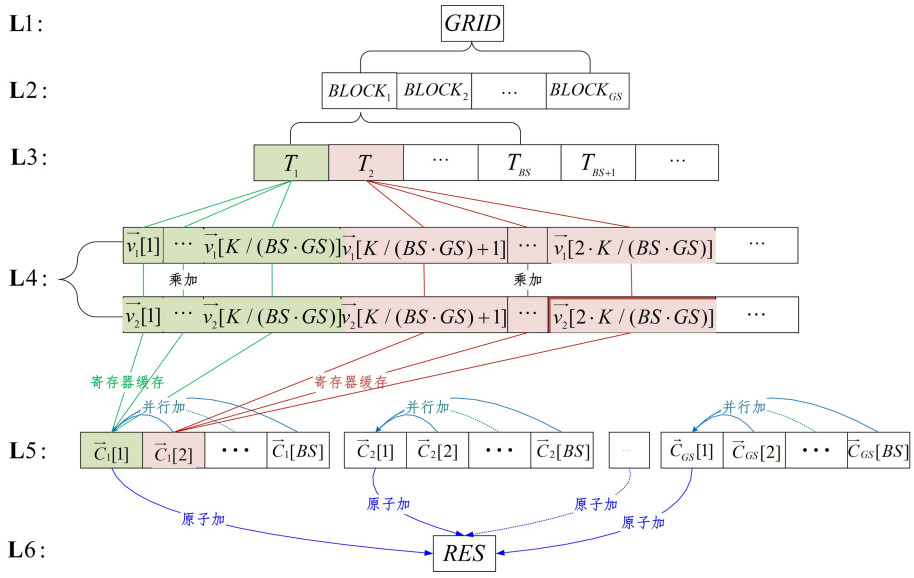

图 3 合并内存读取内积算法

Fig. 3 Merging memory read inner product algorithm

2.2 数值实验

本章节给出具体的数值实验以验证 PHVM 的高效性。在该实验中，针对水平矩阵乘竖直矩阵 $\boldsymbol{E}\times\boldsymbol{F}$，其中 $\boldsymbol{E}\in\mathbb{R}^{M\times K}$，$\boldsymbol{F}\in\mathbb{R}^{K\times N}$，$M\ll K$，$N\ll K$，我们给出了非合并取内存的基本并行矩阵乘法 BasicMM、合并取内存的 PHVM 与 CUDA 的 cuBLAS 库提供的通用矩阵乘法 cublasSgemm 的性能对比，其中 cuBLAS 库为 NVIDIA 针对 GPU 的底层设计实现并优化的基础算子库，具有更新快、通用性强、性能高等优势[19-22]。本实验所用的矩阵均为程序随机生成，矩阵中所有元素服从范围为[−1,1]的均匀分布。表 1 列出了实验所用矩阵组的具体信息。

表 1 随机矩阵对信息

Tabel 1 Stochastic matrix pair information

矩阵对	M	K	N	矩阵元素类型	矩阵元素分布
MTXA	3	5×10^7	3	FP32	U(−1,1)
MTXB	5	3×10^8	5	FP32	U(−1,1)
MTXC	7	2×10^9	7	FP32	U(−1,1)

我们通过 CUDA 并行实现这些方法。实验所用 CPU 型号为 Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz，GPU 型号为 A100-PCIE-40GB，CUDA 版本为 11.1，操作系统为 CentOS Linux release 7.8.2003(Core)，g++编译器使用 g++(GCC) 9.3.0。分别使用 cublasSgemm，PHVM 与 BasicMM 对表 1 中的矩阵对做乘法，得到三者核函数运行时间对比(见图 4−图 6)。图 4、图 5、图 6 分别对应 cublasSgemm，PHVM，BasicMM 算法计算矩阵对 MTXA，MTXB，MTXC 的结果对比，可以看到左乘矩阵的行数越小，PHVM 较于 cublasSgemm 的速度优势越明显。因为针对同一个矩阵对数据，主机内存到设备内存的数据传输量固定，所以我们仅统计核函数的运行时间。

实验数据表明，采取合并取内存策略的 PHVM 运行时间明显少于采取非合并取内存策略的 BasicMM，说明合并内存取全局内存的确是有很大的加速效果的。另外，cublasSgemm 的运行时间展示了 cuBLAS 确实并未做水平矩阵与竖直矩阵乘法的优化：图 5 所用矩阵对与图 6 所用矩阵对的元素数量相当，前者的 M 要小于后者，如果利用了此类结构特征，后者的运行时间理应大于前者(如 PHVM 与 BasicMM 的实验结果所示)，但是 cublasSgemm 的实验结果却相反。这也体现了 cuBLAS 是闭源求解库的弊端，开发人员只能使用其提供的应用程序接口，而对其内部实现细节一概不知，无法针对应用场景做优化。

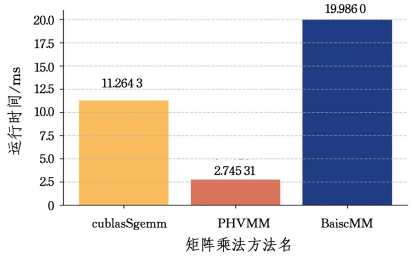

图 4 矩阵对 MTXA 性能对比图

Fig. 4 Performance comparison of MTXA matrix pair

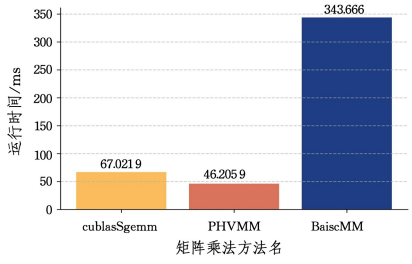

图 5 矩阵对 MTXB 性能对比图

Fig. 5 Performance comparison of MTXA matrix pair

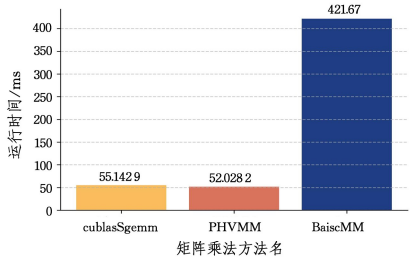

图 6 矩阵对 MTXC 性能对比图

Fig. 6 Performance comparison of MTXA matrix pair

3　针对 PHVM 的 GPU 核函数配置最优化理论模型

3.1　最优化模型的提出

在 CUDA 编程中，为了有效地利用 GPU 的多核心结构，需要将计算任务划分为多个线程块，每个线程块中包含多个线程。网格是线程块的集合，包含多个线程块。网格和线程块大小都可以通过核函数运行配置来调整，以满足不同的应用需求。核函数的执行效率严重依赖于 CUDA 网格大小及线程块大小的设置。核函数配置往往通过工程师的经验设定，NVIDIA 厂商只给出了设置核函数配置的经验准则，并未给出通用的核函数配置。

基于 CUDA 的基本优化准则[16,21]，本章节针对 PHVM 的计算流程以及具体 GPU 的硬件配置，建立了 PHVM 中核函数配置的最优化理论模型。该模型的输入包括待求解问题规模、GPU 的硬件参数与算法的 GPU 流多处理器使用率。优化目标为最小化 PHVM 运行时间。输出理论最优 CUDA 的网格大小以及线程块大小。

考虑用 PHVM 求解大规模水平矩阵乘竖直矩阵 $\boldsymbol{E}\times\boldsymbol{F}$（见图 1），其中 $\boldsymbol{E}\in\mathbb{R}^{M\times K}$，$\boldsymbol{F}\in\mathbb{R}^{K\times N}$，$M\ll K$，$N\ll K$。由于求解问题规模大，在配置合理的情况下 GPU 线程理应满载运行，因此在建模的过程中我们考虑以串行的方式计算每一个结果矩阵中的元素，优化目标为：

$$\min_{GS,BS} M\cdot N\cdot t_{\text{single\_elem}} \tag{1}$$

其中，$t_{\text{single\_elem}}$ 为结果矩阵中单个元素的计算消耗，共 $M\cdot N$ 个元素。在给出 $t_{\text{single\_elem}}$ 的具体表达式之前，我们先给出模型约束的分析如下。

为了同时兼顾 GPU 线程的并行度与 GPU 硬件的利用率，GPU 允许处于活动状态的线程数量有限。令 SMS 为单个流处理器中能够并行的线程数量上限，SM_{NUM} 为 GPU 中流处理器的数量，可以得到约束：

$$BS\cdot GS=SMS\cdot SM_{\text{NUM}} \tag{2}$$

CUDA 线程的执行方式为单指令多线程，最小执行单位为线程束 Warp，令 WS 为线程束中线程的数量，为了使每次指令执行都调用 WS 个线程，给出约束：

$$BS=k_1\cdot WS, \tag{3}$$

其中，k_1 为正整数。

在 CUDA 中，一个流多处理器 SM 可同时执行多个线程块，但是一个线程块不能同时在多个 SM 上运行，所以为了使流多处理器激活尽可能多的线程，给出约束：

$$k_2\cdot BS=SMS, \tag{4}$$

其中，k_2 为正整数。

综上所述，将优化目标式(1)与约束条件(2)－(4)整合，得到优化模型：

$$\min_{GS,BS} M\cdot N\cdot t_{\text{single\_elem}}$$
$$\text{s.t.}\begin{cases} BS\cdot GS=SMS\cdot SM_{NUM} \\ BS=k_1\cdot WS \\ k_2\cdot BS=SMS \end{cases} \tag{5}$$

其中，BS，GS，k_1，k_2 为正整数。

下面分析计算单个结果矩阵元素的耗时 $t_{\text{single\_elem}}$。单个元素需要一次内积操作，PHVM 中采用的内积算法如图 3 所示。首先聚焦于单个线程，对照图 3，在步骤 $\hat{L}4$ 中，单个线程需要访问全局内存 $\frac{2\cdot K}{BS\cdot GS}$ 次以加载向量数据，该步需要时间 $\frac{2\cdot K}{BS\cdot GS}\cdot\sigma_g$，其中 σ_g 为单次访问全局内存的时间。再对这些数据对进行“乘加”操作，并将结果暂存于该线程的寄存器中，该步需要时间 $\frac{K}{BS\cdot GS}(\sigma_{\text{add}}+\sigma_{\text{mul}})$，其中 σ_{add} 为单次浮点加操作时间，σ_{mul} 为单次浮点乘操作时间。步骤 $L5$ 首先将线程上步的计算结果暂存进其所在线程块的共享内存，然后并行加线程块内部共享内存的暂存结果。我们考虑计算量最多的线程，其计算时间为 $\sigma_s+2\cdot\log BS\cdot\sigma_s+\log BS\cdot(\sigma_{\text{add}}+\sigma_s)$。步骤 $L6$ 为原子加操作，由每个线程块的第一个线程将该线程块中所有线程的累积结果并入位于全局内存的 RES 中，其需要时间 $\sigma_s+\sigma_g+\sigma_{\text{add}}+\sigma_g$。

至此，结果矩阵单个元素的耗时为：

$$t_{\text{single\_elem}}=t_{\text{prep\_comp}}+t_{\text{post\_store}} \tag{6}$$

其中：

$$t_{\text{prep\_comp}}=\frac{K}{BS\cdot GS}\times(2\cdot\sigma_g+\sigma_{\text{add}}+\sigma_{\text{mul}})+\sigma_s+\log BS\cdot(\sigma_{\text{add}}+3\cdot\sigma_s) \tag{7}$$

$$t_{\text{post\_store}}=\sigma_s+\sigma_g+\sigma_{\text{add}}+\sigma_g \tag{8}$$

我们称步骤 $\{\hat{L}4,L5\}$ 为前置计算，步骤 $L6$ 为后置存储，$t_{\text{pre\_comp}}$ 和 $t_{\text{post\_store}}$ 分别为前置计算耗时与后置存储耗时。

GPU 线程并行与 GPU 线程并发是两种不同的概念。并发是指多个任务共享计算资源，并按照一定的时间顺序执行。GPU 的并发能力是指其能够同时激活的最大线程数 $SMS\cdot SM_{\text{NUM}}$。并行是指同时执行多个任务，每个任务运行在不同的处理器或计算核心上。GPU 的并行能力是指其能够同时执行的最大线程数。为了更进一步分析式(6)，我们给出如下假设。

(1)计算数据的规模足够大，满足线程数满载的必要条件。

(2)GPU 核并行处理的线程会将其一直占用，直到该线程完成前置计算为止。

(3)同时调用的线程也会同时完成前置计算。

基于以上假设，我们给出式(6)的优化：

(1)当 $t_{\text{pre\_comp}}<\frac{CoreS}{BS}\cdot t_{\text{post\_store}}$ 时：

$$t_{\text{single\_elem}}=t_{\text{pre\_comp}}+GS\cdot t_{\text{post\_store}} \tag{9}$$

(2)当 $t_{\text{pre\_comp}}\geqslant\frac{CoreS}{BS}\cdot t_{\text{post\_store}}$ 时：

$$t_{\text{single\_elem}}=\frac{GS\cdot BS}{CoreS}t_{\text{pre}_{\text{comp}}}+\frac{CoreS}{BS}GS\cdot t_{\text{post}_{\text{store}}} \tag{10}$$

系数 $\frac{GS\cdot BS}{CoreS}$ 为计算结果矩阵单个元素的线程块调用轮数：总共有 $CoreS$ 个处理核，故可并行执行 $\frac{CoreS}{BS}$ 个线程块的线程；总共有 GS 个线程块，故需要 $\frac{GS\cdot BS}{CoreS}$ 次执行完所有的线程块。由假设(2)可知，同一轮计算的所有线程块同时结束，所以这些线程块的后置存储步骤会串行执行。

式(9)为前置计算理论耗时小于后置存储的理论耗时，计算过程如图 7 所示，当前线程在每一轮前置计算完成后释放硬件资源，供下一轮线程进行前置计算。此时，后置计算是整个矩阵乘计算的瓶颈。

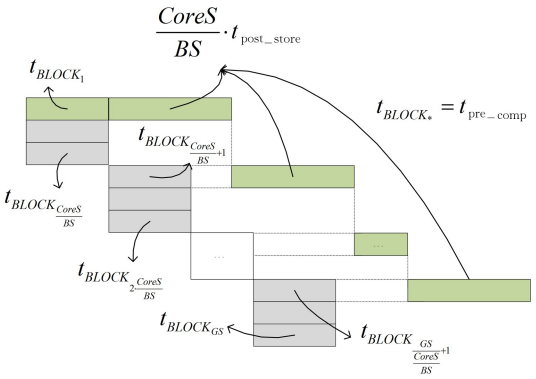

图 7 前计算时间小于单 Block 原子加时间示意图

Fig. 7 Diagram of previous calculation time is less than a single block atom plus time

式(10)为前置计算理论耗时大于或等于后置存储的理论耗时,计算过程如图 8 所示,每一轮的后置存储操作被该轮的前置计算操作重叠隐藏(Overlap)。此时,前置计算是整个矩阵乘计算的瓶颈。

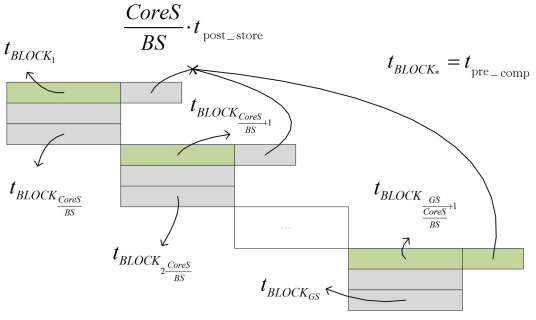

图 8 前计算时间大于单 Block 原子加时间示意图

Fig. 8 Diagram of previous calculation time is greater than a single block atom plus time

进一步,我们注意到在执行 CUDA 程序时,不同的算法对流多处理器的利用率也不同。SM 利用率是指 GPU 中的流多处理器(SM)处于处理数据状态的时间占总时间的百分比。在统计 t_{pre_comp} 时,我们假设了流多处理器满载,现将 SM 的利用率引入模型,以尽可能消除误差。

基于式(7)中 t_{pre_comp} 的表达式,有:

$$t_{pre\_comp}=\left(\frac{K}{BS\cdot GS}(2\cdot\sigma_g+\sigma_{add}+\sigma_{mul})+\sigma_s+\log BS\cdot(\sigma_{add}+3\cdot\sigma_s)\right)\cdot\frac{1}{u_{SM}} \tag{11}$$

其中,$\frac{1}{u_{SM}}$ 为流多处理器的利用率。

3.2 模型验证实验

本章节给出具体的数值实验以验证核函数配置模型的有效性。在该实验中,针对水平矩阵乘竖直矩阵 $\boldsymbol{E}\times\boldsymbol{F}$,其中 $\boldsymbol{E}\in\mathbb{R}^{F\times K}$,$\in\mathbb{R}^{K\times N}$,$M\ll K$,$N\ll K$,我们给出了 PHVM 的核函数实际运行时间(ms)、模型理论时间(时钟周期)在不同 GPU 上的趋势对比。

本实验所用的矩阵均为程序随机生成,矩阵中所有元素服从范围为[-1,1]的均匀分布。表 2 列出了实验所用矩阵组的具体信息。

表 2 随机矩阵对信息

Tabel 2 Stochastic matrix pair information

矩阵对	M	K	N	矩阵元素类型	矩阵元素分布
MTX1	5	3×10^7	5	FP32	U(-1,1)
MTX2	7	3×10^7	7	FP32	U(-1,1)
MTX3	9	3×10^7	9	FP32	U(-1,1)

实验所用 CPU 型号为 Intel(R) Xeon(R) Gold 5218 CPU @ 2.30 GHz,操作系统为 CentOS Linux release 7.8.2003(Core),g++编译器使用 g++(GCC) 9.3.0,CUDA 版本为 11.1,GPU 型号分别为 A100-PCIE-40GB,GeForce-RTX-2080-Ti。

通过 Wong 等[25]开源的 GPU 基准测试工具,我们测试出了两块 GPU 的操作延时,包括单精度浮点加法 σ_{add}、单精度浮点乘法 σ_{mul}、单次访问全局内存 σ_g 以及单次访问共享内存 σ_s。表 3 列出了这两个 GPU 的硬件参数信息,表 4 给出了这两个 GPU 的操作延时及运行 PHVM 时对应流多处理器的利用率,其中 σ_{add},σ_{mul},σ_g,σ_s 的统计单位为时钟周期。SMs 利用率 u_{SM} 是在 GPU 上运行 PHVM 得来:使用 NVIDIA 提供的工具 NVIDIA(R) Nsight Compute Command Line Profiler[1)] 以生成运行 PHVM 时的日志文件,通过 NVIDIA NSight Compute 可视化工具[2)]得到 u_{SM}。

表 3 GPUs 硬件参数

Tabel 3 GPUs hardware parameters

GPU 型号	SM_{NUM}	SMS	WS	$CoreS$
RTX-2080Ti	68	1024	32	4352
A100-40GB	108	2048	32	6912

表 4 GPUs 操作延时以及运行 PHVM 的 SMs 利用率

Tabel 4 GPUs operation latencies and SM utilization during PHVM execution

GPU 型号	σ_{add}	σ_{mul}	σ_g	σ_s	u_{SM}
RTX-2080Ti	1	2	346	49.39	0.27
A100-40GB	1	2	317	34.46	0.16

图 9 与图 10 分别给出了 PHVM 在 GPU GeForce-RTX-2080-Ti 与 A100-PCIE-40GB 上的实际运行时间(毫秒)以及理论模型时间(时钟周期)随核函数执行配置改变的变化趋势,图中'RUN''THEO'后缀分别为实际运行结果与本文理论模型得出结果。实验数据显示:针对表 2 中的矩阵对数据组,PHVM 在 GPU GeForce-RTX-2080-Ti 上的最佳核函数配置均为"GS(136),BS(512)",与理论模型保持一致;PHVM 在 GPU A100 上的最佳核函数配置均为"GS(864),BS(256)",与理论模型保持一致。同时,理论模型也较为准确地预测出了 PHVM 性能急剧下降的拐点:PHVM 在两块 GPU 上均表现出随着 GS 增大其运行时间先减少后增加的特点,根据式(9)与式(10)可知,这是因为随着 GS 增大,前置处理时间 t_{pre_comp} 减小,后置存储时间 t_{post_store} 增大,拐点即 t_{pre_comp} 减小量刚好等于 t_{post_store} 增加量的情况,超过了此阈值,t_{post_store} 会成倍增长。

实验数据表明,我们提出的核函数配置最优化理论模型较好地反映了 PHVM 在 GPU 上运行的实际状态,其推导的理论最优核函数配置与实际最优运行核函数配置相符。当研

1) https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html

2) https://docs.nvidia.com/nsight-compute/NsightCompute/index.html

究人员在已知参数的 GPU 上部署计算 PHVM 算法时，可以根据该理论模型自动计算出最优的核函数参数配置组合。

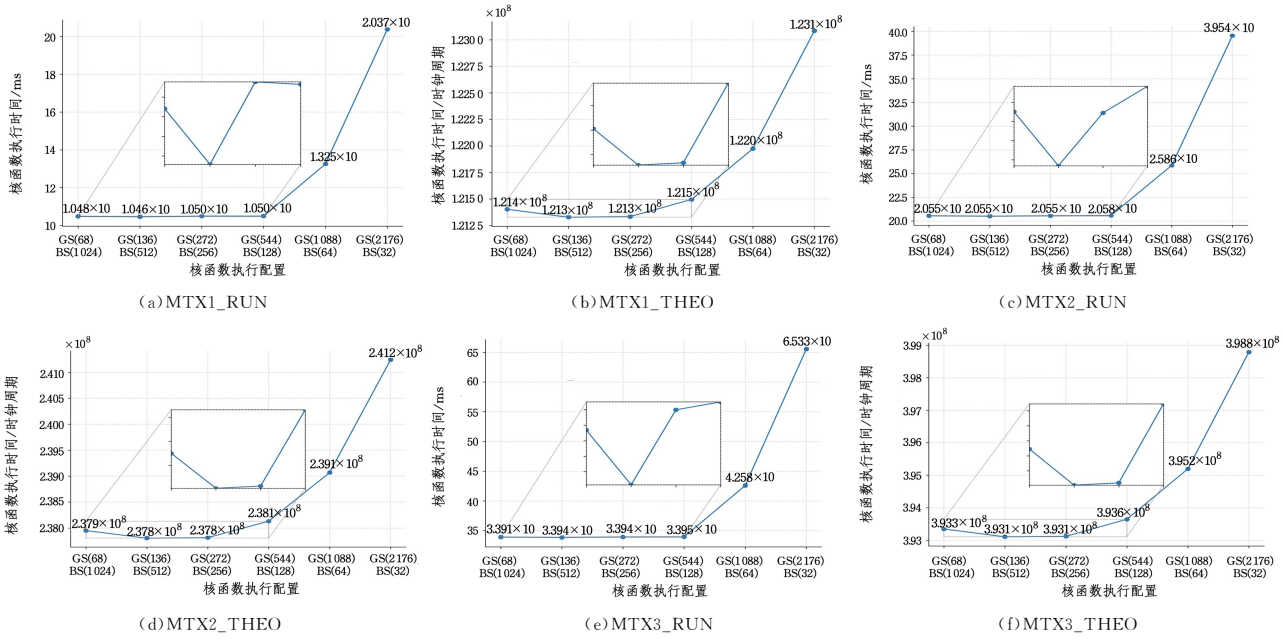

图 9　PHVM 实际运行时间与理论模型时间在 GPU 2080Ti 上的趋势对比

Fig. 9　Comparison of actual running time and theoretical model time of PHVM on GPU 2080Ti

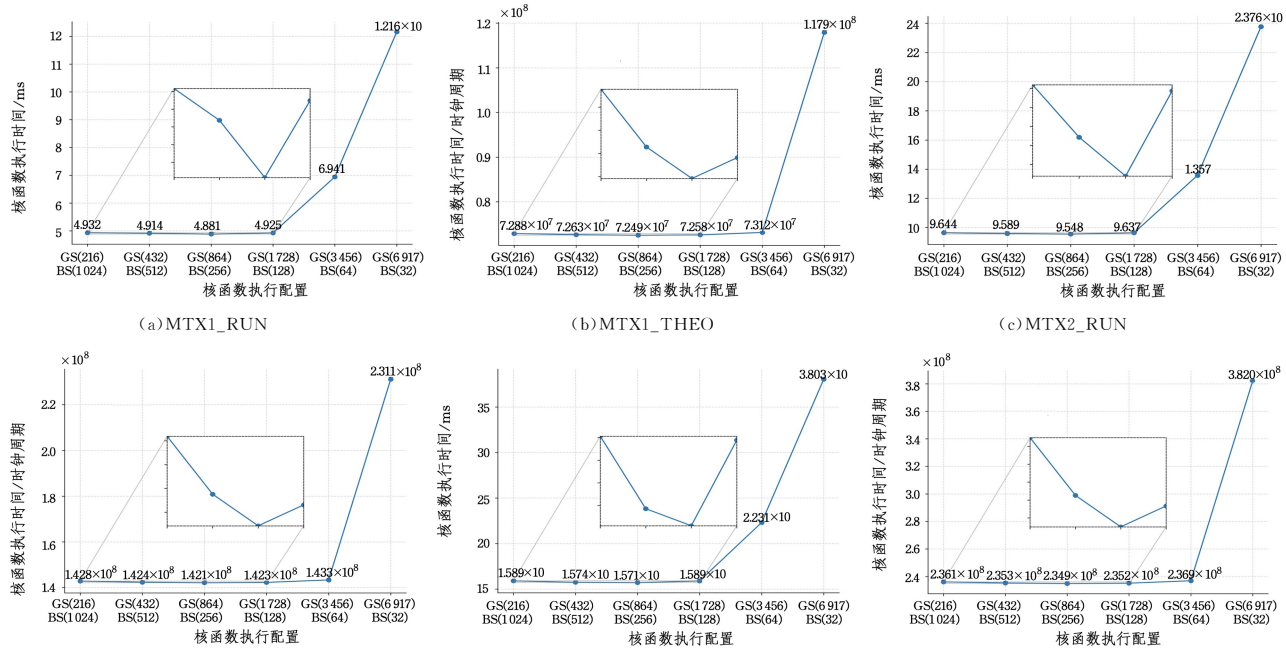

图 10　PHVM 实际运行时间与理论模型时间在 GPU A100 上的趋势对比

Fig. 10　Comparison of actual running time and theoretical model time of PHVM on GPU A100

需要注意的是，通过该模型得出的理论运行总时间要长于实验所得的运行总时间，这是因为模型尚未考虑 GPU 不可编程内存(缓存)的访存优化。在未来的工作中，可以考虑将更多的 GPU 硬件参数如 L1 和 L2 缓存等考虑进模型建立中，以更为准确地预估程序执行时间。

结束语　本文提出了一种基于 GPU 的并行矩阵乘算法 PHVM 以解决矩阵乘问题——水平矩阵乘竖直矩阵。数值实验表明，当左乘矩阵的水平维度远远大于竖直维度时，PHVM 要显著优于 NVIDIA cuBLAS 库中的通用矩阵乘法 cublasSgemm。

更进一步，GPU 核函数配置直接影响计算效率，但迄今针对最优核函数配置的研究极为有限。本文给出了 PHVM 的核函数配置最优化理论模型，其以 GPU 的硬件参数、流多处理器的利用率、左乘矩阵的竖直维度为输入参数，输出理论最优的核函数配置(线程块数量、单个线程块中线程数量)。从数值实验来看，该理论模型较准确地反映了 PHVM 的实际运行状态，描述了 PHVM 算法运行时间随核函数配置(网格大小、线程块大小)变换的变化趋势，通过该模型得到的理论最优核函数配置与实际最优运行核函数配置相符。当研究人员在已知参数的 GPU 上部署计算 PHVM 时，可以根据该理论模型自动计算出最优的核函数参数配置组合。

本文对 PHVM 建立的理论模型可以直接得出最优核函

数配置，但是由于目前该模型尚未考虑 GPU 不可编程内存的访存优化，所以通过该模型得出的理论运行总时间与实验所得的运行总时间仍存在一定偏差。在未来的工作中，可以考虑将更多的 GPU 硬件参数如 L1 和 L2 缓存等考虑进模型建立中，以更为准确地预估程序执行时间。

参 考 文 献

[1] AHMED N, NATARAJAN T, RAO K R. Discrete cosine transform [J]. IEEETransactions on Computers, 1974, 100(1): 90-93.

[2] GERSHO A, GRAY R M. Vector quantization and signal compression: volume 159 [M]. Springer Science Business Media, 2012.

[3] LECUN Y, BENGIO Y, HINTON G. Deep learning [J]. Nature, 2015, 521(7553): 436-444.

[4] STRASSEN V. Gaussian elimination is not optimal [J]. Numerische Mathematik, 1969, 13(4): 354-356.

[5] COPPERSMITH D, WINOGRAD S. Matrix multiplication via arithmetic progressions[C]// Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing. 1987: 1-6.

[6] DAVIS T A, RAJAMANICKAM S, SID-LAKHDAR W M. A survey of direct methods for sparse linear systems [J/OL]. Acta Numerica, 2016, 25: 383-566. https://www.cambridge.org/core/article/survey-of-direct-methods-for-sparse-linear-systems/8AE7AC55909389F7EA1F027855AC4044. DOI: 10.1017/S0962492916000076.

[7] CASASENT D, TAYLOR B K. Banded-matrix high-performance algorithm and architecture [J/OL]. Applied Optics, 1985, 24(10): 1476-1480. https://opg.optica.org/ao/abstract.cfm?URI=ao-24-10-1476. DOI: 10.1364/AO.24.001476.

[8] CHALLACOMBE M. A general parallel sparse-blocked matrix multiply for linear scaling scf theory [J/OL]. Computer Physics Communications, 2000, 128(1): 93-107. https://www.sciencedirect.com/science/article/pii/S0010465500000746. DOI: https://doi.org/10.1016/S0010-4655(00)00074-6.

[9] RUBENSSON E H, RUDBERG E. Locality-aware parallel block-sparse matrix-matrix multiplication using the chunks and tasks programming model [J/OL]. Parallel Computing, 2016, 57: 87-106. https://www.sciencedirect.com/science/article/pii/S0167819116300606. DOI: https://doi.org/10.1016/j.parco.2016.06.005

[10] CHOWDHERY A, NARANG S, DEVLIN J, et al. Palm: Scaling language modeling with pathways [J]. arXiv: 2204.02311, 2022.

[11] ASAIMEH M, DENOLF K, LO J, et al. Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels[C]// 2019 IEEE International Conference on Embedded Software and Systems(ICESS). IEEE: 1-8.

[12] LINDHOLM E, NICKOLLS J, OBERMAN S, et al. NVIDIA Tesla: A unified graphics and computing architecture [J/OL]. IEEE Micro, 2008, 28(2): 39-55. https://dx.doi.org/10.1109/mm.2008.31.

[13] NVIDIA, VINGELMANN P, FITZEK F H. CUDA, release: 10.2.89 [EB/OL]. 2020. https://developer.nvidia.com/cuda-toolkit.

[14] FATAHALIAN K, SUGERMAN J, HANRAHAN P. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication[C]// Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware. 133-137.

[15] HERAULT T, ROBERT Y, BOSILCA G, et al. Generic matrix multiplication for multi-gpu accelerated distributed-memory platforms over parsec[C]// 2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA). IEEE: 33-41.

[16] CLBLAS [EB/OL]. https://github.com/clMathLibraries/clBLAS.

[17] WANG Z W, CHENG L L, ZHAO W Q. Parallel Computation Performance Analysis Model Based on GPU [J]. Computer Science, 2014, 41(1): 31-38.

[18] YIN M J, XU X B, XIONG Z G, et al. Quantitative Performance Analysis Model of Matrix Multiplication Based on GPU [J]. Computer Science, 2015, 42(12): 13-17.

[19] DALTON S, OLSON L, BELL N. Optimizing sparse matrix-matrix multiplication for the ssGPU [J]. ACM Transactions on Mathematical Software(TOMS), 2015, 41(4): 1-20.

[20] CORMEN T H, LEISERSON C E, RIVEST R L, et al. Introduction to algorithms, third edition (3rd ed) [M]. The MIT Press, 2009.

[21] GUIDE D. CUDA C best practices guide [J]. NVIDIA, July, 2013.

[22] STIVAL P, GIRAUD L. Performance and accuracy of the matrix multiplication routines: cuBLAS on NVIDIA tesla versus MKL and ATLAS on Intel nehalem [Z]. 2010.

[23] KHALILOV M, TIMOVEEV A. Performance analysis of CUDA, OpenACC and OpenMP programming models on Tesla v100 GPU[J]. Journal of Physics: Conference Series, 2021, 1740(1): 012056.

[24] BARRACHINA S, CASTILLO M, IGUAL F D, et al. Evaluation and tuning of the level 3 cuBLAS for graphics processors [C]// 2008 IEEE International Symposium on Parallel and Distributed Processing. IEEE: 1-8.

[25] WONG H, PAPADOPOULOU M M, SADOOGHI-ALVANDI M, et al. Demystifying GPU microarchitecture through microbenchmarking [C]// IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS 2010). 2010: 235-246. DOI: 10.1109/ISPASS.2010.5452013.

LEI Chao, born in 1997, postgraduate. His main research interests include parallet algorithm and iterative linear system solver.

LIU Jiang, born in 1979, Ph.D, associate professor, is a member of CCF (No. 50519M). His main research interests include computability theory, formal methods and computer algorithms.