文章编号：2095-2163(2022)11-0192-05 中图分类号：TP391 文献标志码：A

基于 CUDA 并行优化的矩阵相乘算法研究

赵志建
(江苏联合职业技术学院 南京工程分院，南京 211135)

摘 要： 矩阵相乘作为线性代数的基础运算，不仅在数学领域被大量使用，在应用数学、物理学、计算机科学、人工智能等领域也得到了广泛应用。基于 CUDA 并行优化的矩阵相乘算法的提出，有效解决了传统 CPU 低吞吐量，高延时的问题；同时，为了充分利用共享内存资源，还提出了合并内存优化、内存冲突优化以及循环延展(Loop Unrolling)等优化算法来深度挖掘并行算法性能；通过在不同硬件平台上针对不同优化算法做了充分的对比实验分析，实验结果表明基于 CUDA 并行优化的矩阵相乘算法具有更好的性能。

关键词： 矩阵相乘；共享内存；CUDA 并行优化

Research on matrix multiplication algorithm based on CUDA parallel optimization

ZHAO Zhijian
(Nanjing Engineering Vocational College, Jiangsu Union Technical Institute, Nanjing 211135, China)

【Abstract】 As the basic operation of linear algebra, matrix multiplication is not only widely used in the field of mathematics, but also promoted in the fields of applied mathematics, physics, computer science, artificial intelligence, etc. The proposed matrix multiplication algorithm based on CUDA parallel optimization effectively solves the problems of low throughput and high latency of traditional CPUs. At the same time, in order to make full use of shared memory resources, optimization algorithms such as coalesced memory, memory conflict and loop unrolling are also proposed to deeply mine the performance of parallel algorithms. In this paper, sufficient experimental comparison and analysis on different hardware platforms are conducted. The experimental results show that the matrix multiplication algorithm based on CUDA parallel optimization has better performance.

【Key words】 matrix multiplication; shared memory; CUDA parallel optimization

0 引 言

矩阵相乘作为数值分析统计学和机器学习中最为常见的数学运算，在 FEA 平衡方程、线性回归、决策树、朴素贝叶斯等等的求解上都可以分解成系列矩阵相乘或者矩阵乘向量的运算。在深度学习领域，常用的卷积(Convolution)、全连接、批归一化(Batch-Normalization)、下采样(MaxPooling)等计算机视觉中常用算子操作也都离不开矩阵相乘运算。常用的编解码器(Encoder -Decoder)、注意力机制(Multi-Head Attention)等自然语言处理中的基本算子依旧离不开矩阵相乘运算。

随着矩阵维度的激增，传统单 CPU 矩阵相乘算法的高复杂度带来了巨大的性能瓶颈，为了缓解单线程大矩阵相乘运算的耗时问题，一种基于共享内存的多线程并发机制应运而生。通过将大矩阵相乘任务划分给多个子线程，提高计算性能；另一种是将大矩阵划分成多个子模块单独相乘后再相加，以减少内存访问次数，提高性能。但是就目前而言，深度学习的应用正日趋普及，大矩阵相乘的运算量突增，对于实时性要求很高的人脸识别、无人驾驶、医疗影像分割等应用来说，传统 CPU 平台实现的矩阵运算已无法满足需求，亟需一种更加高效的并行计算模式打破该性能瓶颈。

英伟达工智能计算公司首次定义了 GPGPU 概念，并提出了 CUDA (Compute Unified Device Architecture)并行计算架构，同时支持硬件和软件。CUDA 可利用图形处理器中的多颗计算核心进行通用计算处理，计算性能显著提升，包含 CUDA 指令集架构(ISA)以及 GPGPU 内部的并行计算引擎，还方便开发人员直接使用 C 语言来为 CUDA 架构编写程序，并在支持 CUDA 的 GPGPU 流处理器(Stream Multiprocessor, SM)上以超高性能实现运行。CUDA 并行计算架构的问世，使得矩阵运算性能得到质的飞跃。本文通过使用 CUDA 来做矩阵相乘运算，并充分利用 SM 资源对其性能进行优化，

作者简介：赵志建(1976-)，男，副教授，主要研究方向：计算机应用。

收稿日期：2022-04-13

且在不同 GPGPU 硬件平台上针对不同优化算法做了充分的实验对比及性能分析。

1　相关工作

1.1　矩阵乘积定义

矩阵相乘是一种将 2 个矩阵乘积运算，得到第 3 个矩阵的二元运算。设 $\boldsymbol{A}$ 为 $M \times K$ 的矩阵，$\boldsymbol{B}$ 为 $K \times N$ 的矩阵，那么称 $M \times N$ 的矩阵 $\boldsymbol{C}$ 为矩阵 $\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的乘积，记作 $\boldsymbol{C} = \boldsymbol{AB}$，其中矩阵 $\boldsymbol{C}$ 的第 i 行第 j 列如公式(1) 所示：

$$(\boldsymbol{AB})_{ij} = \sum_{k=1}^{K} a_{ik}\, b_{kj} = a_{i1}\, b_{1j} + a_{i2}\, b_{2j} + \cdots + a_{iK}\, b_{Kj} \tag{1}$$

1.2　并行计算架构简介

CPU 实现的并行计算大多依据多处理器共享内存机制进行多线程并行编程，典型的框架包括 MPI (Message Passing Interface)，OpenMP (Open Multi - Processing)，TBB (Intel Threading Building Blocks)，OpenCL(Open Computing Language)等。

目前，主流的 GPGPU 实现的并行计算架构有 CUDA 架构、ROCM、OpenCL 等。NVIDIA 提出的 GPGPU 作为现如今最为流行的并行框架，其整体结构主要由大量的 SM 和 DRAM 存储等构成，每个 SM 又由大量计算核(又称 CUDA 核)、LDU(Load-Store Units)、SFU(Special-Function Units)、寄存器、共享内存等构成。GPGPU 具有高并行度计算能力的基础，每个 SM 支持数百线程并发执行，每个 GPGPU 通常有多个 SM，所以一个 GPGPU 可以并发执行数千线程。CUDA 采用和 CPU 编程中常见的单指令多数据(SIMD)架构类似的单指令多线程(SIMT)架构来管理和执行线程，每 32 个线程为一组，被称为线程束。一个线程块只能在一个 SM 上被调度，而且一旦线程块在一个 SM 上被调度，就会保存在该 SM 上直到执行完成。需要注意的是，这 2 种层级并不是完全一一对应的，比如在同一时间，一个 SM 可以容纳多个线程块。

在 SM 中，共享内存和寄存器是非常重要的资源。共享内存被分配在 SM 上的常驻线程块中，寄存器在线程中被分配。线程块中的线程通过这些资源可以进行相互的合作和通信。尽管线程块里的所有线程都可以逻辑地并行运行，但并不是所有线程都可以同时在物理层面执行。因此线程块中的不同线程可能会有不同的运行速度，且在需要时可以使用 CUDA 语句进行线程的同步。

1.3　内存优化算法

在大多数 GPGPU 应用程序中，性能优化的关键点在于如何高效访问内存，尤其共享内存的合理分配使用。

典型的 GPGPU 内存优化算法包括共享内存优化、内存合并优化、内存冲突优化等。共享内存相较于全局内存而言，延迟要低上大约 20~30 倍，而带宽要高出约 10 倍，因此合理分配共享内存是性能优化的关键。矩阵分块思想与 CPU 矩阵分块思想相同。对齐访问含义就是如果“内存事务”(32 和 128 字节两种)的访问首地址是缓存粒度(L1 的 128 字节或 L2 缓存的 32 字节)的偶数倍，即实现了对齐访问。在 L1 缓存的情况下，由“128 字节内存事务”进行访问，如果一个线程束访问的地址是连续的 128 字节，且首地址又是 128 的倍数，那么这种访问就称为合并访问，内存合并对齐访问对性能提升起着关键作用。往往为了获得较高的内存带宽，共享内存被划分成了多个大小相等的存储器模块，称为 bank。内存 bank 冲突表示当一个线程束中的不同线程访问一个 bank 中的不同的字地址时，就会发生 bank 冲突。如若没有 bank 冲突，共享内存的访存速度将会非常快，而如果在使用共享内存时发生了 bank 冲突的话，性能将会降低很多，所以避免内存 bank 冲突尤为重要。不同于内存优化，循环延展是一种以编程复杂为代价来提升并行代码性能的高级的编程方式，是一种指令集优化，其性能较内存优化提升更为明显。

2　CPU 并行优化算法

CPU 实现的矩阵相乘伪代码，具体见算法 1。通过 3 个 for 循环即可完成公式(1)中表达的矩阵相乘运算。

算法 1　矩阵相乘串行实现

```
/* 定义 3 个矩阵 A, B, C */
for i from 0 to M do
    for j from 0 to N do
        for k from 0 to K do
            C(i, j) += A(i,k) * B(k, j)
```

2.1　OpenMP 并行优化

OpenMP 是基于共享内存模型的多线程并行模式，适合于应用在单机多核心平台上。程序开始时只有一个主线程，程序中的串行部分都由主线程执行，并行的部分是通过派生其他线程来执行。目前主流编译器默认都已支持 OpenMP，只需要在第一

个 for 循环之前加上“#pragma omp” 语句,表示动态分配线程数,且保证每个 CPU 线程单独并行地完成矩阵点乘任务。算法实现伪代码具体见算法 2。

算法 2 矩阵相乘并行实现

```
/*定义 3 个矩阵 A, B, C */
#pragma omp //开启 OpenMP 并行优化
for i from 0 to M do
  for j from 0 for N do
    for k from 0 to K do
    C(i, j) += A(i, k) * B(k, j)
```

2.2 访存优化

无论对于串行、还是 OpenMP 并行实现都未经过任何优化,访存延迟和通信开销会随着维度 M, N, K 的增加而增大。例如:对于 $M=N=K$ 的大型方阵,矩阵乘积运算次数为 N^3、即时间复杂度为 $O(N^3)$,所需的数据量为 $O(N^2)$,从而产生 N 阶的计算强度。而该算法又依赖于访存中保存的一个大工作集,这就使得随着 M、N 和 K 增长时,CPU 需要来回传送数据,显然不符合减小访存的思想。

C/C++中,默认会按行优先储存数据(一维数组),*ijk* 枚举顺序将会使得内层 k 循环中 $B[k, j] = B[k * K + j]$ 在内存中的枚举出现不连续、即按列读取,显然降低效率。若此时采用 *ikj* 的枚举顺序将提高访存效率,伪代码具体见算法 3。

算法 3 矩阵相乘访存优化实现

```
/*定义 3 个矩阵 A, B, C*/
for i from 0 to M do
  for k from 0 to K do
    S = A(i, k)
    for j from 0 to M do
    C(i, j) += S *B(k, j)
```

算法 3 中,在 k 循环中先读取 $\boldsymbol{A}[i, k]$ 保存到寄存器变量 $\boldsymbol{S}$ 中,在内层 j 循环计算时直接读取 $\boldsymbol{S}$,而 $\boldsymbol{B}[k, j]$ 和 $\boldsymbol{C}[i, j]$ 在 j 循环中是连续访问的。需要指出的是,在外层 k 循环中,*omp* 并行后去掉了最外层 i 循环,$\boldsymbol{A}[i, k]$ 也是连续读取的,这样就极大提高了访存效率。

2.3 矩阵分块优化

将矩阵乘法的计算转化为其各自分块矩阵相乘后相加,能够有效减少乘数矩阵和被乘数矩阵调入内存的次数,可加快程序运行速度。矩阵分块优化思想如图 1 所示,通过将原始矩阵进行分块,并将每个分块看作另一个矩阵的元素参与矩阵乘运算,接着将相乘结果进行累加,从而完成一个矩阵分块的矩阵乘,其他块的处理流程也和这个一样。

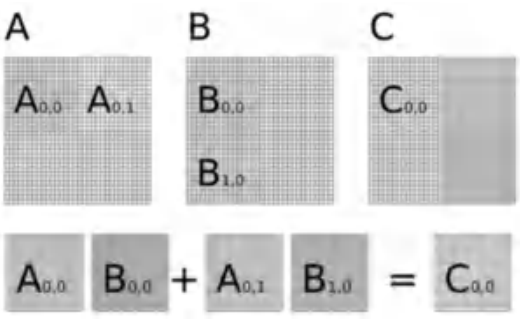

图 1 矩阵分块优化思想

Fig. 1 Matrix block optimization

3 CUDA 并行优化算法

CUDA 允许用户定义被称为内核(kernel)的 C 语言函数,在调用此类函数时,将由 N 个不同的 CUDA 线程并行执行 N 次,这与普通 C 语言函数只执行一次的方式有所不同。在定义内核时,需要使用 global 声明限定符并使用一种全新的<<<…>>>语法启动内核,同时还要指定每次调用的 CUDA 线程数。通过让每个线程对应矩阵 $\boldsymbol{C}$ 中一个元素来进行计算 ,每个线程从矩阵 $\boldsymbol{A}$ 中读取一行向量,从矩阵 $\boldsymbol{B}$ 中读取一列向量,对这 2 个向量做相乘累加运算,再将结果写回矩阵 $\boldsymbol{C}$。

$\boldsymbol{A}$、$\boldsymbol{B}$、$\boldsymbol{C}$ 三个矩阵都保存在 GPGPU 的全局内存中,每个线程都进行了大量重复的全局内存访问操作,虽然线程束机制优化了全局内存的访问效率,最大程度实现了合并访问,但 GPGPU 全局内存的读取速度仍然不高、即带宽有限。

3.1 共享内存优化

CUDA 中每个线程都有自己的私有的全局内存和寄存器,用来保存在核函数内不加修饰的声明的局部变量;线程块有自己的共享内存(Shared Memory),并对块内所有的线程可见。相较于全局内存 400~600 个时钟周期的访问延迟,共享内存只有 20~30 时钟周期访问延迟,且带宽也比全局内存高 10 倍,极大程度上提高了访存效率。

共享内存优化算法的核心思想借鉴了矩阵分块优化思想,通过充分利用数据的局部性,让一个线程块内的子线程先从全局内存中读取分块矩阵数据,并写入到共享内存中,在计算时,直接从共享内存中读取分块数据进行矩阵乘和累加操作,从而大大降低了访问延迟。接下来,让子矩阵块分别在矩阵 $\boldsymbol{A}$ 的行向以及矩阵 $\boldsymbol{B}$ 的列向上滑动,直到计算过所有

N 个元素的相乘累加为止。

3.2　合并内存优化

内存优化的目标在于通过更少的内存事务获得更多的内存请求，因此需要尽量进行对齐合并访问。内存合并访问是指所有线程访问连续且对齐的内存块，内存块大小支持 32 字节、64 字节以及 128 字节，分别表示线程束中每个线程以一个字节（1×32=32）、16 位（2×32=64）、32 位（4×32=128）为单位读取数据。

以合并内存访问 128 字节为例，每个线程读取一个浮点变量，那么一个线程束（32 个线程）将会执行 32×4=128 字节的合并访存指令，通过一次访存操作完成所有线程的读取请求，其缓存有效利用率达到 128/128=100%，如图 2 所示。

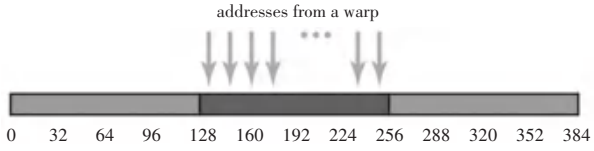

图 2　合并内存访问

Fig. 2　Coalesced memory access

非合并内存访问的对比如图 3 所示，128 字节的数据没有进行内存对齐，首地址位于 96～128 字节之间，为了访问 128 字节之前的数据，必须访问从 0～127 字节的整段内存，其缓存的有效利用率仅有一半，128/256=50%。

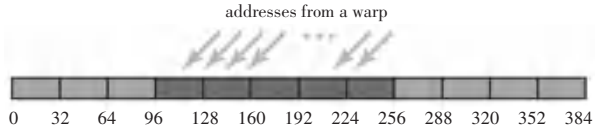

图 3　非合并内存访问

Fig. 3　Non-coalesced memory access

3.3　内存冲突优化

往往为了获得较高的内存带宽，共享内存被划分成了多个大小相等的存储器模块，称为 bank。一个 bank 内对多个地址进行读取和写入的操作可以同时进行，大大提高了整体带宽。当每个线程访问一个 32 位大小的数据类型的数据（如 int，float）时，就不会发生 bank 冲突，例如图 4 呈现了一种非内存 bank 冲突的场景。

但是很多情况下，无法充分发挥 bank 的作用，以致于共享内存的带宽受阻，这可能是因为遇到了 bank 冲突。例如，当同一个线程束中不同线程去访问共享内存中同一个 bank 的不同字地址时，就会发生 bank 冲突，例如图 5 中同一个线程束中多个线程访问了 Bank 0 的数据。

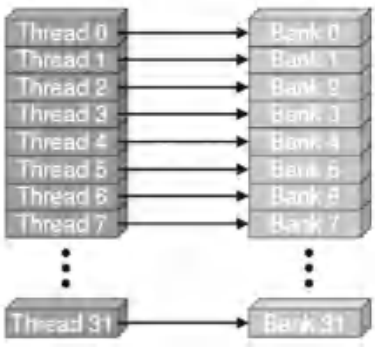

图 4　非内存 bank 冲突

Fig. 4　Non-memory bank conflict

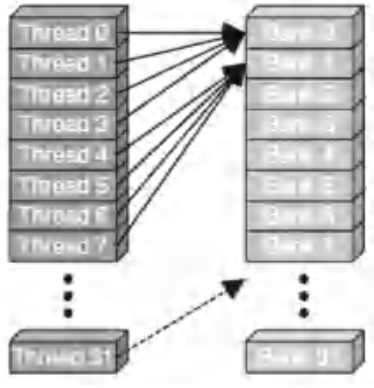

图 5　内存 bank 冲突

Fig. 5　Memory bank conflict

避免内存 bank 冲突常用的优化思路有 2 个：

（1）典型的线程访问方式：每个线程束的线程和每个 bank 的 ID 一一对应或者每个线程对应唯一的 bank。

（2）多播机制：当一个线程束中的多个线程同时访问一个 bank 的相同字地址时，会将该字广播给这些线程，从而也不会产生 bank 冲突。

3.4　循环延展优化

循环延展（Loop Unrolling）不同于内存优化，是一种指令级优化。前面提到的所有优化算法实现都离不开 for 循环的运用，而实际上 for 循环是一种以牺牲计算性能为代价的编程思路。

循环延展优化思想的提出，主要是为了降低循环开销，为具有多个功能单元的处理器提供指令集并行，同时也有利于指令流水线的调度。目前基于 CUDA 编程的编译器默认都已支持循环延展化，其实现方式和 OpenMP 并行优化算法类似，只需在 for 循环前添加“#pragma unroll”语句，编译器将会识别该语句，自动对其进行展开，而后并发去执行。具体伪代码见算法 4。

算法 4　矩阵相乘循环延展优化

/＊定义 3 个矩阵 ***A***，***B***，***C***＊/

```
#pragma unroll //循环展开指令
for i from 0 to M Step 1
    C(i) += A(i) * B(i)
/* 编译器将自动循环展开结果 */
    for i from 0 to M do Step 3
        C(i) += A(i) * B(i)
        C(i + 1) += A(i + 1) * B(i + 1)
        C(i + 2) += A(i + 2) * B(i + 2)
```

4 性能分析

在不同设备上进行 $M = N = K = 2\,048$ 阶矩阵相乘及优化算法的性能对比,矩阵相乘 CPU 并行算法的性能对比见表 1。明显看出基于 OpenMP 并行优化实现算法稍优于串行算法;矩阵相乘分块优化算法明显优于未分块算法;最优的仍是访存优化算法带来的性能提升。因此,内存优化一直是性能瓶颈的难点、也是挑战。

表 1 矩阵相乘 CPU 并行优化算法性能对比

Tab. 1 Performance comparison of matrix multiplication CPU parallel optimization algorithm ms

算法	X86_64 88 核
算法 1 矩阵相乘串行实现	96 234.63
算法 2 矩阵相乘 OpenMP 并行优化实现	91 933.13
算法 3 矩阵相乘访存优化实现	18 494.86
矩阵相乘分块优化实现	42 503.48

从表 1 明显看出,虽然基于 CPU 的并行优化算法较串行算法有了很大提升,但运行时间仍然较长,最优矩阵相乘访存优化算法也需 18.4 s,这个时间明显无法满足实时性应用需求。

基于 CUDA 并行优化矩阵相乘算法的运行时间,见表 2。

表 2 矩阵相乘 CUDA 并行优化算法性能对比

Tab. 2 Performance comparison of matrix multiplication CUDA parallel optimization algorithm

算法	1080Ti (ms)	Pascal (ms)	V100 (ms)
未经任何优化的矩阵相乘算法	40.36	43.04	34.10
基于共享内存优化的矩阵相乘算法	35.10	38.40	27.50
基于内存合并优化的矩阵相乘算法	30.13	34.23	16.23
内存冲突优化的矩阵相乘算法	28.02	30.64	13.46
基于循环延展优化的矩阵相乘算法	10.04	8.52	5.88

由表 2 明显看出,未经任何优化的 CUDA 并行算法比 CPU 实现的最快访存优化算法提升了 400 倍之余。尽管原生 CUDA 矩阵相乘实现算法得到了性能上的飞跃,但原生实现并没有真正充分利用 GPGPU 硬件资源,利用率往往达不到 100%。通过使用共享内存优化的矩阵分块优化算法,性能得到了明显提高,这是因为共享内存访问带宽明显高于全局内存。其次,在 CUDA 内存使用中,经常会遇到内存并非对齐、内存 bank 冲突等现象,通过使用 NVIDIA 提供的 nvprof 和 nvvp 性能分析工具可以发现内存使用中存在的问题。通过内存合并对齐优化之后的性能较未优化实现有很大提升,解决了内存 bank 冲突后也得到了部分性能提升。除此之外,还发现通过指令集优化的循环延展方法性能最为出色,这点得益于编译器优化。

5 结束语

随着 GPGPU 的普及,陆续推出了 CUDA、ROCM、OpenCL 等并行计算架构,不仅解决了 CPU 低算力带来的高延时,同时还为高实时性要求的人工智能应用提供了强有力的支撑。本文借用 CPU 平台和当今主流的 CUDA 并行计算架构实现了数学领域常用的矩阵相乘并行计算,并对其进行了有效的性能优化,提高了利用率,从而能够充分利用 GPGPU 硬件资源。实验结果表明,合理使用共享内存优化、指令集编译器优化能带来明显的性能提升。

参考文献

[1] LARSON M G, BENGZON F. The finite element method: Theory, implementation, and applications[M]//Texts in Computational Science and Engineering. Berlin/ Heidelberg: Springer, 2012, 10: 289-325.

[2] GOODFELLOW I, BENGIO Y, COURVILLE A. Deep learning[M]. USA: The MIT Press, 2016.

[3] CHO K, BAHDANAU D. Learning phrase representations using RNN encoder-decoder for statistical machine translation[C]// Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing(EMNLP). Doha, Qatar: ACL, 2014: 1724-1734.

[4] LI Jian, TU Zhaopeng, YAO Baosong, et al. Multi-head attention with disagreement regularization[C]//Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP). Brussels, Belgium: ACL, 2018: 2897-2903.

[5] HAN T D, ABDELRAHMAN T S. hiCUDA: High-level GPGPU programming[J]. IEEE Transactions on Parallel and Distributed Systems, 2011, 22(1): 78-90.

[6] Advanced Micro Devices, Inc. ROCm: Platform for development discovery and education around GPU computing[EB/OL]. [2022]. https://www.amd.com/zh-hans/no.

[7] STONE J E, GOHARA D, SHI G. OpenCL: A parallel programming standard for heterogeneous computing systems[J]. Comput. Sci. Eng., 2010, 12(3): 66-72.