Stream-K：面向 GPU 上稠密矩阵乘法的工作中心并行分解

Muhammad Osama
mosama@ucdavis.edu 加利福尼亚大学戴维斯分校 加利福尼亚州戴维斯，美国

Duane Merrill
dumerrill@nvidia.com 英伟达公司 加利福尼亚州圣克拉拉，美国

Cris Cecka
ccecka@nvidia.com 英伟达公司 加利福尼亚州圣克拉拉，美国

Michael Garland
mgarland@nvidia.com 英伟达公司，加利福尼亚州圣克拉拉，美国

John D. Owens
jowens@ucdavis.edu 加利福尼亚大学戴维斯分校，加利福尼亚州戴维斯，美国

arXiv:2301.03598v1 [cs.DS] 9 Jan 2023

摘要

我们介绍*Stream-K*，一种以工作为中心的矩阵乘法（通用矩阵乘法）及密集线性代数相关计算的并行化方法。与当代主要基于数据块的分解方法不同，我们的方法将内层循环迭代总量平均分配给各个物理处理单元。无论特定问题的输出分块能否在底层处理单元之间高效划分，该方法都能近乎完美地利用计算资源。

在 GPU处理器上，我们的 *Stream-K* GEMM 并行化在峰值情况下可实现高达 14× 和 6.7× 的加速比；与 CUTLASS 和 cuBLAS 等最先进的数学库相比，在 32,824 种 GEMM问题几何形状上的平均性能表现更高且更加稳定。此外，对于每种浮点精度，我们只需采用一种分块大小配置即可实现这一性能，而如今的数学库则使用复杂的内核选择启发式方法，从大量内核变体中进行选择。

关键词：矩阵乘法、GPU、负载均衡

1 引言

通用矩阵乘法（GEMM）、卷积及其他类似计算，构成了许多深度学习和科学计算应用中的主要工作负载。例如，GPU 等高性能处理器在计算 GEMM 时，旨在实现接近理论峰值的 100% 数学运算吞吐量。然而，要做到这一点，必须对工作进行分解，使底层物理核心得到充分利用。正如我们将要展示的，要在各种不同问题形状和规模下实现如此高的处理器利用率，仍然颇具挑战。

传统的通用矩阵乘法实现会通过对输出矩阵进行数据并行分块来划分计算，并将各个输出块的独立生成任务分配给并发线程（或线程组）[1, 8, 14]。每个输出分块的工作量是规则的，分块生成任务通常以“波次”的形式调度到空闲的物理核心上。当波次数量较多时，整体工作负载的均衡性更好，处理器利用率也最高；也就是说，输出块的数量远远超过核心数量。

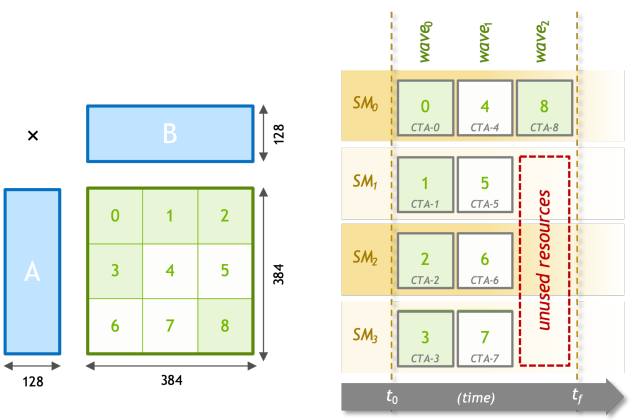
然而，随着处理器规模不断增大，这种超额订阅已大幅减少。核心数量增加后，生成给定数量的数据块所需的波次会减少。核心越大，越会促使矩阵采用更大的分块因子，从而减少波次，但每个数据块会更大。一般而言，波次较少的执行调度更容易受到量化效率低下的影响，即当输出块数量不是处理器核心数量的整数倍时，处理器利用率不足。当最后一波未满时，未使用的核心必须等待剩余线程执行数百万（甚至数十亿）条乘加（MAC）指令，才能执行任何依赖于这些指令的工作。

图 1a 展示了一个假设场景：某图形处理器拥有四个流式多处理器核心（SM）。若将一次384×384×128 通用矩阵乘法计算分块为九个 128×128输出分块，数据并行分解的吞吐量最高只能达到处理器额定吞吐量的 75%。如图1b所示，将分块大小减半可将这一理论利用率上限提升至 90%。然而，更细粒度的分块因子会降低缓存和暂存器的利用效率，也可能无法带来任何实际的性能提升。

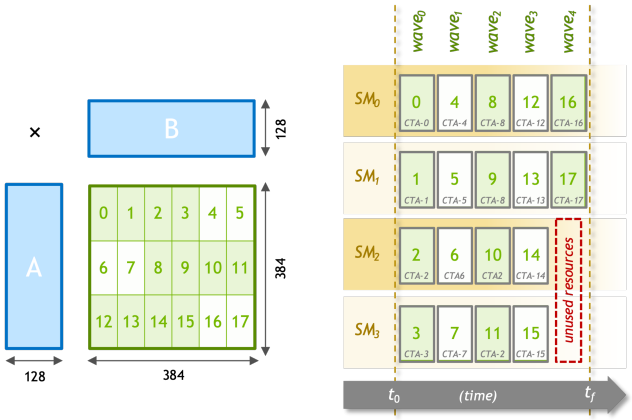
对于图形处理器等日益宽大的处理器而言，量化效率低下是一个值得关注的问题，因为其每核心算术逻辑单元数和每处理器核心数目前都达到数百。因此，许多常见的类通用矩阵乘法工作负载如今都会出现一个最后的部分填满波次，其耗时占总计算时间的相当大一部分。

目前，基于图形处理器的数学库和深度学习库采用的解决方案，是部署由多种分块配置组成的集成。当理想的分块因子无法很好地整除时，库会从工作量较小的分块方案中进行选择，例如图1b和图2a所示的方案。

“A”类发行声明（已批准公开发布，不限发行范围）。

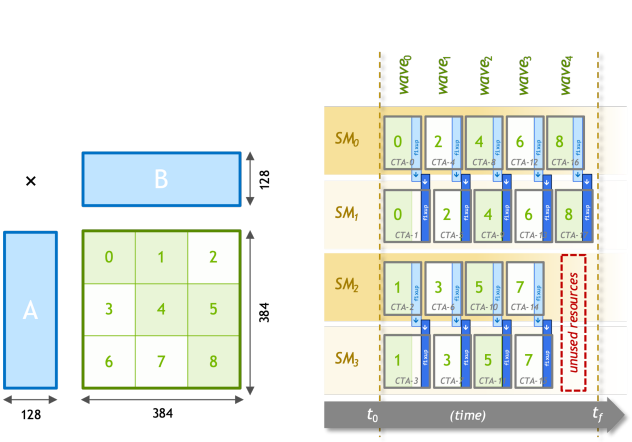


(a) 数据并行分解，网格大小g=9 个协作线程阵列，
较大的 $128 \times 128 \times 128$ 协作线程阵列工作量，
处理器利用率上限为 75%

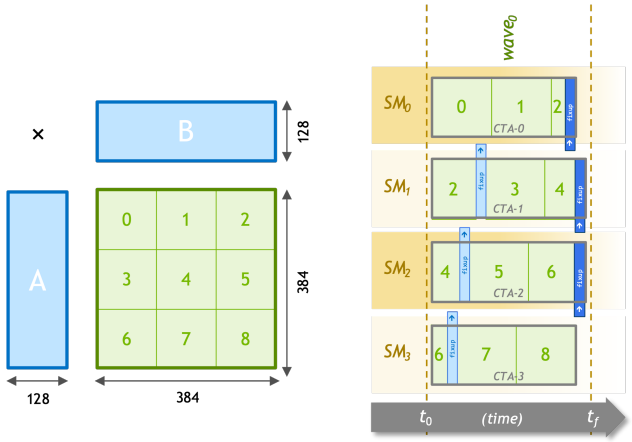


(b) 数据并行分解，网格大小 g=18 个协作线程阵列，
较小的 $128 \times 64 \times 128$ 协作线程阵列工作量，
处理器利用率上限为 90%

图 1. 数据并行 执行调度在假设的四个 SM 的 GPU 上执行 $384 \times 384 \times 128$ 通用矩阵乘法。



(a) 固定分割 分解，分割因子为s=2，
网格大小为 g=18 个协作线程阵列，协作线程阵列的工作量较小，
量化效率为 90%



(b) 基本*Stream-K* 分解，网格大小为 g=4 个协作线程阵列，
协作线程阵列的工作量较大，
且量化效率接近 100%

图 2. 在假想的四个 SM 的 GPU 上， $384 \times 384 \times 128$ 通用矩阵乘法的分块拆分执行调度。

然而，基于数据块的集成给数学库带来了性能和工程方面的挑战，尤其是那些致力于在各种问题规模和形状下实现尽可能高性能的库。大型集成的可分发代码体积可能成为问题。例如，NVIDIA 的 cuBLAS 库 [15]体积达数百兆字节；对于给定的 API 入口点，它通常会针对每种架构提供二十多个预编译的内核特化。大型集成还需要复杂的选择启发式方法。在我们的评估中，我们展示了这些启发式方法难以始终为任意问题找到最优配置。

与这些基于数据块的方法不同，我们的 *Stream-K* 分解始终将通用矩阵乘法计算所需的乘加循环迭代总量平均分配给各个流式多处理器。由于单次乘加循环迭代的指令工作量远小于整个输出块的工作量，核心工作负载的任何差异实际上都微不足道。*Stream-K* 无论问题形状如何，都采用理想的分块因子；其通信开销随处理器宽度（而非输出块数量）扩展，并且只需编译为一个内核。

我们使用包含 32,824 种通用矩阵乘法形状和尺寸的庞大语料库，评估了在 NVIDIA 的 CUTLASS 库中实现的 *Stream-K* [8]。与 CUTLASS 采用相同分块因子的数据并行 实现相比，*Strea m-K* 在各种 GEMM 问题上都能提供显著更高的性能

表现，在 NVIDIA A100 GPU 上实现了高达 14× 的加速比。

为了凸显基于集成的解决方案所面临的实际挑战，我们还评估了 NVIDIA 的 cuBLAS 库，以及由预言机驱动的数据并行 CUTLASS 分块集成方案。与这两种集成方案相比，我们表明，单内核*Stream-K* 同时实现了：(1) 更高的平均性能；(2) 更稳定的性能。与 cuBLAS 相比，*Stream-K* 的加速比最高可达 6.7×，且在计算受限问题上几乎不会出现性能下降。

2 背景

通用矩阵乘法（GEMM）定义为乘积$\mathbf{C} = \alpha\mathbf{AB} + \beta\mathbf{C}$，其中 α 和 β 是标量值，**A**、**B**和**C** 是矩阵。（为简单起见，本文始终假设$\alpha = 1$、$\beta = 0$。）我们根据给定 GEMM 问题的计算规模来描述其形状。例如，一个$m \times n \times k$ GEMM 需要$m \times k$和$k \times n$ 两个输入矩阵**A** 和**B**，分别执行$m \times n \times k$次乘加运算，并生成一个$m \times n$输出矩阵**C**。

通用矩阵乘法是许多大规模工程和科学应用中对性能至关重要的子程序。它在 LU 分解、QR 分解和 Cholesky 分解等矩阵分解方法中发挥着重要作用。工程学、气候模拟、宇宙学、量子化学及其他科学领域中的高性能建模与仿真应用都依赖这些分解方法。

矩阵乘法也是现代深度学习（DL）方法的基础构件。深度神经网络（DNN）的训练通常在大型分布式系统上，使用海量数据集进行 [13]。许多 DL 训练和推理操作都可表示为矩阵乘法。例如，图像识别和计算机视觉模型依赖卷积，而卷积可直接实现为滤波器数据集与图像数据集的乘积 [4]。Transformer 架构已成为自然语言处理及其他应用领域的主流，其性能几乎完全受大型矩阵乘积运算的限制。

Larsen 和 McAllister 关于 GPU 矩阵乘法的早期研究，将该计算描述为多重纹理乘法与混合操作 [11]。后续 GPU 架构提供了用户可编程的共享内存，从而支持性能更高的数据并行方案；这些方案采用两级分块（共享内存和寄存器），并通过广泛的微基准测试分析 [2, 14, 17, 19] 和自动调优 [5, 7, 12]来确定数据块大小。

MAGMA GPU 数学库或许是最早针对多样化通用矩阵乘法问题形状进行优化的库之一 [9]。他们的解决方案将一组受限的分块参数应用于模板化的 CUDA C++ 代码模板，为每个 API 基元生成数百种数据并行 变体（例如，用于半精度转置-转置通用矩阵乘法的 hgemm_tt()）。他们评估了这些变体，从中提炼出一个小型集成，通常包含三到五个内核，能够在各种问题形状下整体表现良好。对于给定问题，内核选择和调度由通过简单手写规则表达的尺寸阈值决定。

后续的 GPU 数学库采用了更复杂的代码生成和内核选择组件。例如，ISAAC 项目使用机器学习技术，预测给定 GEMM 形状的最优分块和／或拆分参数配置，然后通过 PTX 级代码生成器在线或离线实例化 [18]。

NVIDIA 的 cuBLAS [15]库提供了扩展的 cublasGemmEx 接口，允许调用方从 24 种不同的 GEMM“算法”中进行选择。使用默认接口时，经过精心训练的启发式方法会在这一庞大的备选空间中进行选择。这些算法实现了多种不同的 数据并行 和 固定分割 变体；为了优化代码，cuBLAS 通常会将每种变体分别组装成架构专用的内核程序。通用矩阵乘法 API 功能、策略变体和微架构的交叉组合，使代码分发规模日益庞大，可执行代码超过数百兆字节。

鉴于当代深度学习发展迅速、变化频繁，近期研究聚焦于编程模型，旨在简化高性能内核的表达与构建，这些内核会改变或补充通用矩阵乘法计算。CUTLASS C++ 库提供了数据移动类和乘累加类，可在 GPU 线程层级的所有层次上组合自定义的类通用矩阵乘法计算 [8]。Triton [19] 是一种领域专用语言，专注于表达、变换和优化块/瓦片概念，以进行张量编程。Halide [16] 和 TVM [3] 等其他领域专用编程语言，将逐点运算符的表达与循环调度分离。Fireiron [6]则进一步将数据移动结构加入调度语法。

3 现有工作分解策略

现代处理器通常将**A**、**B**和**C** 存储在容量大、速度慢且距离较远的内存中，同时可访问容量小、速度快的暂存器或缓存。任何通用矩阵乘法实现的首要目标，都是充分利用这些本地存储资源，使最终实现成为计算受限型。

3.1 顺序缓存分块

通用矩阵乘法的经典缓存分块形式将计算空间划分为多个数据块，并选择一种遍历顺序以展现内存局部性。算法 1给出了一个包含六重循环的简化实现。最内层的三个循环在分块因子 BLK_M,

BLK_N 和 BLK_K 范围内迭代，而最外层的三个循环则遍历各个数据块。如果缓存能够容纳三个矩阵各自的一个数据块，那么这些元素之间的数据复用将显著减少对末级存储器的访问次数 [10]。

算法 1 顺序缓存分块 GEMM。

```
▷ 瓦片处理外层循环
对于 mm ← 0 到 m 步长为 BLK_M执行
    对于 nn ← 0 到n 步长为 BLK_N 执行
        ▷ 将输出数据块初始化为零
        对于 mmm ← mm 到(mm + BLK_M) 执行
            对于nnn ← nn 到 (nn + BLK_N) 执行
                C[mmm,nnn] ← 0
            结束循环
        结束循环
        ▷ 对该数据块执行乘加迭代
        对于 kk ← 0到 k 步长为 BLK_K 执行
            ▷乘加迭代（完全展开）
            对于mmm ← mm 至 (mm + BLK_M) 执行
                对于nnn ← nn 至 (nn + BLK_N) 执行
                    对于 kkk ← kk 至 (kk + BLK_K) 执行
                        C[mmm,nnn] ← C[mmm, nnn] +
                          (A[mmm,kkk] × B[kkk,nnn])
                    结束循环
                结束循环
            结束循环
        结束循环
    结束循环
结束循环
```

3.2 数据并行

如算法 2所示，数据并行图形处理器上的通用矩阵乘法实现被分解到由并行线程块组成的网格中，这些线程块也称为协作线程阵列（CTAs）[1]。网格的大小经过设置，使每个 CTA 都生成自己的 (BLK_M × BLK_N) 输出分块。

为便于说明，算法3的MacLoop()子程序封装了用于计算 CTA 输出分块各元素的乘加工作负载。该子程序在累加域中执行一系列乘加循环迭代，例如通用矩阵乘法中的k轴。每次乘加循环迭代包含每线程体积为 (BLK_M × BLK_N × BLK_K) /CTA_THREADSMAC 操作。随着计算的进行，输入矩阵的片段会被暂存到流式多处理器的共享内存中，以便各个线程在本地重复使用。

尽管 MacLoop() 的这种特定实现方式为每个输出分块元素部署一个线程，但 CUTLASS [8] 和 cuBLAS [8] 中的复杂实现会：(1) 将每个线程的乘加循环迭代完全展开；(2) 在线程束和/或线程级别实施额外分块；(3) 协调软件流水线，在乘加循环迭代之间移动共享内存数据。

[1]图形处理器线程块以协作线程阵列的形式协同调度，从而虚拟化硬件的流式多处理器核心（SM）。

遗憾的是，如图 *1*所示，这种经典的数据并行分解方式在现代 GPU 上容易受到量化效率低下的影响。尽管采用多种不同的分块因子可能会带来提高处理器利用率的机会，但对于任意问题规模，几乎不可能实现完美量化。此外，较小的分块因子有两个缺点：(1) 在流水线实现中，每次乘加循环迭代的指令较少，难以掩盖全局内存和共享内存传输的延迟；(2) 相较于乘加指令，内存操作所占比例更高，这可能导致计算不受限。

算法 2数据并行 GPU 通用矩阵乘法。

```
_共享_累加器[BLK_M,BLK_N]
iters_每个_数据块 ← ⌈k/BLK_K⌉
▷ 为每个输出分块实例化一个协作线程阵列
派生 协作线程阵列[x] 在 [ ⌈m/BLK_M⌉ × ⌈n/BLK_N⌉ ] 执行
    ▷ 执行此数据块的乘加迭代
    累加器 ← MacLoop(x, 0, iters_per_tile)
    ▷ 将累加器存储到输出分块
    StoreTile(C, x, accum)
合并
```

算法 3 用于执行一系列乘加循环迭代的 CTA 范围的 MacLoop() 子程序。

```
过程 MacLoop(tile_idx, iter_begin, iter_end)
    _共享_ 累加器[BLK_M,BLK_N]
    _共享_ 片段_a[BLK_M,BLK_K]
    _共享_ 片段_b[BLK_K,BLK_N]
    ▷ 确定输出分块坐标
    mm ← BLK_M × (数据块_idx / ⌈m/BLK_M⌉)
    nn ← BLK_N × (数据块_idx % ⌈m/BLK_M⌉)
    ▷ 将本地累加器存储初始化为零
    累加器 ← 0
    ▷ 对该数据块执行指定范围内的 MAC 迭代
    对于 迭代 ← 迭代_开始 到 迭代_结束 执行
        kk ← 迭代 ×BLK _K
        ▷将全局矩阵片段复制到本地存储
        片段_a ← 加载片段(A, mm, kk)
        片段_b ← 加载片段(B, kk, nn)
        分叉 线程[mmm,nnn] 位于 [BLK_M、BLK_N]执行
            ▷每线程乘加迭代（完全展开）
            对于 kkk ← 0到 BLK _K do
                累加器[mmm, nnn] ← 累加器[mmm,nnn] +
                  (frag_a[mmm,kkk] ×frag_b[kkk,nnn])
            结束循环
        join
    结束循环
    返回 累加器
结束过程
```

3.3 固定分割

或者，可以通过沿累加维度进行并行化，降低分配给每个协作线程阵列的工作粒度。对于给定的输出分块，加法的结合律允许将迭代域拆分到多个

并发协作线程阵列中，再通过一个依赖性的“修正”步骤对每个协作线程阵列计算出的部分和进行归约。我们重点介绍这种固定分割方法，如算法4所示，其中每个输出分块由s 个协作线程阵列协同生成。值得注意的是，当分割因子s时，它的功能与$s = 1$完全相同。

该固定分割分解方法也应用于 CUTLASS 和 cuBLAS。分割因子作为运行时参数实现，使单个内核可执行程序能够支持多个工作量，同时保留理想的分块因子，以实现最佳的数据共享和延迟隐藏。然而，如图2a所示，通过均匀分块拆分实现完美量化的可能性很小。此外，通信和同步的额外开销会随整体问题规模和分割因子而增加。

算法 4 固定分割 图形处理器通用矩阵乘法，分割因子为 s。

```
1: _共享_累加器[BLK_M,BLK_N]
2: 迭代次数_每_数据块 ← ⌈k/BLK_K⌉
3: 迭代次数_每_分割 ← ⌈迭代次数_每_数据块/s⌉
4: ▷ 实例化s 每个输出分块的协作线程阵列数
5: fork协作线程阵列[x,y] 位于 [ ⌈m/BLK_M⌉ × ⌈n/BLK_N⌉, s] 执行
6:     ▷ 执行此拆分对应范围内的乘加迭代
7:     迭代 ← y ×次乘加迭代_每_个拆分
8:     迭代_结束位置 ←最小值(每个数据块的乘加迭代次数_, 迭代 + 每个拆分的乘加迭代次数_)
9:     累加器 ← MacLoop(x, iter, iter_end)
10:    ▷ 合并各协作线程阵列中的部分和
11:    如果y ≠ 0 那么
12:        ▷ 将累加器存储到临时全局存储
13:        存储部分结果(partials[x,y], 累加器)
14:        发出信号(flags[x,y])
15:    else
16:        ▷ 累加来自其他协作线程阵列、且对当前对象有贡献的部分和
       tile
17:        对于协作线程阵列 ← i到s </style>执行
18:            等待(标志[x，协作线程阵列])
19:            累加器 ← 累加器 + 加载部分结果(部分结果[x，协作线程阵列])
20:        结束循环
21:        ▷将累加器存储到输出分块
22:        存储数据块(C，数据块_id，累加器)
23:    结束条件判断
24: 连接
```

4 我们的*Stream-K* 分解

我们的*Stream-K* 分解采用分块拆分并行化，其拆分接缝与分块结构本身完全解耦。尽管我们采用了熟悉的分块策略来实现数据复用，但我们改为将通用矩阵乘法计算量量化为乘加循环迭代，即由整个协作线程阵列执行的小规模工作单元，每个单元的规模为 BLK M × BLK N × BLK K。正如 _ _ _算法 5中*Stream-K* 所示，它将通用矩阵乘法中乘加循环迭代的总工作量，均匀分配到由固定数量g 个协作线程阵列组成的网格中。每个协作线程阵列所负责的乘加循环迭代范围，会连续映射到通用矩阵乘法形状的$m \rightarrow n \rightarrow k$线性化表示中，并可能跨越输出数据块边界。

算法 5基本 *Stream-K* GPU 通用矩阵乘法，网格大小为 g。

```
1: _共享_累加器[BLK_M,BLK_N]
2: 每个数据块的迭代次数_per_tile ← ⌈k/BLK_K⌉
3: 总迭代次数_m/BLK_M⌉ × ⌈n/ BLK_N⌉ × 每个数据块的迭代次数_tile
4: 每个协作线程阵列执行 _ 次迭代_ ← ⌈总计 _ 次迭代 / g⌉
5: ▷ 实例化 g 个协作线程阵列
6: 派生 协作线程阵列[x]，在 [g] 中执行
7:     迭代 ← x ×次迭代_每个_协作线程阵列
8:     迭代_结束 ← 迭代 +次迭代_每个_协作线程阵列
9:     ▷ 迭代处理外层循环
10:    当 迭代 <迭代结束 执行_
11:        数据块_索引 ← 迭代 / 每个数据块的迭代数_
12:        数据块_迭代 ← 数据块_索引 × 每个数据块的迭代数_
13:        数据块_迭代_结束 ←数据块_迭代 + 每个数据块的迭代数_
14:        ▷执行此数据块的乘加迭代范围
15:        局部迭代 ← 迭代 - 数据块迭代_                    _
16:        本地迭代结束 ←_   _
17:            min(iter_end, 数据块_iter_end) - 数据块_iter
18:        累加器 ←
19:            MacLoop(数据块_id, local_iter, local_iter_end)
20:        ▷合并各协作线程阵列之间的部分和
21:        数据块开始 ← 迭代 =数据块迭代_                    _
22:        数据块_结束 ← (迭代_结束 ≥ 数据块_迭代_结束)
23:        如果 ¬数据块已开始则_
24:            ▷将累加器存储到临时全局存储
25:            存储部分结果(部分结果[x], 累加器)
26:            发出信号(标志[x])
27:        else
28:            ▷将累加器存储到输出分块
29:            如果 ¬数据块已结束那么_
30:                ▷ 累加来自其他协作线程阵列的部分和，这些协作线程阵列也对该数据块有贡
       献到该数据块
31:                cta_end ← 数据块_迭代_结束 / 每个数据块的迭代数_
32:                对于cta ← (x+1) 属于 cta_end 执行
33:                    等待(标志[cta])
34:                    累加器 ←累加器
35:                        + 加载部分结果(部分结果[cta])
36:                结束循环
37:            结束条件判断
38:            存储数据块(C, 数据块_id, 累加器)
39:        结束条件判断
40:        迭代 ← 数据块 _iter_end
41:    结束 while 循环
42: 连接
```

如果某个协作线程阵列的起始迭代和/或结束迭代不与数据块边界重合（通常情况下确实如此），它就必须将自己的部分结果与同样覆盖该数据块的其他协作线程阵列的部分结果合并。在此基本实现中，**C** 中的每个输出数据块都由执行了该数据块$k = 0$ 次乘加循环迭代的协作线程阵列写入。不过，在写入之前，它必须先累加临时全局存储中来自其他协作线程阵列的部分和。值得注意的是，*Stream-K*的通信、同步和全局存储开销均与问题规模无关，而是随协作线程阵列数量g变化。

Stream-K 的另一项优势是，当输出数据块的数量多于协作线程阵列时，同步等待很可能微不足道。在这种情况下，每个输出数据块最多由两个协作线程阵列负责，而分块处理

偏斜确保负责累加的协作线程阵列不会在协作者完成生成贡献值之前就需要这些值。

继续前面的示例，图 2b展示了在假设的四个 SM 的 GPU 上，*Stream-K* 执行 $384 \times 384 \times 128$通用矩阵乘法问题的基本调度。为了充分利用 GPU，我们启动$g = 4$ 个协作线程阵列。假设 BLK_M =为 128，BLK_N = 128，且 BLK_K = 4，每个协作线程阵列负责的 $128 \times 128 \times 288$ 工作量包含 72 次乘加循环迭代。因此，量化效率达到 100%，因为四个 SM 执行的乘加指令数量都相同。

此外，单次乘加循环迭代的工作量比整个输出分块的工作量 32× 小。因此，采用 32 路固定分割分解也能实现 100% 的量化效率，但代价是 8× 更高的“修正”开销。此外，由于共享部分和时写入者与读取者之间存在时间偏移，*Stream-K* 更能有效隐藏协作线程阵列之间同步的延迟。

Stream-K 也可推广到 固定分割和数据并行分解。当网格大小 g 是输出块数量的整数倍时，*Stream-K* 的运行方式与固定分割分解完全相同。类似地，当g等于输出块数量时，*Stream-K* 的运行方式与数据并行分解完全相同。我们利用这一推广特性，在下一节中构建一种针对 *Stream-K* 分解的优化混合方案（5.2）。

5 实现细节

上一节介绍的工作分解可以通过多种不同方式实现，以适应不同硬件架构和软件库设计的需求。我们的实现面向 NVIDIA GPU，旨在集成到 cuBLAS 和 CUTLASS 等现有库中。本节将介绍我们如何配置所启动的内核，并提出一种混合方案，以帮助用户在尽可能广泛的问题形状范围内获得最高的通用矩阵乘法性能。

我们还要强调，这些确实属于实现的内部细节。对于 BLAS 类库的用户而言，它们完全透明，也不会改变库的接口。唯一可观察到的影响是性能特征有所提升，我们将在第6节对此进行分析。

5.1 内核配置

用于对通用矩阵乘法计算进行分块的数据块大小，当然是决定通用矩阵乘法内核性能的关键参数。对于现代 NVIDIA GPU，合适的数据块大小取决于图形处理器 Tensor Core 所支持的矩阵形状。根据大量实测经验，我们为每种支持的精度选择了能够在超大规模通用矩阵乘法中达到图形处理器峰值太浮点运算每秒 99% 的最小协作线程阵列范围数据块大小，适用于大型通用矩阵乘法。对于实验中使用的 NVIDIA A100 图形处理器，这些大小分别为64×64×16 （FP64 问题）和 128×128×32 （FP16→32问题）。

要通过 *Stream-K*并行化实现通用矩阵乘法的最高性能，还需要针对具体问题进行一定程度的动态配置。启动内核之前，我们会选择一个网格大小，以期在当前问题形状下获得最佳性能。这与基于集成的方法形成对比：后者通过静态生成许多基于工作负载分解和分块因子的内核变体，来适应各种问题形状。

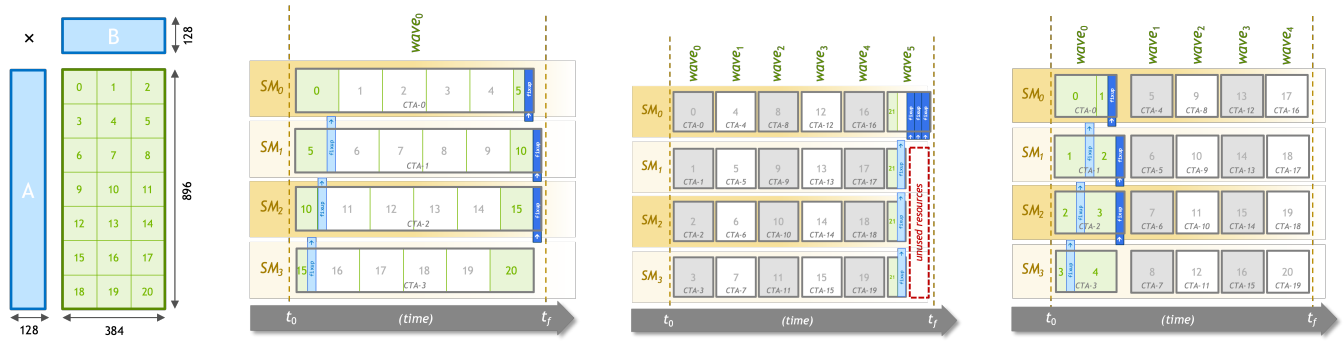
我们的网格大小选择启发式基于一个简单的解析模型，该模型通过均衡分配每个协作线程阵列的乘加循环迭代次数，尽可能降低读取、写入和累加部分和的开销。该解析模型的详细信息见补充材料（附录 A.1）。模型参数可通过实测数据轻松确定，且每种目标架构只需确定一次。随后，便可将所得参数静态编译到库中。同样，这与基于集成的方法形成对比：后者依赖可能复杂的启发式方法和机器学习模型，在运行时进行内核选择。

5.2 数据并行混合化

基本的 *Stream-K* 分解在某些情况下会出现分块处理偏斜，可能对缓存性能产生不利影响。当输出块的数量 t 不是网格大小 g的整数倍时，每个协作线程阵列中首次乘加循环迭代的起始k偏移量将各不相同。根据输入矩阵的大小和形状以及分块因子的不同，这种偏斜可能导致这些片段无法在图形处理器的缓存结构中通过不同协作线程阵列实现数据复用。例如，在图 3a中，四个协作线程阵列各自沿 k 轴的初始片段偏移量依次为 $k = 0$、$k = 32$、$k = 64$ 和 $k = 96$。此外，协作线程阵列之间这种 32 个元素的偏斜会在整个通用矩阵乘法计算过程中持续存在。

分块处理偏斜是 *Stream-K* 工作负载均衡策略的直接结果。不过，我们可以通过将 *Stream-K* 的迭代均衡应用于总迭代域中一个更小且与数据块对齐的区域，限制这种偏斜持续的时间，从而使剩余数据块能够以完整且时间对齐的波次生成。

最简单的混合方案是图 *3b* +所示的“数据并行 *Stream-K*”调度方式。它仅在最终不满的数据并行波次中，对剩余的数据块进行迭代均衡。完整波次的总数为$w = \lfloor t/p \rfloor$，其中 t 是输出块的数量，p 是图形处理器中的流式多处理器核心数。因此，每个 *Stream-K* 协作线程阵列获得均等的迭代份额，且该份额少于一个数据块所需的迭代量。不幸的是，这种策略



(a)基础版 Stream-K　　(b) DP + 单数据块 SK　　(c) 双数据块 SK +DP

图3. 假设在一款四个 SM 的 GPU 上，对 $896 \times 384 \times 128$ 通用矩阵乘法采用基础版 *Stream-K* 与混合执行调度的对比。

几乎无法隐藏三个或更多协作线程阵列处理同一数据块时，交换部分和所产生的同步延迟。在这些情况下，负责累加的协作线程阵列可能不得不等待其他协作线程阵列的贡献变得可见，因为除最后一个协作线程阵列外，其余协作线程阵列几乎会同时完成各自最后的迭代。此外，我们方案中用于聚合部分结果的基本版本在单个协作线程阵列内串行执行，因此，当每个数据块的贡献协作线程阵列数量较多时，很可能导致流式多处理器的工作负载不均衡。

我们采用“*Stream-K*+ 数据并行”双数据块混合调度来解决这些问题，如图 3c 所示。它少执行一个完整的数据并行波次，作为交换，每个 *Stream-K* 协作线程阵列分配到的迭代次数超过一个数据块的工作量（但少于两个数据块）。当 $w \geq 2$ 时，这能显著改善延迟隐藏，而且每个累加协作线程阵列只需接收来自另一个参与协作线程阵列的部分结果。除此之外，该调度的行为与“*DP* + 单数据块 *SK*”调度完全相同。这种混合方法既改善了内存访问模式，也增强了延迟隐藏。它还展示了通用 *Stream-K* 循环结构的灵活性，可在同一个内核实例中实现不同的调度策略。

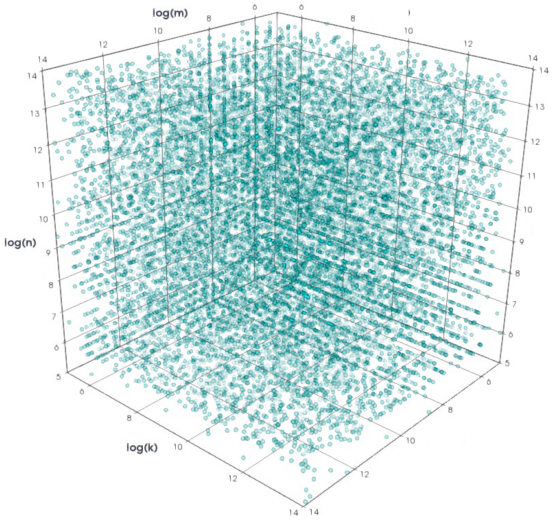


图 4。用于性能评估的 32,824 种通用矩阵乘法问题形状和尺寸的测试域。$\{m\} = \{128 \ldots 8192\}$，$\{n\} = \{128 \ldots 8192\}$，$\{k\} = \{128 \ldots 8192\}$

6 性能评估

我们使用 NVIDIA 的 CUDA C *Stream-K* 模板抽象库 CUTLASS 实现了该分解++，用于编写类似通用矩阵乘法的计算。CUTLASS 提供了优化后的等效实现，对应算法 `MacLoop()`中的协作线程阵列级子程序3，可执行分块、分块处理和软件流水线数据移动，与闭源的 cuBLAS 和 cuDNN 实现类似。我们的评估涵盖两种情况：(1) 双精度 FP64 通用矩阵乘法；(2) 混合精度 FP16→32 通用矩阵乘法。对于后一种情况，输入矩阵**A** 和 **B** 由半精度 FP16 值组成，而内部累加结果及输出矩阵**C**的值则为单精度 FP32。

硬件环境。 我们的测试 GPU 为 NVIDIA A100，包含 108 个流式多处理器核心。为确保测量稳定，我们将功耗上限锁定为 400 W，并将流式多处理器时钟频率锁定为 1005 MHz（为其动态峰值的 ~71%）。由此，FP64 张量核心峰值吞吐量为 13.9 太浮点运算每秒，混合精度 FP16→32 张量核心峰值吞吐量为 222.3 太浮点运算每秒。

数据集。 我们的测试语料库旨在近似反映 GPU 数学内核库所要支持的设备级通用矩阵乘法问题在广度和范围上的巨大规模。如图 4所示，我们评估了 32,824 种不同的问题规模和形状，这些规模和形状是在 m、n和k 矩阵维度的取值范围内按对数尺度随机采样得出的，其体量跨度达六个数量级。

	vs. CUTLASS 64 × 64 × 16	vs. cuBLAS	vs. cuBLAS > 150每字节操作数	vs. CUTLASS 预言机
平均值	1.23×	1.06×	1.03×	1.05×
标准差	0.45	0.10	0.03	0.09
Min	0.77×	0.68×	0.99×	0.70×
Max	5.63×	2.55×	1.24×	1.64×

表1。 *Stream-K* FP64 相对性能

	vs. CUTLASS 128 × 128 × 32	vs. cuBLAS	vs. cuBLAS > 150每字节操作数	vs. CUTLASS 预言机
平均值	1.63×	1.13×	1.15×	1.12×
标准差	1.46	0.45	0.12	0.37
Min	0.80×	0.64×	0.98×	0.61×
Max	14.7×	6.74×	1.85×	4.63×

表 2. *Stream-K* FP16→32相对性能

方法论。 对于两种通用矩阵乘法精度，我们都构建了一个 *Stream-K* 内核，并按照第 5 节中的指南对其进行专门化。此外，这些内核实现了我们的“二数据块 *Stream-K* + 数据并行”混合分解。我们的评估将每个*Stream-K* 内核与以下方案进行比较：

1. 相同分块因子的默认 数据并行 CUTLASS 内核；
2. 该精度对应的 cuBLAS 集成（CUDA 11.6）；以及 3. 一个理想化预言机，它总能为给定的通用矩阵乘法实例选择性能最高的数据并行 CUTLASS 分块因子来执行。

对于 FP64 问题，预言机会从由 {(32×32×16), (32×64×16), (64×64×16), (64×128×16), (128×128×16)} 分块因子特化组成的集成中进行选择。对于 FP16→32，它会从由 {(64×64×64), (64×128×32), (128×128×32), (128×256×32)} 分块因子特化组成的集成中进行选择。这些特定特化是对应 cuBLAS 通用矩阵乘法内核集成的开源严格子集替代方案。

图 6a和图 5a中的“屋顶线”图展示了单个 数据并行 CUTLASS 内核所产生的性能分布。它们绘制了 FP64 和 FP16→32 处理器利用率随计算强度变化的百分比。理想情况下，通用矩阵乘法实现的性能响应应呈现为一个窄带，并紧密贴合机器受带宽和计算能力限制时的性能上限。这里，数据并行 内核在任意给定的算术强度范围内都表现出相当大的动态范围。相比之下，图 *Stream-K* 内核在图 6d 和图 5d 中的性能响应则紧密得多。表 1 和表 2中的数据也印证了这些观察结果，其中 *Stream-K* 内核性能优于其数据并行 FP64 和 FP16→32 对应实现，平均加速比分别为 1.23× 和 1.63×。在极端强扩展场景中，当$m \times n$较小而k较大时，我们的*Stream-K* 内核分别实现了最高 5.63× 和 14.7 × 的加速比。

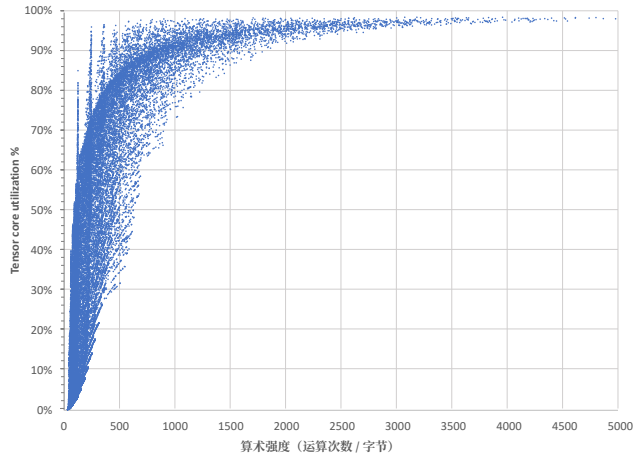
表 1和表 2的第二列比较了我们的*Stream-K* 性能与 cuBLAS 的性能。平均而言，我们的 FP64 和 FP16→32 *Stream-K* 通用矩阵乘法内核的吞吐量分别比对应的 cuBLAS 集成高 6% 和 13%，峰值提升分别为 2。55× 和 6。74×。对于涵盖 32K 种通用矩阵乘法问题形状和尺寸的范围而言，这是显著的改进，而且其可执行代码（每种精度仅需一个内核）20×少于 NVIDIA 的商用通用矩阵乘法库 cuBLAS。

此外，FP64 与 FP16→32cuBLAS 性能响应（图6b 和图 5b）与我们假设的 CUTLASS 预言机集成（图 6c 和图 5c）之间的对比，揭示了设计能够持续提供良好性能的内核选择启发式方法所面临的困难。尽管可以使用相同的分块因子特化，cuBLAS 的动态范围仍显著大于理想化的 数据并行 CUTLASS 预言机。我们 *Stream-K* 内核的性能跨度则更窄，性能最高可达理想预言机的 4。6× 这凸显了它们能够实现以分块为中心的工作分解根本无法达到的利用率水平。

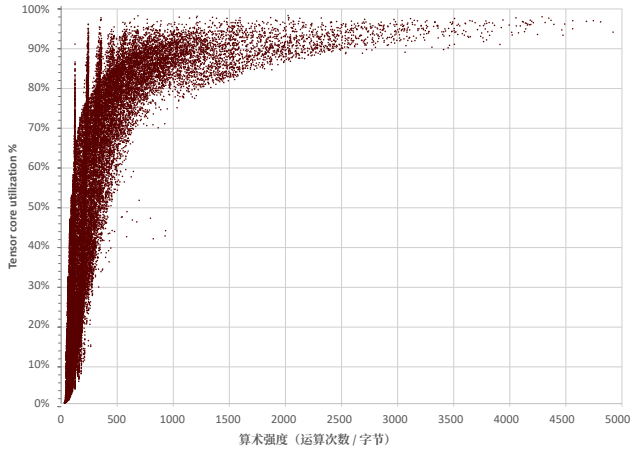
最后，我们观察到，在一些规模较小、受带宽限制的问题形状下，我们较大的分块因子无法与 cuBLAS 竞争。然而，如果将范围限制在计算受限问题的领域（即计算强度为 > 150ops/byte 的 FP64 问题，以及计算强度为→ 32 的 FP16 问题 > 400ops/byte），图 7a和图 7b表明，我们单一的 *Stream-K* 内核的性能全面优于 cuBLAS 集成。在这些阈值以下的区域中，“波动较大”的相对性能并不令人意外，因为*Stream-K* 试图通过增加内存负载来加速内存受限计算。这表明未来有几种研究方向，例如针对内存受限区域进行单独的成本建模，和/或将第二个具有较小分块大小的 *Stream-K*内核纳入双内核集成。

7 结论

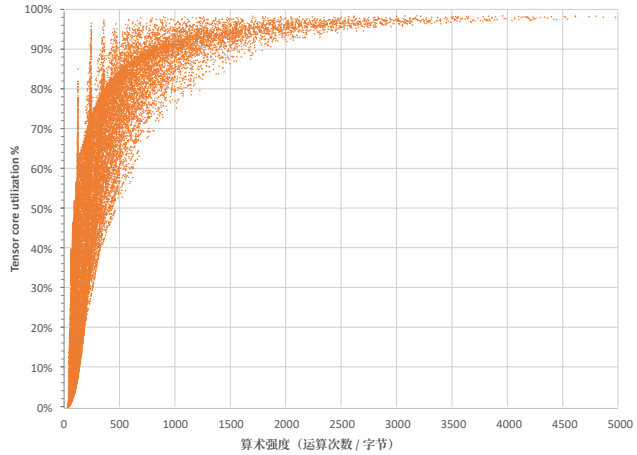
我们介绍了*Stream-K*，这是一种新颖的并行工作负载分解技术，可用于在 GPU 等宽架构上调度通用矩阵乘法（GEMM）及类似计算。与其他分块拆分技术不同，我们以乘加循环迭代作为处理器核心间工作负载量化的单位。这有助于实现出色的强扩展性和工作负载均衡，因为其开销（1）相对于问题形状保持恒定，且（2）远小于整个输出分块的开销。



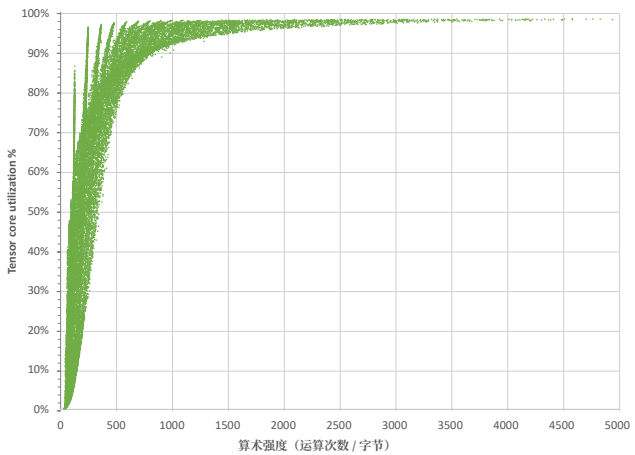
(a) CUTLASS FP16→32数据并行“屋顶线”性能（分块因子 = 128×128×32）。



(b)cuBLAS（集成）



(c) 理想化 CUTLASS 神谕（集成）



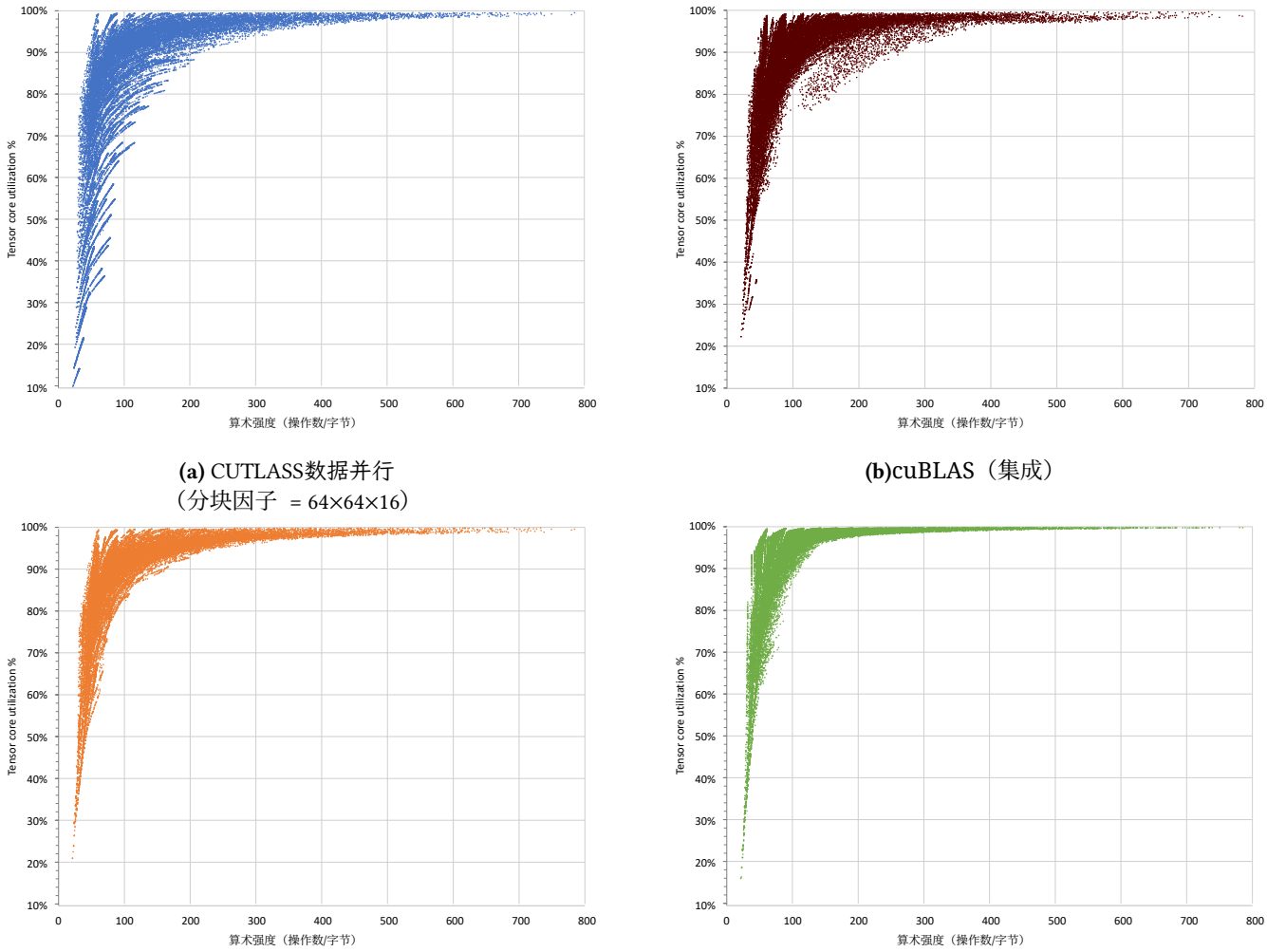
(d)*Stream-K*（分块因子 = 128×128×32）

图 5.FP16→FP32 通用矩阵乘法在 NVIDIA A100 上针对 32K 种通用矩阵乘法问题形状和尺寸的“屋顶线”性能利用率分布。

此外，*Stream-K* 产生的分割边界数量为$O(p)$，且受处理器核心数量限制。因此，强扩展和工作负载均衡的开销随处理器宽度而非问题规模变化。对于许多无法承担分配大量临时存储（其大小相当于问题输出）的应用而言，这是一个很受欢迎的特性。

最后，我们在各种通用矩阵乘法形状和规模上评估了我们的 *Stream-K* 方法。我们证明，单一的*Stream-K* 分块配置能够：(1) 达到与 NVIDIA cuBLAS 库相当或更高的绝对性能，即便后者的处理器利用率接近峰值；(2) 同时实现高得多的性能一致性。此外，*Stream-K* 是库构建与维护的理想选择，因为它有助于将分布规模减少一个数量级，并且无需复杂的手工编写启发式方法或用于内核选择的机器学习模型，也不会牺牲性能。*Stream-K* 已在 CUTLASS 2.11 中开源（https://github.com/NVIDIA/cutlass），使用 CUDA 11.8 编译即可复现本文展示的性能。

对于未来的工作，我们认为，莫顿顺序等缓存感知的分块访问模式是一个值得探索的优化方向。我们还认为，对于其他同样受量化效率低下困扰的通用矩阵乘法类工作负载，*Stream-K* 分解也可能带来类似的性能提升。



(a) CUTLASS数据并行（分块因子 = 64×64×16）

(b)cuBLAS（集成）

(c) 理想化 CUTLASS 神谕（集成）

(d)*Stream-K*（分块因子 = 64×64×16）

图 6。 NVIDIA A100 上涵盖 32K 个问题形状和尺寸的 FP64 通用矩阵乘法“屋顶线”性能利用率分布。

致谢

本材料所述工作由国防高级研究计划局（DARPA）根据合同编号 HR0011-18-3-0007 资助。本文所表达的任何意见、研究结果、结论或建议均属于作者，不一定反映美国政府的观点。“A”类发行声明（批准公开发布，不限发行）。我们谨此感谢 Louis Feng、Valentin Andrei、Zhongyi Lin 和 Serban D. Porumbescu 对本文早期草稿提出的意见。

参考文献

[1] Ahmad Abdelfattah、David Keyes 和 Hatem Ltaief。2016年。KBLAS：用于 GPU 加速器上稠密矩阵向量乘法的优化库。*ACM* 数学软件汇刊 42卷，3期（2016年6月），1–31页。 https://doi.org/10.1145/2818311

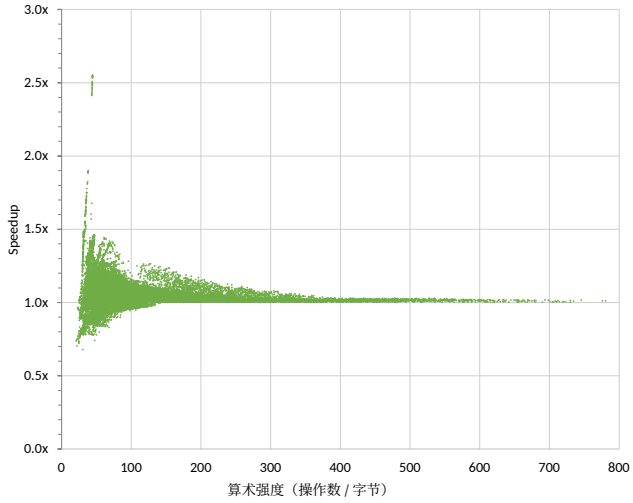
[2] Sergio Barrachina、MaribelCastillo、Francisco D. Igual、Rafael Mayo 和 Enrique S. Quintana-Orti。2008年。图形处理器上三级 CUBLAS 的评估与调优。载于 *2008*年 *IEEE* 国际并行与分布式处理研讨会。IEEE。 https://doi. org/10.1109/ipdps.2008.4536485

[3] Tianqi Chen、Thierry Moreau、Ziheng Jiang、Haichen Shen、Eddie Q Yan、Leyuan Wang、Yuwei Hu、Luis Ceze、Carlos Guestrin 和 Arvind Krishnamurthy。2018。TVM：深度学习端到端优化技术栈。*CoRR* 1802.04799（2018年2月）。arXiv：1802.04799

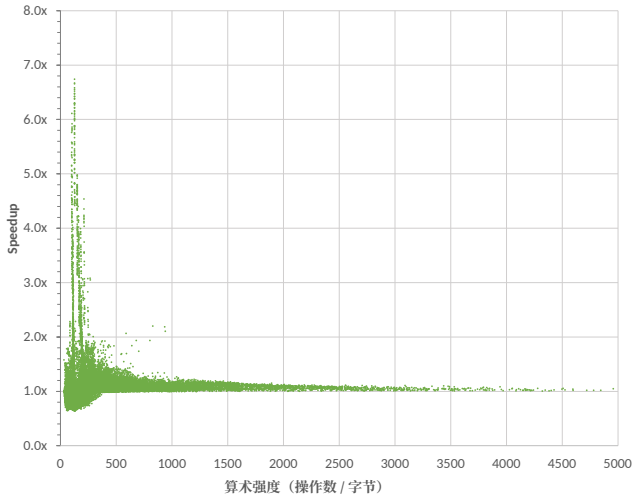
[4] Sharan Chetlur、Cliff Woolley、Philippe Vandermersch、Jonathan Cohen、John Tran、Bryan Catanzaro 和 Evan Shelhamer。2014。cuDNN：深度学习高效基元。*CoRR* 1410.0759（2014年10月）。arXiv：cs.NE/1410.0759v1

[5] Xiang Cui、Yifeng Chen、Changyou Zhang 和 Hong Mei。2010 年。面向 GPGPU 的带缓存稠密矩阵乘法自动调优。载于第 *16* 届并行与分布式系统国际会议论文集（*ICPADS 2010*）。237–242。 https://doi.org/10.1109/icpads.2010.64

[6] Bastian Hagedorn、Archibald Samuel Elliott、Henrik Barthels、Rastislav Bodik 和 Vinod Grover。2020 年。Fireiron：面向 GPU 的数据移动感知调度语言。载于 *ACM* 国际会议论文集



(a) FP64 *Stream-K* 相对于 cuBLAS 的加速比。



(b) FP16→32*Stream-K* 相对于 cuBLAS 的加速比。

图 7。*Stream-K* 相对于 cuBLAS 的加速比。

并行体系结构与编译技术会议（PACT ’ 20 10 1145 3410463 3414632 ）。https://doi.org/ . / .

[7] Changhao Jiang 和 Marc Snir。 . 2005 . 图形硬件上矩阵乘法性能的自动调优。载于第 14 届并行体系结构与编译技术会议论文集（PACT ’ 05 185 194 10 1109 2005 10 ）。– 。

[8] https://doi.org/ . /pact. 。Andrew Kerr、Duane Merrill、Julien Demouth 和 John Tran。 。CUTLASS：CUDA C 中的快速线性代数++。 2017 （2017）。https://devblogs. nvidia.com/cutlass-linear-algebra-cuda/ [9] Jakub Kurzak、Stanimire Tomov 和 Jack Dongarra。2012 年。Fermi GPU 的 GEMM 内核自动调优。IEEE 并行与分布式系统汇刊 23, 11（2012 年 11 月），2045–2057。https://doi.org/10. 1109 2011 311 /tpds. 。

[10] Monica D. Lam、Edward E. Rothberg 和 Michael E. Wolf。1991 年。分块算法的缓存性能与优化。载于第四届编程语言与操作系统的体系结构支持国际会议论文集（ASPLOS IV）。63–74。https://doi.org/ 10 1145 106972 106981 . / 。E. Scott Larsen 和 David

[11] McAllister。2001 年。使用图形硬件实现快速矩阵乘法。载于 2001 年 ACM/IEEE 超级计算会议论文集（SC ’01）。55:1–55:6。 https://doi.org/10.1145/582034. 582089[12] Yinan Li、Jack Dongarra 和 Stanimire Tomov。2009 年。关于 GPU 通用矩阵乘法自动调优的说明。载于国际计算科学会议（ICCS 2009）。884–892。 https://doi.org/10.1007/978-3-642-01970- 8 89 [13] Peter Mattson、Christine Cheng、Gregory Diamos、Cody Coleman、Paulius Micikevicius、David Patterson、Hanlin Tang、Gu-Yeon Wei、Peter Bailis、Victor Bittorf、David Brooks、Dehao Chen、Debo Dutta、Udit Gupta、Kim Hazelwood、Andy Hock、Xinyuan Huang、Daniel Kang、David Kanter、Naveen Kumar、Jeffery Liao、Deepak Narayanan、Tayo Oguntebi、Gennady Pekhimenko、Lillian Pentecost、Vijay Janapa Reddi、Taylor Robie、Tom St John、Carole-Jean Wu、Lingjie Xu、Cliff Young 和 Matei Zaharia。2020 年。MLPerf 训练基准。载于《机器学习与系统论文集》，I. Dhillon、D. Papailiopoulos 和 V. Sze（编），第 2 卷。336–349。[14] Rajib Nath、Stanimire Tomov 和 Jack Dongarra。2010 年。面向 Fermi 图形处理单元的改进版 Magma 通用矩阵乘法。《国际高性能计算应用杂志》24, 4（2010 年 11 月），511–515. https://doi.org/10.1177/1094342010385729

[15] 英伟达公司 2020. CUDA cuBLAS 库（v9.2）。（2020）。 http://developer.nvidia.com/cublas。[16] Jonathan Ragan-Kelley、Connelly Barnes、Andrew Adams、Sylvain Paris、Frédo Durand 和 Saman Amarasinghe。2013。Halide：一种用于优化图像处理流水线中并行性、局部性和重新计算的语言与编译器。载于第 *34* 届 *ACM SIGPLAN* 编程语言设计与实现会议论文集（*PLDI ’13*）。519–530。 https://doi.org/10.1145/2491956.2462176[17] Guangming Tan、Linchuan Li、Sean Treichle、Everett Phillips、Yungang Bao 和 Ninghui Sun。2011。Fermi GPU 上 DGEMM 的快速实现。载于高性能计算、网络、存储与分析国际会议论文集（*SC11*）。美国华盛顿州西雅图， 35 1 35 11 10 1145 2063384 2063431 ： –： 。

[18] https://doi.org/./. 2017 Philippe Tillet 和 David Cox。输入感知的计算密集型高性能计算内核自动调优。载于高性能计算、网络、存储与分析国际会议论文集（*SC17*）。文章 43，共 12 页。 https://doi.org/10.1145/3126908.3126939[19] Philippe Tillet、H. T. Kung 和 David Cox。2019。Triton：一种用于分块神经网络计算的中间语言与编译器。载于第 *3* 届 *ACM SIGPLAN* 机器学习与编程语言国际研讨会论文集（*MAPL 2019*）。10–19。https://doi.org/10.1145/3315508.3329973

A 补充材料

A.1 Stream-K 配置的解析建模

在实践中，调用*Stream-K* 分解并不总是有利的，即使让尽可能多的协作线程阵列同时驻留在图形处理器上也是如此。由于它采用分块拆分方法，因此除了简单的数据并行分解之外，还会产生修正开销。因此，核心问题在于强扩展：在额外开销导致投资回报为负之前，能够表达多少额外并行性？根据问题形状，最佳拆分可能足以填满整个处理器（即$g \leftarrow p$）、完全不拆分（即$g \leftarrow t$），或介于两者之间。

为了预测这一拐点，我们提出一种简单的方法，将*Stream-K* 的运行时间建模为网格大小 g 的函数。在图形处理器上没有其他工作的情况下，整个 *Stream-K* 调度的运行时间将与其中一个负责输出数据块的协作线程阵列相同，我们将其表述如下：

$$time_{CTA}(g) \leftarrow a + b(FixupPeers(g) > 1) + c(ItersPerCta(g)) + d(FixupPeers(g) - 1)$$

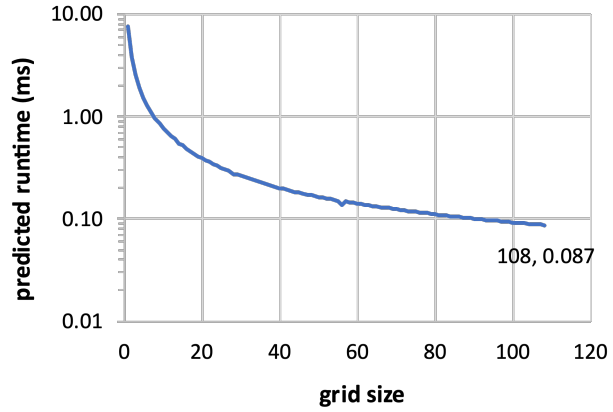
其中：

$$ItersPerCta(g) \leftarrow \left\lceil \frac{\lceil \frac{m}{\text{BLK_M}} \rceil \times \lceil \frac{n}{\text{BLK_N}} \rceil \times \lceil \frac{k}{\text{BLK_K}} \rceil}{g} \right\rceil$$

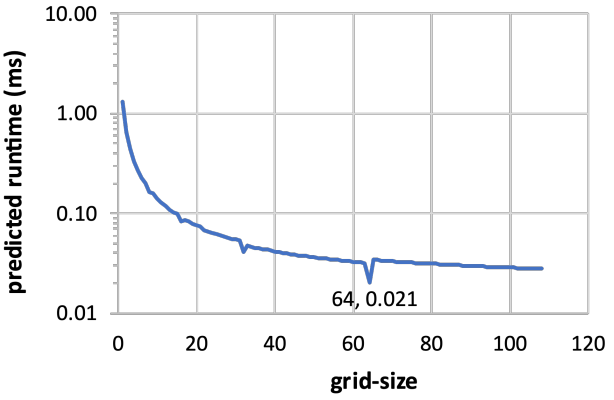
$$FixupPeers(g) \leftarrow \left\lceil \frac{\lceil \frac{k}{\text{BLK_K}} \rceil}{ItersPerCta(g)} \right\rceil$$

该协作线程阵列运行时间模型包含四个部分。a 工作负载涵盖每个协作线程阵列产生的一次性、固定开销，例如网格启动延迟、初始强制缓存未命中，以及将最终输出数据块写入**C**的开销等。第二部分b 纳入了有条件产生的开销：当输出块数量无法在处理器间均匀分配时，需要输出临时部分和。第三部分——每次迭代的工作负载c——表示每次乘加迭代中的指令与停顿工作负载。最后，每个协作者的工作负载d 是从另一个负责覆盖同一数据块的协作线程阵列读取并累加部分和的开销。工作负载常数集合 $\{a, b, c, d\}$因分块因子、矩阵数据类型和图形处理器微架构的不同组合而异，可通过微基准测试进行实测确定。

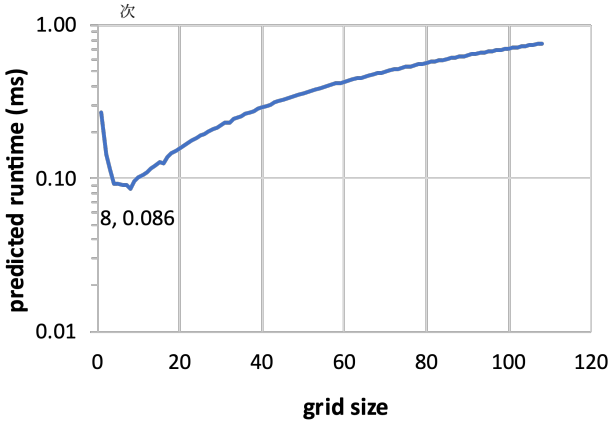
图 8 展示了我们的网格大小选择模型在 NVIDIA A100 图形处理器上针对半精度浮点数通用矩阵乘法进行参数化时的行为，所用分块因子为 BLK_M = 128、BLK_N = 128和 BLK_K = 32。具体而言，我们重点展示



(a) 通用矩阵乘法 $256 \times 3584 \times 8192$
56 个输出块，每个数据块迭代 256 次
$g_{best} \leftarrow$ 108 个协作线程阵列，每个协作线程阵列迭代 132/133 次



(b) 通用矩阵乘法 $1024 \times 1024 \times 1024$
64 个输出块，每个数据块迭代 32 次
$g_{best} \leftarrow$ 64 个协作线程阵列，每个协作线程阵列迭代 32 次



(c) 通用矩阵乘法 $128 \times 128 \times 16384$
1 个输出分块，每个分块 512 次迭代
$g_{best} \leftarrow$ 8 个协作线程阵列，每个协作线程阵列 64 次迭代

图8. NVIDIA A100（108 个流式多处理器）上 BLK_M=128、BLK_N=128、BLK_K=32 配置下*Stream-K* 的性能建模

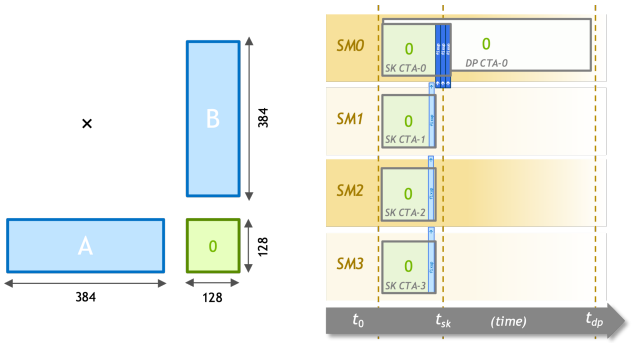


图9。在假设的四个 SM 图形处理器上，对 $128 \times 128 \times 384$ GEMM 的数据并行 和*Stream-K* 执行调度进行强扩展对比。数据并行 会使巨大的k维度在单个协作线程阵列内顺序处理，而*Stream-K* 则能够利用k维度上的并行性。

三种强扩展通用矩阵乘法场景，其中数量输出块数量不足以让 108 个流式多处理器核心上形成一个完整波次。

第一个 GEMM 形状通过较大的k维度进行累加，以生成一个短而宽的输出矩阵。在这种情况下，相对于不断增加的接缝修正开销，乘加循环时间的减少持续改善。因此，最优网格大小对应于$g = 108$个协作线程阵列所能达到的最大并行性。

第二种形状通过中等大小的k维度进行累加，以生成一个包含 64 个输出块的方阵。在这种情况下，b和d的修正开销超过了乘加循环迭代次数减少带来的收益，正如$g = 64$个协作线程阵列处全局最小值的“凹陷”所示。

第三种形态在经过极大的 k维度累加后生成一个输出数据块，与图 9中的执行调度类似。尽管强扩展的空间相当大，但串行归约的每个对等方开销完全由单个协作线程阵列承担。对于网格大小 $g > 8$，这些累加开销开始超过迭代次数进一步减少所带来的收益。