最优软件　　流水线与张量核心 GPU 的线程束专门化　　s

Rupanshu Soi*† Rohan Yadav*† Fredrik Kjolstad AlexAiken†斯坦福大学 斯坦福大学 斯坦福大学 斯坦福大学Maryam Mehri Dehnavi Michael Garland Michael Bauer英伟达 英伟达 英伟达

arXiv:2512.18134v1 [cs.PL] 19 Dec 2025

摘要

GPU 架构的复杂性持续增长，近年来的版本引入了越来越强大的矩阵乘法和数据移动专用固定功能单元，以配合高度并行的通用核心。为了充分发挥这些机器的性能，软件必须采用复杂的调度，以最大限度地利用所有硬件资源。由于实现此类调度十分复杂，程序员和编译器通常都会采用软件流水线（SWP）和线程束专门化（WS）等程序变换，以在实践中实现这一目标。然而，如何最佳地结合使用 SWP 和 WS 是一个颇具挑战性的问题，目前只能通过脆弱的编译启发式方法与容易出错的人类直觉共同处理，对解空间缺乏深入了解。为解决这一问题，我们将 SWP 和 WS 创新性地表述为一个联合优化问题，可由现成的约束求解器以整体方式求解。我们将这一方法实现为 Twill，这是首个能够为大类迭代程序自动推导最优 SWP 和 WS 调度的系统。Twill 不依赖启发式方法，易于扩展到新的 GPU 架构，并且能够保证生成最优调度。我们展示了，Twill 能够重新发现专家为 Flash Attention 在 NVIDIA Hopper 和 Blackwell GPU 架构上手动设计的 SWP 和 WS 调度，从而证明这些调度是最优的。

1 引言

在机器学习对性能永不满足的需求推动下，近期的 GPU 不断扩大用于矩阵乘法（GEMM）的专用功能单元的规模并增强其能力；在 NVIDIA GPU 上，这类单元称为张量核心（TC），此外，批量数据移动单元也在不断发展。随着这些专用功能单元的能力日益增强，通用单元与专用功能单元之间的数据移动与浮点运算次数的相对性能比率会发生显著变化，往往相差数倍。此外，面向专用功能单元的接口在不同架构代际间也经常发生重大变化，涉及数据可放置的位置、发出操作所需的线程数，乃至异步执行模型。因此，每个面向张量核心 GPU 的程序都可能需要针对每一代架构采用不同的最优调度，以充分利用可用的通用单元和专用功能单元 [19,27,33]。

迭代程序的一种常见性能可移植性解决方案是软件流水线（SWP） [6,7,25]。SWP 允许程序员编写简单的循环来描述计算，随后编译器可以自动变换该循环，以利用迭代内部及迭代之间的指令级并行性，从而得出能够最大限度利用所有功能单元的调度。重要的是，将同一个循环编译到不同机器上时，编译器会根据各架构的底层约束定制调度，以确保同一程序在所有环境下都能高效运行 [6]。要实现特定的 SWP 调度，编译器必须合成一个新循环；然而，新型张量核心 GPU 提出的要求（例如需要许多线程协同发出张量核心操作 [2,3],）使得标准顺序循环无法表达大多数调度。

为了在张量核心 GPU 上实现 SWP 调度，编译器 [5, 13,26,36–38] 和开发者 [28,34] 采用了线程束专门化（WS） [10,11] 作为一种编程范式。不同于 GPU 编程的标准数据并行风格（即为每个线程分配相同的计算），在 WS 中，称为线程束的线程子集协作执行程序的不同部分，通过共享内存交换数据，并在必要时进行同步。尽管 WS 是实现张量核心 GPU 上 SWP 调度所必需的程序变换，但它也带来了性能权衡（例如额外的通信和同步），使得在实践中难以确定如何最佳地应用它。

目前，张量核心 GPU 上针对软件流水线和工作共享的方法，或是源于人类直觉 [19,28,33]，或是通过编译器启发式方法 [13, 36, 38]。此外，两者之间的相互作用

*共同贡献。
†本工作完成于英伟达。

尚未得到充分理解，也没有用于分析组合方案最优性的技术框架。这种情况的危害可从 Flash Attention 的案例中看出 [17,18],： Hopper 发布与 Flash Attention 开发之间相隔了一年 3 [33],；后者为 Hopper 提出了定制的软件流水线调度和工作共享策略，而在此期间，关键推理工作负载浪费了无数个计算周期。

为解决这一问题，我们指出，确定最佳 SWP 调度和 WS 策略的问题可以且应该同时解决。我们发现，确定能够最大限度提高资源利用率的 SWP 调度，可以归约为传统的模调度问题 [25,32],，并且可以以最优方式 [22,35],求解，从而针对任何给定的机器模型，获得可能的最大吞吐量。随后，我们解释了为何 WS 是一种依赖于其他变换的程序变换，必须将其纳入联合考量，才能为最新的 TC GPU 架构计算出可实现的 SWP 调度。我们扩展了模调度方法，将确定最佳 SWP 调度和 WS 策略的问题表述为统一的约束满足问题，并由现成的可满足性模理论（SMT）求解器进行整体求解 [9]。

为展示我们的方法，我们介绍 Twill[1],，这是一个系统，可根据具有简单控制流的循环的高级基于分块的描述，自动找出最佳 SWP 调度和 WS 策略。Twill 只需调整与各类 GPU 操作的可量化成本相对应的特定于机器的约束，即可为不同的 GPU 架构生成最佳调度。与我们所知的所有现有 SWP 和 WS 系统相比 [13,36–38],，我们的实现不依赖启发式方法，易于扩展以支持新的 GPU 架构，并能保证为一大类重要的迭代程序（即不含额外控制流的单层嵌套循环）生成最优调度。本工作的具体贡献如下：

- 将为张量核心 GPU 构建 SWP 调度的问题映射为模调度。
- 将 SWP 和 WS 表述为一个基于约束的联合优化问题。
- Twill 是一个实现我们方法的系统，可为实际程序发现最优的 SWP 调度和 WS 策略。

我们将 Twill 应用于 Flash Attention 算法，并为 NVIDIA Hopper 和 Blackwell 架构推导出调度方案。我们展示了，Twill 能够根据注意力机制的高级描述，自动生成与 Flash Attention 3 [33] 和 Flash Attention 4 [19] 中所提出算法相匹配的调度方案。随后，我们实现这些调度方案，并展示由此生成的程序性能可达到 cuDNN 和 Flash Attention 等库中经过手工调优的实现的 1% 以内。

[1]Twill 是一种纺织编织方式，以独特的对角线图案交织经线和纬线。

2 GPU 架构背景

NVIDIA GPU 由若干称为流式多处理器（SM）的独立处理器组成。每个 SM 都包含独立的功能单元，用于支持通用浮点运算或整数运算，还包含用于内存操作的加载和存储单元。SM 还配备了多种类型的内存，包括用于存储线程局部数据的寄存器文件、由软件管理且可供 SM 上所有线程访问的暂存区，称为共享内存，以及可从任意 SM 访问的片外全局内存。

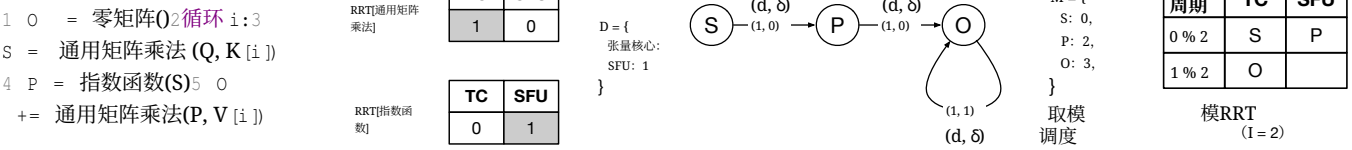
流式多处理器执行由 32 个线程组成的组，称为线程束。线程束采用单指令多线程（SIMT）模型执行：线程束中的每个线程都有自己的指令流，但在给定周期内，只有一部分线程同意执行同一条指令并发射该指令，其余线程则被屏蔽，无法执行。每个线程均按顺序执行：如果线程的下一条指令因未解决的依赖或同步而被阻塞，则无法继续发射任何指令。Hopper 和 Blackwell 流式多处理器拥有四个执行上下文，可容纳活动线程束，因此每个周期最多可有四个线程束发射指令。当线程束数量多于执行上下文数量时，流式多处理器的线程束调度器会动态选择最多四个已准备就绪的线程束来发射指令。

现代图形处理器通过集成异步加速器来处理完整的数据瓦片，从而扩展了这种架构。用于加速 GEMM 操作的张量核心（TC）最早是在 Volta 架构中引入的 [4]。近期，Hopper 架构 [3] 引入了张量内存加速器（TMA）单元，用于在全局内存和共享内存之间异步移动数据瓦片。由于处理的是大型瓦片，一条 TMA 或 TC 指令可能需要执行数千个时钟周期，这与大多数浮点或整数算术指令形成鲜明对比；后者通常只需执行几十个周期。

Blackwell 架构 [2] 与 Hopper 类似，但有两项重要增强。Blackwell 张量核心支持更大的分块大小和更高的吞吐量，并且每个 SM 都包含一种名为张量内存的新型内存，可用于存放张量核心的输入和累加器。要在 Blackwell 张量核心累加器上执行通用计算，必须显式地将数据从张量内存移动到寄存器文件中。尽管这些改动在概念上简单明了，却会显著影响程序的组织方式，才能在 Blackwell 上实现峰值性能。

3 Tensor Core GPU 调度

软件流水线（SWP） [7] 通过重新安排顺序程序中的操作，利用循环迭代内部及迭代之间的指令级并行（ILP），最大限度地提高可用功能单元的利用率。我们使用取模调度 [25,32] 来计算流水化软件流水线调度，并

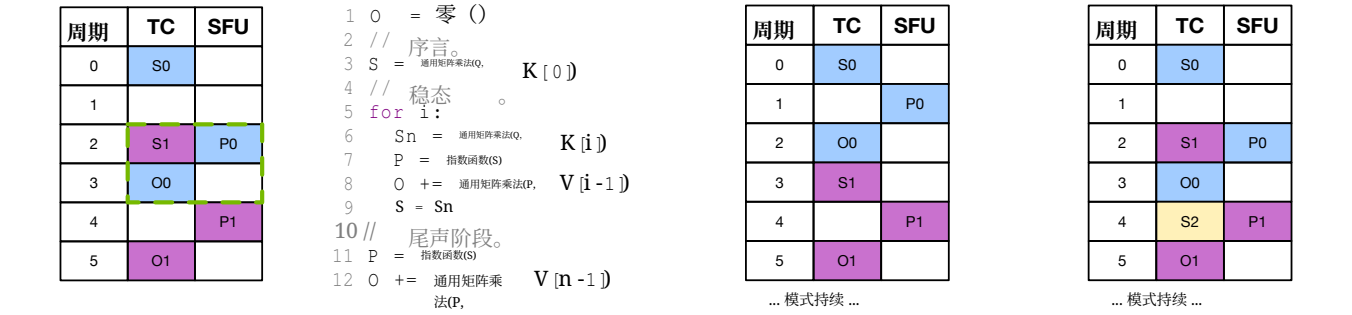


(a) 面向 1个流式多处理器的简化 Flash Attention 伪代码。

(b) 指令RRT 和 Hopper 机器描述。

(c) 图 1a 的循环依赖图表示1a。

(d) 有效的模调度和模RRT，其中 $I = 2$、$L = 4$。



(e) $\lceil L/I \rceil$ 图 1d 的副本1d's 调度间隔I 个周期。

（f）由图生成的代码1e 采用稳态循环。

（g）按序执行，实现 1/3个。每周期。

(h) 流水线执行，实现 1/2个。序言之后每周期。

图1：以基于分块的方式表示的简化版 Flash Attention 的模调度。机器成本以 Hopper 为准，在 Hopper 上，通用矩阵乘法 和 指数函数 在一个分块上的开销大致相同。模调度恢复了 Flash Attention 的 3 [33] 流水线。

第 3.1 节借助图 1所示的运行示例介绍标准术语。该图展示了如何对简化版 FlashAttention 进行流水线化 [18]。代码（图 1a）、指令开销信息（图 1b）以及依赖图（图 1c）经过两个中间步骤（图 1d 和 1e），最终得到图 1f中的流水线化代码。在此示例中，由于第二次矩阵乘法之前的结果存在数据依赖，程序暴露了指数运算的延迟，因此要实现最大张量核心利用率，必须采用 SWP（图 1g）。不过，正如我们在第 3.2 节所讨论的，在较新的架构上，单个执行线程可能无法体现这种流水线调度。为解决这个问题，我们将在第 3.3 节介绍如何使用 WS，将操作分配到多个执行线程中，从而实现最优流水线调度，以及由此带来的编译挑战。

3.1 模调度

模调度 [22, 25, 32] 是软件流水线（SWP）的一类常用算法。它会转换给定的循环，使其在遵循输入程序的数据依赖关系以及底层机器容量限制的同时，达到最大吞吐量（单位时间内完成的迭代数）。在模调度中，输入循环由依赖图表示$G = (V, E)$ 其中 V 是循环中的指令集合，而E 定义指令之间的依赖关系。我们假设 G 描述的是一个没有控制流的单层嵌套循环。在 Twill 中，每条指令表示一个瓦片级操作，该操作利用整个流式多处理器（SM）的资源；此类依赖图可以从标准的基于数据块的编程模型中提取（参见第 5 节）。

每个$v \in V$ 都关联着一个资源预留表（RRT），它指明了执行v期间所使用的功能单元。RRT$[v]$ 是一个二维整数数组，其中每列对应一种功能单元，每行对应执行v的一个时钟周期，表项则给出在该执行时钟周期内，被f 占用的v 实例数量。机器上f 的实例总数称为f的容量，由机器描述D给出。运行示例中每个操作对应的 RRT，以及机器描述D 如图1b所示；两条指令都只需一个周期即可执行，但占用的功能单元不同。为简单起见，我们假设每个功能单元每个周期都可以调度并执行一条指令；而在真实机器上，不同功能单元执行某些指令的时间通常更长，并且对其接收新指令的频率也有限制。对于处理大型数据瓦片的张量核心 GPU，RRT 这一抽象在更粗粒度的计算层面仍然适用，此时一个“周期”可能代表更粗的时间粒度。我们将在第5.2节介绍如何处理确定时间粒度这一问题。

每条边$e \in E$都是一个元组（u，v，d δ），其中u是数据依赖的源，v 是汇。时钟周期延迟 $d \geq 0$ 表示v 必须至少在d 个周期之后发射

u 发射。最后， $\delta \geq 0$ 是迭代延迟，表示循环迭代 i 中 v 的实例必须至少在循环迭代 i $-\delta$ 中 u 的实例之后 d 个周期发射。例如，一条带有 $\delta = 1$ 的边表示 u 和 v 之间的依赖会跨循环传递到紧接着的下一次迭代。注意力示例的依赖图如图1c所示，其中来自 S 和 P 的边具有 d $= 1$ （来自相应的 RRT）和 $\delta = 0$，而 O 上的循环携带依赖具有 $\delta = 1$。为简化说明，图中使用变量名（这些名称均唯一）而非指令名。每条指令的 RRT（图 1b）与依赖图 G（图1c）共同构成模调度的输入。

对给定循环运行时，取模调度会找到一个启动间隔 I 和模调度 M，并据此构造流水线循环。I 对应于新循环迭代被发射（或完成）的速率，因此与吞吐量成反比；$I = 1$ 是可能的最小值，意味着每个周期都会启动一个新的循环迭代。M 将每条指令映射到其必须被发射的时钟周期，该周期从其所属迭代的开始处计数（因此得名“取模”）。如果$M(v) = k$，那么在流水线循环中，v 将在时钟周期k、$I + k$、$2I + k$.....被发射。当M 和I 的选择能够确保流水线循环满足E 中的所有依赖边，并使所有功能单元均不超出容量时，该选择就是有效的。

运行示例的一个有效模调度如图1d所示。与之配套的是模RRT，这是一种类似于RRT的数据结构，用于表示流水线循环稳态下的功能单元使用情况。在模调度中，P和O都比仅针对单次迭代的最优调度晚一个周期。根据数据依赖，M(P) $= 1$ 和M(O) $= 2$ 本来是有效的，但根据模RRT则无效，因为O会与S在TC上发生冲突，因为 $0 \equiv 2$ (模2)。人们已经开发出多种算法，用于针对给定依赖图G构造M和I，其中包括贪心 [25, 32] 算法，以及利用整数线性规划（$\mathbb{Z}$LP）的最优 [22, 35]算法。最优算法能够生成具有尽可能小I值、因而具有尽可能高吞吐量的模调度。

一旦确定了模调度 M 和启动间隔 I，便可以用它们合成一个执行该调度的程序。实现模调度的一种标准方法 [6] 是使用一个顺序循环，其中包含：1）用于预热循环的序言，2）包含重复执行的循环体的稳态，以及 3）用于排空循环的尾声阶段。要构造这三个部分，令 L 为M的长度，即M中的周期数。软件流水线循环通过重叠$\lceil L/I \rceil$ 个M副本构造而成，每个副本都比前一个副本偏移I 个周期。随后，可以从这个交错调度中读出流水线循环：前（$\lceil L/I \rceil - 1$）·I个周期对应序言，接下来的I 个周期对应循环稳态，其余周期则是用于排空流水线的尾声阶段。

运行中的注意力示例的模调度排布如图1e所示，其中蓝色和紫色分别表示调度的不同副本，虚线框表示稳态。构造好调度后，即可用它生成代码。该示例生成的伪代码如图 1f所示。按照图 1a 所示按序执行时，每 3 个周期完成 1 次循环迭代（图 1g）；而经过序言阶段后，流水线执行每 2 个周期完成 1 次迭代（图 1h）。

对正在运行的 FlashAttention 注意力示例应用模调度，仅凭算法和机器的高层描述，就能得出 Flash Attention 专家设计的精确流水线 3 [33] 。

3.2 代码生成挑战

对于 Hopper 之前的图形处理器，图 1f 中经过模调度的瓦片级程序可以降低为以单指令多线程方式执行的单线程代码，其中每个数据并行线程都会在数据瓦片互不重叠的子集上执行相同的程序。遗憾的是，针对 Hopper 或 Blackwell 图形处理器功能单元计算出的多数模调度，无法以标准方式实现为单线程程序。有四个复杂因素经常阻碍单线程代码生成。首先，也是最常见的情况，近期图形处理器中张量核心的规模要求多个线程束协作发出大型通用矩阵乘法计算。大多数图形处理器编译器基于顺序中间表示（例如 LLVM 和 PTX）构建，因此并不适合为协作式线程束生成代码。

第二个问题源于模调度倾向于增大计算的工作集，以便让程序多个迭代中的活跃变量同时保留在片上。模调度所带来的这种固有内存压力，再加上为更大的张量核心提供数据所需的数据量增加，使得将计算的工作集保留在片上变得颇具挑战。大多数图形处理器对每个线程可用的寄存器数量设有严格上限，即 255 个；将数据从寄存器文件溢出通常会造成显著的性能损失，因为溢出的数据不太可能继续留在缓存中。对于近期的图形处理器，许多采用单线程实现的模调度都难以达到峰值性能，原因在于溢出代价高昂。

第三个复杂之处在于，一些操作（例如通过多层级GPU内存层次结构进行的传输）在执行时间上具有很大的波动。虽然传统的模调度建议对此类开销采用一个上界 [6]，但这隐含地假设该上界与调度中的其他操作大致处于同一数量级。然而，像TMA传输这样的操作会在全局内存和共享内存之间移动数据瓦片，同一传输的最快与最慢执行时间可能相差一个数量级以上。尽管这些操作是异步执行的，

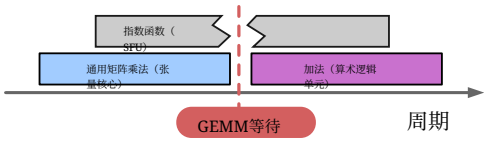


图2：在同一线程束上调度、使用不同功能单元的三个操作的可视化。GEMM 后的阻塞式同步会中断 EXP 的并发发射。

但潜在延迟的动态范围很大，可能会妨碍在代码生成期间安排同步指令的尝试。具体而言，高估会导致利用率不足，而低估则会导致执行期间出现流水线停顿。

最后，Hopper 和 Blackwell 上的固定功能单元采用异步接口，因此在使用结果时需要显式执行阻塞同步。由于 GPU线程采用顺序指令发射，阻塞同步会中断同一线程束上任何并发调度操作的指令发射。图2展示了一个例子：模调度要求加法使用通用矩阵乘法的结果，同时并发执行一个独立的指数函数。请注意，这三项操作分别占用独立的功能单元。若由同一线程束执行所有操作，性能会下降，因为GEMM等待 会中断指数函数的指令发射。

这四个因素对模调度代码生成的影响程度各不相同，具体取决于特定程序的属性。在实践中应对这些因素，需要采用一种新的代码生成方法。

3.3 线程束专门化

线程束专门化 [10,11]（WS）是一种编程范式，可解决在图形处理器上实现模调度时遇到的代码生成难题。线程束专门化程序会将操作分配给不同的线程束，同时确保这些线程束协同执行完整计算。得益于流式多处理器（SM）内线程的层次化分组，WS 才得以实现。由于线程束采用单指令多线程（SIMT）执行模式，当线程束中的所有线程发出相同指令时，其性能最高；而当线程发生分歧时，性能最低。不过，与线程不同，线程束彼此独立，不会因控制流分歧而受到性能惩罚。

尽管工作共享（WS）通常被视为一种独立的优化，但我们的见解是，WS 之所以有用，恰恰是因为它能直接解决第 3.2 节中介绍的所有软件流水线（SWP）代码生成问题。首先，WS 会自然地考虑线程束之间的协作，并能确保部分线程束协同发出张量核心操作。其次，将计算拆分到多个线程束上后，计算可以使用许多线程的寄存器资源，从而更灵活地将 SWP 所需的大型工作集装入片上。第三，可变延迟操作可以分配给专用线程束，而不必与固定延迟操作一起进行静态调度（并相互干扰）。最后，将操作拆分到多个线程束上，可以让某些线程束发出指令，而其他线程束则因等待同步而被阻塞。

由于 WS 解决了 SWP 代码生成所面临的挑战，因此已被编译器广泛采用 [5,13、26、36–38] 以及程序员 [19,28,33]。不过，这也带来了两个复杂问题。首先，WS 并非没有代价，它需要在不同线程束之间通信数据，并协调对共享数据和硬件资源的访问，因此存在相应的权衡。目前，我们所知的所有自动化方法都依赖临时启发式方法来决定这些权衡 [13,36,38]。这些启发式方法通常会为线程束设定典型的“角色”（例如加载线程束和计算线程束），或将线程束视为可分派工作的并行“代理”。若缺乏用于理解这些启发式方法是否最优的技术框架，就难以判断借此生成的程序能否达到峰值性能。

这种方法的另一个重大复杂之处在于，即使采用 WS，也未必能够合成出真正实现模调度的程序。例如，即使在流式多处理器（SM）中将计算尽可能分散到更多线程束上，模调度所需的工作集仍可能无法装入寄存器文件。因此，在编译程序时，若将模调度的计算与使用 WS 进行代码生成视为两个独立步骤，实际生成的代码可能并非最优。为了确保模调度确实能够实现，使用 WS 进行代码生成的约束需要与传统模调度的约束一同直接纳入优化过程。

4 联合优化问题

现在，我们展示如何同时求解寻找模调度和 WS 策略的优化问题。联合优化问题的输入是程序 $G = (V, E)$，输出则是启动间隔最小的模调度，以及能够实现该调度的 WS 策略。结果包括模调度 M^*、启动间隔 I^* 和线程束分配A^*，其中 A^* 是将每个$v \in V$ 分配到一个或多个线程束的方案。

我们通过两个主要步骤来处理这个问题。首先，利用模调度推导出初始模调度 M和启动间隔 I，在满足数据依赖和功能单元约束的同时实现最大吞吐量。随后，以M 和I 为种子构建一个约束系统，并补充 WS 约束。然后，SMT 求解器可以求解该约束系统，找出一个满足模调度和 WS 双方要求、且具有相同I 的M^* 和A^* 。

我们使用 M 和 I 来定义一个初始直线程序 Q，并据此构建约束系统。Q 是通过第 3.2 节中的模调度代码生成过程得到的。回想一下，$\lceil L/I \rceil$ 个 M 副本相互重叠，每个副本相对于前一个副本偏移 I 个周期，从而形成序言、稳态和尾声阶段。我们的洞见是，我们可以

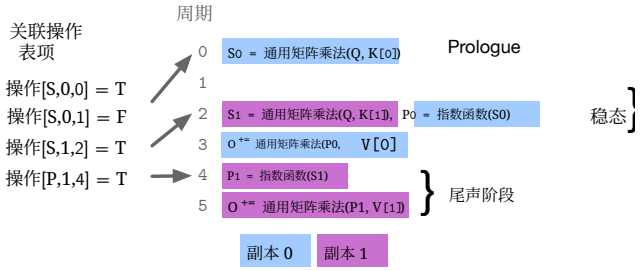


图3：由 Twill 的联合公式分析的直线代码。示例操作表项位于左侧。

$$\forall v,i \quad \sum_t \mathrm{op}[v,i,t] = 1 \qquad \text{(UNIQUENESS)}$$

$$\forall v,i \in [1,\lceil L/I\rceil),t \quad \mathrm{op}[v,0,t] \Rightarrow \mathrm{op}[v,i,t+i\cdot I] \qquad \text{(CONSISTENCY)}$$

$$\forall v,i,t \quad t+\mathrm{cycles}(v) > T \Rightarrow \neg\mathrm{op}[v,i,t] \qquad \text{(COMPLETION)}$$

$$\forall i,t,(u,v,d,\delta)\in E,\ t'\in[0,t+d) \quad \mathrm{op}[u,i,t] \Rightarrow \neg\mathrm{op}[v,i+\delta,t'] \qquad \text{(DEPENDENCE)}$$

$$\forall t,f \quad \sum_{v,i,c\in[0,\mathrm{cycles}(v))} \mathrm{op}[v,i,t-c]\cdot \mathrm{RRT}[v][f,c] \le \mathrm{cap}(f) \qquad \text{(CAPACITY)}$$

图4：确保模调度有效的约束。

假设稳态恰好执行一次，这样便可将这三个部分视为长度为 T 的直线程序。这个假设是合理的，因为稳态可以只执行一次后便结束；而且，由于各次稳态执行完全相同，这个假设是完备的。此外，它大大简化了约束，无须进行任何循环分析。图 7 Q 展示了一个由图 1 中的模调度示例推导出的程序，位于图 3 的右半部分。

4.1 带约束的模调度

联合问题的目标是找到一个模调度 M*，它可以不同于 M，但实现相同的启动间隔。为此，Twill 必须能够修改 M，这就需要以 SMT 约束的形式重新实现模调度。该实现不直接修改 M，而是允许将 Q 修改为 Q* ，使得 Q* 与 Q 具有相同的启动间隔和长度，但可能来自不同的模调度。实现这一目标的约束见图4。我们定义一个三维布尔数组，使得 op[v,i,t] 表示操作 v ∈V 在迭代 i ∈ [0, ⌈L/I⌉) 中被调度到 Q*的时钟周期 t ∈ [0,T)。唯一性约束确保每个操作都恰好被调度一次。一致性约束确保 Q* 可以由某个模调度得到。完成约束要求所有操作都在 Q*的最后一个时钟周期 T 之前完成。最后，依赖约束和容量约束的作用与模调度中的相应约束相同。为简洁起见，我们省略守卫条件，但假设引用 i ∉ [0, ⌈L/I⌉) 的约束

$$\forall t,m \quad \sum_{v,i} \mathrm{live}[v,i,t]\cdot \mathrm{footprint}(v,m) \le \mathrm{capacity}(m) \qquad \text{(内存容量)}$$

$$\forall v \quad (\exists(v,u,d\ \delta>0)\in E) \Leftrightarrow \text{存活}[v\ \lceil L/I\rceil - 1,T] \qquad \text{(INIT)}$$

$$\forall v,i,t \quad (\text{存活}[v,i,t]\wedge \mathrm{op}[v,i,t]) \Rightarrow \neg\text{存活}[v,i,t-1] \ \text{(LIVEPROP-1)}$$

$$\forall v,i,t \quad (\text{活跃}[v,\ i,\ t]\wedge \neg\mathrm{op}[v,\ i,\ t]) \Rightarrow \text{活跃}[v,\ i,\ t\ -1] \ \text{(LIVEPROP-2)}$$

$$\forall v,\ i,\ t \quad (\neg\text{活跃}[v,\ i,\ t]\wedge \bigvee_{(v,u,_,\delta)\in E} \mathrm{op}[u,\ i+\delta,\ t]) \Rightarrow \text{活跃}[v,\ i,\ t-1] \qquad \text{(DEADPROP-1)}$$

$$\forall v,i,t \quad (\neg\text{活跃}[v,i,t]\wedge \bigwedge_{v,u,_,\delta)\in E} (\quad \neg\text{运算符}[u,i+\delta,t]) \Rightarrow \neg\text{存活}[v,i,t-1] \qquad \text{(死属性-2)}$$

图 5：内存分配约束。

不会被输出。此外，op 中对 $t \notin [0,T)$ 的引用会被映射为 false。op 的一个满足赋值会导出有效的 Q^* 和 M^*，其中 $M^*(v) = t$ 当且仅当 $\mathrm{op}[v,0,t]$。

4.2 内存感知约束

为确保模调度的工作集能够保留在片上，我们引入额外约束来强制满足内存容量要求，从而缩小 Q* 有效实例的集合。这些约束假设 Q* 采用静态单赋值形式（SSA） [8,16],, 即每个变量只定义一次，且其生存期在最后一次使用时结束。我们定义第二个布尔数组 live[v, i,t],，其中 live[v,i,t] = 1 表示 v 的第 i 个实例的结果在时间 t 处于活跃状态。利用 live，定义内存容量约束十分直接，如图 5 所示；内存占用和容量由输入图和机器模型定义。内存容量约束中较为复杂的部分是建立用于定义 live[v,i,t] 的约束；这部分无法交由外部分析处理，因为对 Q* 的更改会影响不同值处于活跃状态的时机。这些约束见图 5，其推导基于经典的活跃性逆向数据流算法 [6]。活跃性约束分为三组：初始条件、活跃性传播和非活跃性传播。初始时，只有 Q* 中每条指令最后一个副本的循环携带结果在最后一个时钟周期处于活跃状态，即在时间 T 处于活跃状态。随后，活跃性沿直线程序逆向传播：如果某个操作的结果在时间 t 处于活跃状态，那么在时间 t −1 也处于活跃状态，除非该操作被调度在时间 t。非活跃性以类似方式传播：如果某个操作的结果在时间 t 处于非活跃状态，那么在时间 t −1 仍处于非活跃状态，除非该操作的某个使用者被调度在时间 t。

4.3 Warp 分配约束

最后，我们引入一些会影响 WS 决策的约束，如图 6所示。我们定义另一个布尔数组 opw[v,w],，用于指示操作 v 是否已分配给线程束 w；

$\forall$v $\sum$opw[v,w] $=1$ (线程束唯一性) w$\forall$v 变量_延迟(v) $\Leftrightarrow$ opw[v,W$_{vl}$] (可变延迟)$\forall$t,w $\sum$live[v,i,t] $\cdot$opw[v,w] $\cdot$ regs(v) $\leq$寄存器_限制() v,i (寄存器限制)$\forall$(u,v,d δ) $\in$ E,t,i,w,w$'$ $\neq$w,s $\in$ [0,spillcost(u)) op[u,i,t] $\wedge$ opw[u,w] $\wedge$ opw[v,w$'$] $\Rightarrow\neg$op[v,i $+\delta$,t $+$d $+$s](跨线程束溢出)$\forall$(u,v_,_) $\in$E,t,w,i,o $\neq$v op[v,i,t] $\wedge$ opw[v,w] $\wedge$ blocking(u,v) $\Rightarrow\forall$i$'$, t$'\in$ [t $-$(周期(o) $-$ 1),t], $\neg$(op[o,i$'$,t$'$] $\wedge$opw[o,w]) (并发性)

图 6：线程束分配约束。

opw 直接定义最终的线程束分配 A^*。与 op 类似，我们定义了一个唯一性约束，即每个操作必须且只能分配给一个线程束。该约束可自然扩展至跨多个线程束的操作，例如 Hopper 和 Blackwell 上的线程束组级操作。

第一项约束是将具有可变且静态未知延迟的操作分配到指定线程束 W$_{vl}$，使其与具有静态已知且固定延迟的指令的调度和放置分开。第二项约束对每个线程束施加寄存器限制；该约束与图 5 中的容量约束类似，但求和范围限定为每个线程束。此约束迫使 V 的划分满足寄存器限制。然而，除非通过共享内存传输（或溢出）数据，否则线程束 w 中寄存器里的数据无法在线程束 w$'$ 上访问。当数据跨线程束溢出时，有效调度必须在调度使用该数据的操作时计入传输开销；跨线程束溢出约束会纳入这部分额外延迟。我们假设跨 warp 通信的开销以注释的形式提供在 G 的节点上。此约束迫使人们在内存容量、延迟和跨线程束溢出的开销之间进行权衡。最后一项约束是并发约束，它描述了阻塞同步中断并发调度的操作发射所产生的影响。某些边 (u,v) 被指定为阻塞边，表示在可以访问 u 的结果之前，v 必须执行阻塞同步。并发约束规定：如果分配到线程束 w、且调度时间为 t 的操作 v 需要对某个入向依赖执行阻塞同步，那么在 v 开始时，不得在 w 上调度其他操作；这迫使并发操作要么放置到不同线程束上，要么安排在不同时间执行。为简洁起见，此处省略了实现中类似的跨线程束溢出约束，因为通过共享内存传输寄存器数据需要阻塞同步。

整套约束系统涵盖了第 3 节中提出的所有挑战；这些挑战可能导致近期图形处理器上所需的模调度方案无法在实践中实现。通过用统一的通用约束系统全面描述解空间，我们迫使约束求解器同时考虑调度问题的所有方面。此外，随着新机器的发布，可以轻松添加、删除和修改约束，以应对未来架构带来的挑战。因此，我们的方法为求解模调度与工作共享（WS）的联合问题提供了灵活的框架。

5 实现

为落实我们的方法，我们推出了 Twill，这是一个为 Triton [36]程序计算联合 SWP 和 WS 策略的系统。Twill 从一种名为 TTGIR 的中级 Triton 中间表示（IR）中提取依赖图。TTGIR 是一种基于分块的 SSA 中间表示，支持对分块进行算术运算并显式表示数据移动。这些图作为输入提供给 Twill 的优化过程，详见第 4 节。Twill 用户还必须指定目标 GPU 架构。目标架构用于估算指令和数据移动的成本（可通过文档 [1] 或直接测量获取），并支持架构特定建模，例如声明可用的存储器，或标明需要阻塞同步的操作。

如第 4 节所述，Twill 通过将问题表述为 $\mathbb{Z}$线性规划问题 [35] 并将其交由 CBC 求解器 [21]求解，计算初始模调度。随后，Twill 使用得到的模调度和启动间隔来初始化 SMT 约束系统。Twill 将 SMT 查询交由 Yices2 SMT 求解器的量词自由线性整数算术（QFLIA）理论处理 [20]。发现模调度和线程束分配后，Twill 使用标准代码生成技术生成软件流水线中间表示，并为每条指令标注应执行该指令的线程束。生成的中间表示既可供实现指定 WS 策略（例如 Tawa [13] 或 Cypress [38]）的下游编译器使用，也可供专家参考，以便手动实现。

5.1 处理不可满足性

Twill 必须处理约束系统不可满足的情况。不可满足表明，某种资源限制或并发交互阻碍了以初始启动间隔I实现模调度。与模调度中的标准方法类似， [25], 如果约束不可满足，Twill 会使用模调度寻找一个新的候选调度M'，其启动间隔为$I+1$，随后用它为约束系统提供种子，以便

算法1 Twill 的搜索过程

```
过程TWILL(G)
    I ← 0
    当 true 时执行
        I ← I + 1
        M ← OPTIMAL-MODULO-SCHEDULE(G, I)
        如果 M = 失败，则
            继续
        L ← 长度(M)
        当 ⌈L/I⌉ = ⌈LEN(M)/I⌉ 时
            (M*, A*) ← SWP-AND-WS(G, M, I, L)
            如果 (M*, A*) = 失败，则
                L ← L + 1
                继续
            返回 (M*, I, A*)
```

再次尝试求解。通过从可能的最小启动间隔开始单调递增地搜索，我们确保找到满足约束的最高吞吐量调度。除了搜索候选值 I外，Twill 还会搜索不断增大的 L 值（调度总长度），这些值不会影响在给定 $\lceil L/I \rceil$下的结果。I。此搜索过程如算法1所示。

5.2 成本归一化

Twill 实现中的一个关键组成部分是 成本归一化，它使 Twill 求解的优化问题变得易于处理。请看根据公开文档 [1] 计算出的 Hopper 上 128x128x128 通用矩阵乘法的周期计数——约为 1000 个周期。一个合理的依赖图 $G = (V, E)$ 可能包含许多周期计数相近的指令。模调度的最优算法和 Twill 的联合表述不仅关于 $|G|$呈指数级增长，而且关于 $\sum_{(u,v,d,\delta)\in E} d$也呈指数级增长。因此，直接使用估算的周期计数会导致难以处理的 $\mathbb{Z}$线性规划和 SMT 问题。

我们的解决方案基于这样一个观察：将软件流水线问题中的每个周期计数都乘以一个正整数，会得到一个与原问题同构的新问题。关键不在于周期计数本身，而在于周期计数之间的比率。因此，我们希望得到一组更小的新周期计数，使其比率尽可能接近原始周期计数的比率。成本归一化将这一思路表述为一个$\mathbb{Z}$线性规划问题（与模调度所用的问题相互独立）。

设整数列表C 为原始周期计数。我们希望推导出一个新的整数列表C'，使得对于所有i、j，$C[i]/C[j] \approx C'[i]/C'[j]$ 。通过引入一个变量F 来限制比率的变化，可以将其形式化为一个约束。我们还将C' 的总和限制在一定范围内：设置上限和下限，后者用于避免所有值均为零的退化解：

$$\forall i, j \quad -F \le C[i]\cdot C'[j] - C[j]\cdot C'[i] \le F$$

$$1 \le \sum_i C'[i] \le U$$

U 是用户定义的整数参数，用于控制成本分辨率与后续实际计算取模调度的算法运行时间之间的精确权衡。U 的值越小，分辨率越低（即比率变化越大），运行时间也越短。最后， $\mathbb{Z}$LP 的目标是最小化F。Twill 使用 SCIP 求解器 [23]来寻找解，因为在这类问题上，它的速度远快于 CBC 求解器 [21] 。在我们的实验中，我们选择$U = 300$ ，发现 SCIP 在所有情况下都能在 500 毫秒内找到全局最小值。在 Twill 的早期开发阶段，我们采用了一种临时方法：将每个周期计数除以一个固定整数，再通过四舍五入得到整数值。我们发现，即使对于同一程序在不同图形处理器上的情况，也很难保持这种方法的一致性。相比之下，将成本归一化表述为一个 $\mathbb{Z}$LP 问题，既提供了有原则的解决方案，也是发现最优调度所必需的。

5.3 可变延迟优化

如第 3.2 节所述，使用静态调度近似具有较大动态范围的可变延迟操作的执行时间，可能导致精度不足。虽然 Twill 会将可变延迟操作卸载到单独的线程束上，使其能够动态调度，但对于目标循环依赖图中没有传入数据依赖的可变延迟操作，还可以进行进一步优化。我们将这些可变延迟操作称为 流式 操作。在单独的线程束上，流式操作可以先于主软件流水线运行，并在其结果被需要之前完成多次迭代。为此，我们在成本模型中将流式操作的延迟设为零，使延迟已知的静态依赖操作能够得到精确调度。随后，我们将这些流式操作的流水线深度作为参数公开，以供外部自动调优系统进行调优；在现有的 TC GPU 编程系统中，公开此类参数以进行动态调优是常见做法 [5, 15, 28, 34, 36, 38]。许多重要的计算受限内核都包含流式可变延迟操作（例如 TMA 加载输入块），因此有必要针对这一重要情形进行优化。

5.4 局限性与未来工作

目前，我们的 Twill 实现仅支持不含额外控制流的单层嵌套循环。借助软件流水线相关文献中的分层归约技术，可以解除这一限制 [25]，我们将其留作未来工作。虽然 Twill 能够针对目前提出的约束推导出最优流水线，但不会自动确定分块大小，因此必须由人工或更高层的自动调优系统来选择。正如我们将在第 6 节所看到的，Twill 联合问题的求解时间从几十秒到几分钟不等。快速

求解时间并非我们方法的目标；我们以求解时间为代价换取最优性保证。我们设想，像 Twill 这样的系统可作为开发者辅助工具，或作为部署前使用的离线编译工具，而不是在交互式开发过程中持续运行。

6 评估

我们对机器学习文献中的两个重要内核进行了全面评估，并展示了 Twill 能够从基本原理出发，自动为多个代际的 NVIDIA GPU 推导出 SWP 和 WS 策略。据我们所知，Twill 是唯一一个无需针对每一代 GPU 定制启发式方法就能做到这一点的系统。随后，我们实现了这些策略，并证明其性能与经过人工调优的实现相比具有竞争力。

6.1 方法

内核选择。 我们的评估重点是计算受限内核，这类内核需要同时利用流式多处理器上的张量核心和通用功能单元。因此，我们聚焦于融合多头注意力中至关重要的前向传播阶段和反向传播阶段。由于这些阶段对人工智能系统十分重要，研究人员投入了大量精力来开发 SWP 和 WS 策略 [19,33]。反向传播阶段所需的循环结构与前向传播阶段有显著差异，因此能够展现 Twill 在发现不同策略方面的灵活性。

评估平台。 我们在 NVIDIA Hopper 和 Blackwell GPU 上评估 Twill。我们使用一块英伟达 H100 SXM5 80 GB 和一块英伟达 B200 180 GB。所有实验均使用 CUDA 13.0。Twill 联合问题的求解时间是在英特尔至强铂金 8570 的单个核心上记录的。

实验设置。 Twill 对从 Triton 程序中提取的依赖图进行调度。最初，我们打算在 Triton 中使用 Twill，并让 Triton 根据 Twill 的 SWP 和 WS 策略生成代码。然而，要在现代图形处理器上实现高性能，还必须正确处理许多其他关键决策和优化（它们与 Twill 正交）。我们发现，Triton 在代码生成期间做出了许多不正确的决策，例如内存分配、数据布局转换或同步位置安排方面的决策。因此，Triton 要么无法成功编译 Twill 的流水线，要么生成的代码性能不佳。为了展示 Twill 所找到的调度的性能，我们转而将 Twill 的流水线“手动编译”为 CUDA C++。这一过程涉及将流水线化的

以及由 Twill 发射到 CUDA 中的、带线程束注解的中间表示。它实现了指定策略，同时使我们能够正确完成其余（且彼此正交的）降低步骤。

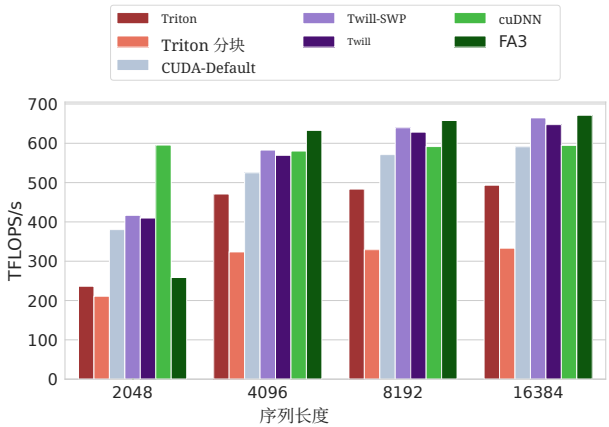


图 7：Hopper FP16非因果前向注意力。（批次=4，NUM_HEADS=32，HEAD_DIM=128）

要以最优方式自动化完成所有这些剩余步骤，需要大量工程投入和进一步研究，因此不属于本工作的范围。

6.2 注意力前向传播

在 Flash Attention（FA）算法 [17–19,33] 的推动下，融合多头注意力的前向传播阶段在多代 GPU 硬件上一直是重要的优化重点。我们聚焦于 Hopper 和 Blackwell 上的融合多头注意力，分别由 Flash Attention 3（FA3） [33] 和 Flash Attention 4（FA4） [19] 针对这两种架构。对于这两种架构，为实现峰值性能，都需要采用新颖的软件流水线和工作共享策略；这些策略是在相应架构发布数月后才由人类发现的。我们证明，Twill 能够在不同架构代际上自动重新发现相同的策略。

我们从 Triton 教程 [30] 中的 FA 实现开始，这是一个未经额外优化的高层级 FA 实现。Triton 程序描述了每个流式多处理器上的工作划分。为了让 Twill 采用联合的软件流水线与工作共享方法，我们按照目标硬件的张量核心指令尺寸对计算进行子分块。通过这种子分块，Twill 能够以适当的粒度控制调度和线程束分配；而 Triton 中间表示只会隐式地处理如何将块划分到多个线程束上。我们将深入讨论这些结果，并依次聚焦于每种架构。

6.2.1 Hopper

FA3 [33] 提出了两项在 Hopper 上优化 FMHA 的主要技术。第一项技术是针对主循环的软件流水线调度，将一个 GEMM 提取到循环的序言中，从而避免暴露指数运算的延迟（见图 1）。第二项技术称为乒乓调度，它会在执行一个子块中的 GEMM 时，调度另一个子块中的指数运算，从而再次提高张量核心利用率。我们将展示 Twill 如何自动推导出这两项优化。

图 7 展示了 Hopper 上前向 FMHA 不同实现的结果。“Triton”和“Triton- Tiled”两组柱状图分别表示 Triton 执行 FA 教程实现和子分块版本时的性能。如第 6.1 节所述，Triton 在编译过程中做出了许多次优决策。“CUDA-Default”柱状图展示了我们将 Triton IR 手动编译为 CUDA C 后的性能++，其性能较 Triton 有所提升。我们首先讨论 Twill 的结果“ T wil l-SWP”（ Twill-SWP）：它仅使用 Twill 的模调度（即只运行算法 最优-模-调度-策略 中的步骤 1）对主循环进行软件流水线处理，并采用 Triton 的 WS 策略，将全局内存加载安排在与其余计算不同的线程束上。这样带来了显著的性能提升，在序列长度为 16384 时，性能与官方 FA3 实现相差不到 1%。如果 Triton 这类系统采用模调度等算法，FA3 的这一组成部分就无需依靠人工发现。最后，“Twill”的结果采用了 Twill 的 SWP 与 WS 联合建模（即运行算法 1的全部内容）；Twill 在 28 秒内推导出一个解。Twill 的功能单元容量约束还原了 FA3 中所述的乒乓调度策略：由于同一时间只有一个操作可以使用张量核心，Twill 会错开各线程束组的执行，使一组发射指数运算，而另一组使用张量核心。Twill 发现了 FA3 所用的软件流水线，但只将其应用于一个线程束组，并判断这足以使功能单元达到饱和。尽管此版本的最终实现性能略低于 Twill-SWP，但两者最初的性能相当（约为 645 TFLOPS/s）；不过，一项正交优化（ TMA 多播）对采用模调度的实现更有利。在 Hopper 上，多个线程束会动态交错，将工作发射到张量核心及其他功能单元，从而自然获得指令级并行，并减轻对静态指令调度质量的依赖。接下来我们将看到，这在 Blackwell 上已不再成立。

6.2.2 Blackwell

由于 TC 速度更快，且需要更多同步操作（张量内存加载与存储），Blackwell 所需的软件流水线和工作共享策略与 Hopper 有很大不同。Blackwell 的结果集如图所示8。

与 Hopper 一样，我们也纳入了教程中的 Triton 结果以及子分块版本。我们将 Triton IR 直接转换为 CUDA 的实现，性能不如 Blackwell 上的 Triton，因为 Triton 的 Blackwell 后端会启发式地应用 FA3 SWP 策略。对于 Hopper，当我们应用模调度和 Triton 的 WS 策略时，性能可与 Triton 持平，但与最佳实现相比仍有很大差距。

Twill 对 SWP 和 WS 的联合优化在 19 秒内完成，并找到一种策略，其性能显著优于 Triton，且可与 cuDNN 和 FA4 相媲美。事实上，发现的策略与所提出的策略完全相同

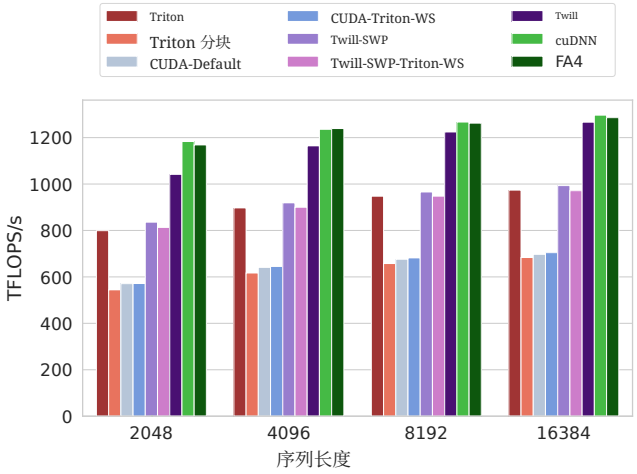


图 8：Blackwell 半精度浮点数非因果前向注意力。（批次=4, NUM_HEADS=32，HEAD_DIM=128）

由 FA4 [19]提出！Twill 的策略采用了不同于 Hopper 的软件流水线，并结合了特定的线程束分配和跨 warp 通信。图 9 展示了 Twill 的 WS 策略，其中使用 Twill 根据该程序的 Triton IR 得到的实际依赖图生成了可视化结果。Twill 将可变延迟操作（绿色）和张量核心通用矩阵乘法（粉色）分配到不同的线程束上；将每个子块的 softmax 计算分配到两组不同的线程束（蓝色和橙色）上；并将两个子块的累加器重新缩放操作分配到第三组线程束（黄色）上。

这种策略不符合“加载器”线程束和“计算”线程束等传统线程束分工。若不考虑相关的软件流水线，这种策略背后的原理并不明显。该流水线与 FA3 类似，会将每个子块的第一次通用矩阵乘法移出循环。在主循环中，流水线会在一个子块执行通用矩阵乘法时，为另一个子块调度指数运算（类似乒乓调度）；与此同时，该子块的累加器正在重新缩放。由于从张量内存读取累加器需要阻塞同步，因此重新缩放被移至第三组线程束；若由负责发出指数运算指令的线程束执行，就会扰乱流水线。然而，将重新缩放移至第三组线程束会产生跨 warp 通信，因为它需要 softmax warp 生成的数据——Twill 会围绕这一延迟和同步进行调度。在序列长度为 16384 时，我们对这一调度的实现与 FA4 的性能差距不到 2%。

现在回到图 8 中的其他计算内核变体。Triton 采用启发式方法，通过将累加器重新缩放移至另一组线程束，复现 FA4 的策略。将这些启发式方法应用于我们完全采用软件流水线的实现（“Twill-SWP-Triton-WS”）或默认转换实现（“CUDA-Triton-WS”）均未带来任何收益，这表明联合考虑软件流水线和工作共享对于实现峰值性能至关重要。

Blackwell 结果中最后一个有力的组成部分，是一些修改，这些修改使 SMT 问题对于最小的

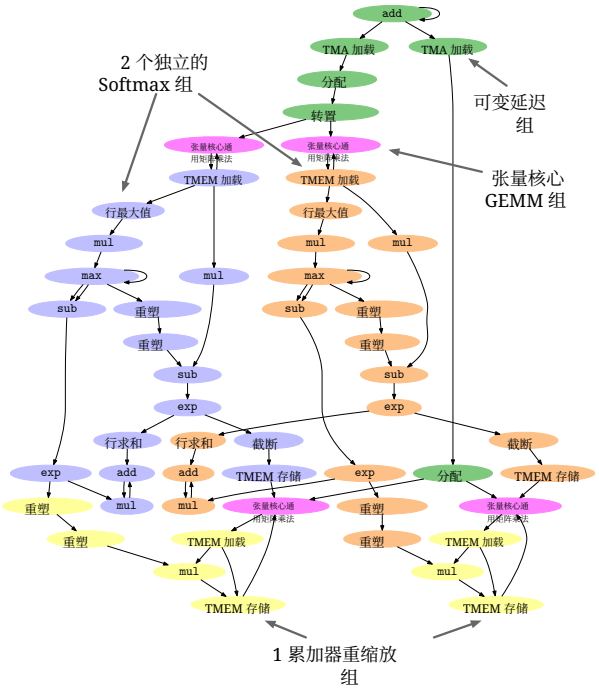


图 9：Twill 的依赖图，展示了为 Blackwell 发现的前向WS策略。颜色表示分配给同一组线程束的指令。

变得不可满足I，从而使 ℤLP 在算法1中成功I 和L，迫使搜索更大的取值。这些修改包括：

- 减少线程束数量。
- 直接使用教程中的 Triton 实现。没有子分块时，指令粒度不够细，无法进行交错。
- 禁止跨 warp 通信。

不可满足配置的多样性体现了寻找高性能软件流水线和工作共享策略的搜索空间之复杂性。Twill 提供了一个技术框架，可用于分析和探索这一复杂的搜索空间，同时保证获得最优结果。

6.3 注意力反向传播阶段

现在我们转向融合多头注意力的反向传播阶段。反向传播阶段与前向传播阶段有很大不同，需要采用不同的软件流水线和工作共享策略。我们没有使用反向传播阶段的 Triton 教程实现，因为它采用了低工作效率的两遍算法。相反，我们按照 FA 中所述，在 Triton 中实现了单遍反向算法3 [33]，从而可以直接与 cuDNN 和参考 FA 实现进行比较。该单遍算法包含五次通用矩阵乘法和一次指数运算，随后执行原子归约并写入全局内存。与前向传播阶段类似，我们也实现了该算法的子分块版本。

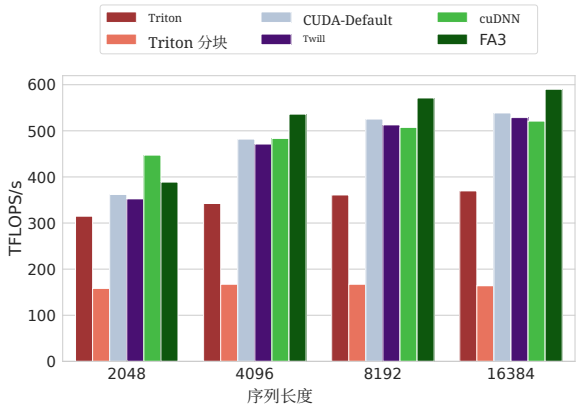


图10：Hopper 半精度浮点数非因果反向注意力结果。（批次=4，头数_，头维度_）

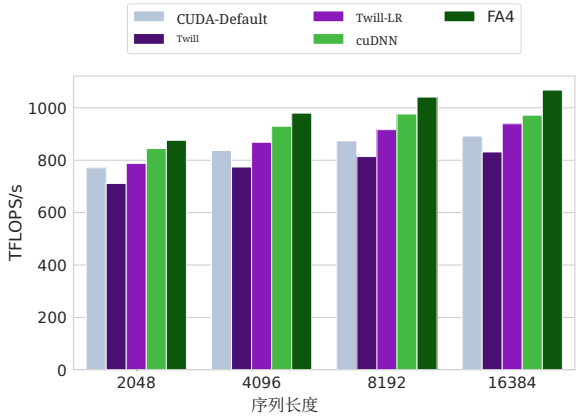


图11：Blackwell 半精度浮点数非因果反向注意力结果。（批次=4，头数_，头维度_）

6.3.1 Hopper

Hopper 的结果如图 10所示。与 Hopper 上的前向计算类似，我们将 Triton IR 移植到 CUDA 后，相较于 Triton 实现了性能提升，并且在最长序列长度下，其性能可与 cuDNN 竞争。Hopper 上的反向注意力计算受寄存器限制，因此 FA3 [33] 不会构建能够利用迭代边界之间 ILP 的软件流水线。Twill 搜索了 88 秒，并找到了类似的调度：它利用了循环迭代内的 ILP，但由于寄存器容量限制，无法跨迭代利用 ILP。Twill 的结果证实，FA3 的开发者并未因其软件流水线策略而错失潜在性能。我们认为，参考 FA3 实现通过使用稍大的分块大小（80x128），从而能够使用更大的通用矩阵乘法，在序列长度为 16384 时比我们的实现快 11%。Triton 仅支持二的幂次分块大小，因此我们无法构建具有相同分块大小的 IR，但没有根本理由表明 Twill 无法处理非二的幂次分块大小。

6.3.2 Blackwell

Blackwell 的结果如图 11所示。Triton 无法为 Blackwell 生成代码，因为目前无法构建包含别名的张量内存分配策略。Blackwell 上的 FA4 策略 [19] 使用三组线程束：其中两组从张量内存中读取累加器以执行指数运算，最后一组将累加器暂存到共享内存中，以便执行输出的原子归约。Twill 在 269 秒内找到了不同的策略，仅使用两组线程束，在执行指数运算和暂存归约数据之间交替进行（“Twill”）。尽管我们检查后确认该调度应能满足每线程寄存器预算，`ptxas` 仍无法在不导致主循环中大量寄存器溢出的情况下为这段代码分配寄存器，从而降低了性能。随后，我们使用降低后的每线程寄存器预算运行 Twill，并在 64 秒内找到了前文所述的相同策略，即使用三组线程束。`ptxas` 成功为这一版本分配了寄存器，使其相较于默认实现获得了小幅加速（“Twill-LR”）。Blackwell 上各反向实现之间的总体差异相对较小；Blackwell 张量核心的高吞吐量使单遍算法受带宽限制，其性能受限于每次迭代完成原子归约的速度。不过，Twill 找到了类似的软件流水线和工作共享策略，隐藏了直接移植 Hopper 反向策略时暴露出的部分延迟。我们认为，剩余的性能差异来自经过充分调优的参考实现中，与内存布局和指令选择相关的正交优化。

7 相关工作

7.1 TensorCoreGPU 编程

GPU 编程领域专业水平差异很大，而人们又希望开展多样化、高性能的机器学习计算，这两者结合催生了一个庞大的编程系统生态，用于面向张量核心 GPU。我们将综述这一系统领域。

在 PyTorch [31] 或 JAX [12], 等深度学习框架之下，Triton [36],Pallas [5] 和 CUDA Tile [15] 等领域专用语言（DSL）提供了用于操作数据瓦片的抽象。随后，各语言的编译器负责生成高效代码，通常能达到中等至较高的性能。TLX [26] 和 TileLang [37] 等略低层次的 DSL 也提供基于分块的编程模型，但能让用户更灵活地控制优化。Gluon [29] 是 Triton 扩展，它向程序员开放 Triton 的低层中间表示（IR）。Gluon 程序员可以借助进一步降低此中间表示的过程获得一定程度的自动化，同时仍能控制内存分配和工作共享（WS）等优化。Cypress [38] 是一种基于任务的编程模型，允许程序员表达计算和数据在图形处理器上的分解方式。Cypress 提供独立的映射接口，用于控制对性能敏感的决策，同时不影响正确性。最后，ThunderKittens [34] 和 CUTLASS [28]等系统让开发者拥有完全的控制权，不将任何对性能敏感的决策交由自动化系统处理。

7.2 软件流水线

面向 CPU 的流水线。 软件流水线是 CPU 的一项经典且成熟的优化，对于提取足够的指令级并行性以使机器达到饱和至关重要。即使对于现代乱序处理器，软件流水线也可以通过用相互独立的指令填充处理器的重排序缓冲区来提高指令吞吐量。基于压缩的 [7] 流水线算法会在逻辑上展开目标循环，直到找到一个固定点流水线。模调度 [25, 32] 流水线算法使用模表数据结构来建模稳态资源使用。借助整数线性规划，模调度算法可以推导出最优解 [22, 35]。Twill 展示了如何将传统的模调度技术应用于张量核心 GPU。

面向张量核心 GPU 的流水线。ALCOP [24] 对加载和计算执行软件流水线，因此不足以推导出能够重叠计算操作的流水线，例如 FA3。PipeThreader [14]通过定义程序空间并在其中搜索来寻找软件流水线策略，同时对每个候选方案进行性能分析，以找出速度最快的方案。相比之下，Twill 采用基于约束的方法，程序空间由约束隐式定义，并通过高度优化的求解器执行搜索。此外，Twill 不依赖性能分析——我们的方法是静态的，而是依靠公开可用的规格来估算操作延迟。此外，PipeThreader 采用传统的启发式线程束专门化方法，即创建生产者和消费者线程束；而 Twill 则从基本原理出发推导线程束专门化策略。最重要的是，PipeThreader 无法为其找到的调度提供任何最优性保证。

7.3 线程束专门化

线程束专门化（WS）这一 GPU 编程技术是在英伟达 GPU 配备张量核心之前提出的。CUDA-DMA [10] 利用线程束专门化，将全局内存和共享内存之间的数据移动与计算分离，从而提高内存带宽。Singe [11] 将燃烧化学计算分配到不同线程束上，以缩小每个线程束的寄存器级工作集，使得无需发生溢出即可模拟大型化学反应。

自 Hopper 起，工作共享（WS）已在高性能张量核心（TC）计算内核中无处不在。面向张量核心 GPU 的编程系统提供了围绕工作共享的多种控制层级。高级

系统，例如 Triton [36],Cypress [38] 或 Tawa [13]，会通过启发式方法定义工作共享策略，然后自动转换源程序以实现所选策略。中级系统，例如 TLX [26] 允许通过源代码级注释指定工作共享策略，然后通过编译器内部的转换算法实现该策略。最后，低级系统，例如 Gluon [29], ThunderKittens [34]或 CUDA C++，要求用户既定义工作共享策略，又通过低级代码实现该策略。Twill 专注于定义工作共享策略这一问题；现有的其他系统都通过机器特定启发式方法或依赖程序员直觉来完成这项工作。

8 结论

我们介绍了 Twill，这是一个能够为张量核心 GPU 探索最优软件流水线和工作共享策略的系统。Twill 提出了软件流水线与工作共享的新型联合表述，可交由 $\mathbb{Z}$LP 和 SMT 求解器处理，从而推导出专家们针对多代 GPU 手动发现的复杂策略。我们还表明，分别考虑这些优化策略的方法无法达到峰值性能。

致谢

我们感谢 Duane Merrill 与我们讨论 GPU 架构，并启发我们对线程束专门化的看法。我们感谢 Vinod Grover 与我们讨论线程束专门化和软件流水线。我们感谢 Evghenii Gaburov、Masahiro Masuda 和 Jason Knight 在 Triton 开发及 Triton 线程束专门化基础设施方面给予的支持。我们感谢 Pradeep Ramani 和 Cameron Shinn 帮助我们对 Hopper 和 Blackwell 计算内核进行性能调试。我们感谢 Benjamin Driscoll 提议使用缩写 $\mathbb{Z}$LP。我们还感谢（排名不分先后）Chris Gyurgyik、Katherine Mohr、AJ Root、Shiv Sundram、Atharva Chougule、Sai Gautham Ravipati 和 Konstantinos Kallas 对本文草稿提出反馈。

参考文献

[1] CUDA C++ 最佳实践指南：CUDA C++ 最佳实践指南 13.0 文档 — docs.nvidia.com。https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html。 [访问日期：11-26-2025]。

[2] 深入了解 NVIDIA Blackwell Ultra：为 AI工厂时代提供动力的芯片。https://developer.nvidia.com/blog/inside-nvidia-blackwell-ultra-the-chip-powering-the-ai-factory-era/。 [访问日期：12-03-2025]。

[3]NVIDIA H100 Tensor Core GPU架构。https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c。 [访问日期：12-03-2025]。

[4]NVIDIA Tesla V100 GPU架构。https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf。 [访问日期：12-03-2025]。

[5] Pallas：一种 JAX 计算内核语言；JAX 文档 — docs.jax.dev。https://docs.jax.dev/en/latest/pallas/index.html。 [访问日期：11-26-2025]。

[6] Alfred V. Aho、Monica S. Lam、Ravi Sethi 和 Jef- frey D. Ullman。《编译器：原理、技术与工具》（第 2 版）。Addison-Wesley Longman Publishing Co., Inc.，美国，2006 年。[7] A. Aiken 和 A. Nicolau。最优循环并行化。*SIGPLAN* 通讯，23(7):308–317，1988 年 6 月。[8] Andrew W. Appel。《现代编译器实现：ML 版》。剑桥大学出版社，美国，第 1 版，1998 年。[9] Clark Barrett 和 Cesare Tinelli。《可满足性模理论》，第 305–343 页。Springer International Publishing，Cham，2018 年。[10] Michael Bauer、Henry Cook 和 Brucek Khailany。CudaDMA：通过线程束专门化优化图形处理器内存带宽。载于*2011* 年国际高性能计算、网络、存储与分析大会论文集，SC '11，美国纽约州纽约市，2011 年。美国计算机协会。[11] Michael Bauer、Sean Treichler 和 Alex Aiken。Singe：利用线程束专门化提升图形处理器的性能。载于第 *19* 届 *ACM SIGPLAN* 并行编程原理与实践研讨会论文集，PPoPP '14，第 119–130 页，美国纽约州纽约市，2014 年。美国计算机协会。[12]James Bradbury、Roy Frostig、Peter Hawkins、Matthew James Johnson、Chris Leary、Dougal Maclau- rin、George Necula、Adam Paszke、Jake VanderPlas、Skye Wanderman-Milne 和 Qiao Zhang。JAX：Python+NumPy 程序的可组合变换，2018 年。[13] Hongzheng Chen、Bin Fan、Alexander Collins、Bas- tian Hagedorn、Evghenii Gaburov、Masahiro Masuda、Matthew Brookhart、Chris Sullivan、Jason Knight、Zhiru Zhang 和 Vinod Grover。Tawa：面向现代图形处理器、支持异步引用的自动线程束专门化。*arXiv* 预印本 *arXiv:2510.14719*，2025 年。

[14] Yu Cheng、Lei Wang、Yining Shi、Yuqing Xia、Lingxiao Ma、Jilong Xue、Yang Wang、Zhiwen Mo、Feiyang Chen、Fan Yang、Mao Yang 和 Zhi Yang。PipeThreader：面向高效深度神经网络执行的软件定义流水线。载于第 *19* 届 *USENIX* 操作系统设计与实现大会论文集，OSDI '25，美国，2025 年。USENIX 协会。

[15] 英伟达公司。NVIDIA CUDA Tile。https://developer.nvidia.com/cuda/tile，2025。 [访问日期：10-12-2025]。

[16] R. Cytron、J. Ferrante、B. K. Rosen、M. N. Wegman 和 F. K. Zadeck。计算静态单赋值形式的一种高效方法。载于第*16*届*ACM SIGPLAN-SIGACT*编程语言原理研讨会论文集，POPL '89，第25–35页，纽约，美国，1989年。美国计算机协会。[17] Tri Dao。FlashAttention-2：借助更好的并行性和工作划分实现更快的注意力计算。*arXiv*预印本 *arXiv:2307.08691*，2023年。[18] Tri Dao、Daniel Y. Fu、Stefano Ermon、Atri Rudra 和 Christopher Ré。FlashAttention：具备I/O感知能力的快速、节省内存的精确注意力计算。载于第*36*届神经信息处理系统国际会议论文集，NIPS '22，雷德胡克，美国，2022年。Curran Associates Inc.[19] Tri Dao、Jay Shah、Ted Zadouri、Markus Hoehnerbach 和 Vijay Thakkar。Flash Attention 4。
h
t
t
p
s://
g
ithub.com/Dao-AILab/flash-attention/tree/main/flash_attn/cute，2025年。[访问日期：02-12-2025]。[20] Bruno Dutertre。Yices 2.2。载于 Armin Biere 和 Roderick Bloem 编，计算机辅助验证，第737–744页，Cham，2014 年。Springer International Publishing。[21] John Forrest 和 Robin Lougee-Heimer。CBC用户指南。载于新兴理论、方法与应用，第257–277页。INFORMS，2005年9月。[22] R. Govindarajan、E.R. Altman 和 G.R. Gao。在资源受限的速率最优软件流水线中，最大限度地降低寄存器需求。载于*MICRO-27*论文集：第*27*届*IEEE/ACM*微体系结构国际研讨会，第85–94页，1994年。[23]Christopher Hojny、Mathieu Besançon、Ksenia Bestuzheva、Sander Borst、Antonia Chmiela、João Dionísio、Leon Eifler、Mohammed Ghannam、Ambros Gleixner、Adrian Göß、Alexander Hoen、Rolf van der Hulst、Dominik Kamp、Thorsten Koch、Kevin Kofler、Jurgen Lentz、Stephen J. Maher、Gioni Mexi、Erik Mühmer, Marc E. Pfetsch, Sebastian Pokutta, Felipe Serrano, Yuji Shinano, Mark Turner, Stefan Vigerske, Matthias Walter, Dieter Weninger 和 Liding Xu。《SCIP 优化套件 10.0》。技术报告，Optimization Online，2025 年 11 月。[24] Guyue Huang、Yang Bai、Liu Liu、Yuke Wang、Bei Yu、Yufei Ding 和 Yuan Xie。《ALCOP：面向 AI 图形处理器的深度学习编译器中的自动负载-计算流水线》。载于 D. Song、M. Carbin 和 T. Chen 编，《机器学习与系统论文集》第 5 卷，第 680–694 页。Curan，2023 年。[25] M. Lam。《软件流水线：一种适用于超长指令字机器的有效调度技术》。载于《ACM SIGPLAN 1988 年编程语言设计与实现会议论文集》（PLDI '88），第 318–328 页，美国纽约，1988 年。美国计算机协会。[26]Meta。TLX 语言。https : / / github . com / facebookexperimental / triton / tree / tlx，2025 年。[访问日期：30-11-2025]。[27]英伟达。CUTLASS Blackwell 前向注意力主循环。https : / / github / NVIDIA / cutlass / blob / a2439551c765c5393aebe557ee75d3a0412d2211 / examples / 77_blackwell_fmha / collective / sm100_fmha_fwd_mainloop_tma_warpspecialized. hpp。 [访问日期：20-11-2025]。[28] NVIDIA 公司。CUTLASS：CUDA 线性代数子程序模板。https : / / docs . nvidia.com/cutlass/，2025 年。访问日期：2025 年 11 月 29 日。[29]OpenAI。Gluon 语言。https : / / github . com / triton - lang / triton / blob / main / python / examples/gluon/01-attention-forward.py，2025 年。[访问日期：30-11-2025]。[30] OpenAI。Triton 教程中的注意力实现。https://github.com/triton-lang/triton/blob/fc9e3a637cd883a411a77459e15c2466aa9d3b7c / python / tutorials / 06 - fused - attention . py，2025 年。[访问日期：30-11-2025]。[31] Adam Paszke、Sam Gross、Francisco Massa、Adam Lerer、James Bradbury、Gregory Chanan、Trevor Killeen、Zeming Lin、Natalia Gimelshein、Luca Antiga、Alban Desmaison、Andreas Köpf、Edward Yang、Zach DeVito、Martin Raison、Alykhan Tejani、Sasank Chil- amkurthy、Benoit Steiner、Lu Fang、Junjie Bai 和 Soumith Chintala。《PyTorch：一种命令式风格的高性能深度学习库》。arXiv 预印本 arXiv:1912.01703，2019 年。

[32] B. Ramakrishna Rau。《迭代模调度：一种用于循环软件流水线的算法》。载于《第*27*届年度国际微体系结构研讨会论文集》，MICRO 27，第 63–74 页，美国纽约，1994 年。美国计算机协会。

[33]Jay Shah、Ganesh Bikshandi、Ying Zhang、Vijay Thakkar、Pradeep Ramani 和 Tri Dao。《FlashAttention-3：借助异步和低精度实现快速、精确的注意力》，2024年。

[34] Benjamin F. Spector、Simran Arora、Aaryan Singhal、Daniel Y. Fu 和 Christopher Ré。《ThunderKittens：简单、快速又可爱的人工智能内核》。*arXiv*预印本 *arXiv:2410.20399*，2024年。

[35] Artour Stoutchinin。面向 MIPS R8000 处理器的软件流水线整数线性规划模型。载于 Victor Malyshkin 编，《并行计算技术》，第 121–135 页，柏林、海德堡，1997年。施普林格·柏林·海德堡。

[36] Philippe Tillet、H. T. Kung 和 David Cox。Triton：用于分块神经网络计算的中间语言和编译器。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，第 10–19 页，纽约，美国纽约州，2019年。美国计算机协会。

[37] Lei Wang、Yu Cheng、Yining Shi、Zhengju Tang、Zhiwen Mo、Wenhao Xie、Lingxiao Ma、Yuqing Xia、Jilong Xue、Fan Yang 和 Zhi Yang。《TileLang：面向人工智能系统的可组合分块编程模型》，2025年。

[38] Rohan Yadav、Michael Garland、Alex Aiken 和 Michael Bauer。《现代图形处理器上的基于任务的张量计算》。《美国计算机协会程序设计语言汇刊》，9(PLDI)，2025年6月。