arXiv:2602.19762v1 [cs.PL] 23 Feb 2026

Hexagon-MLIR：面向高通神经处理单元（NPU）的AI编译栈

Mohammed Javed Absar[1],Muthu Baskaran[2],Abhikrant Sharma[2],Abhilash Bhandari[3],Ankit Aggarwal[2], Arun Rangasamy[3], Dibyendu Das[3], FatemeHosseini[2], Franck Slama[2],Iulian Brumar[2], Jyotsna Verma[2], Krishnaprasad Bindumadhavan[3], Mitesh Kothari[2], Mohit Gupta[2], Ravishankar Kolachana[3], Richard Lethin[2], Samarth Narang[2], Sanjay Motilal Ladwa[3],Shalini Jain[3], Snigdha Suresh Dalvi[2], Tasmia Rahman[2], Venkat Rasagna Reddy Komatireddy[3],Vivek Vasudevbhai Pandya[3],Xiyue Shi[2],和 Zachary Zipper[2]

[1]高通技术国际有限公司[2]高通技术公司[3]高通印度私人有限公司

摘要

本文介绍 Hexagon-MLIR [17, 18],，这是一个面向高通 Hexagon 神经处理单元（NPU）的开源编译栈 [10]，可统一支持 Triton 内核 [24] 和 PyTorch 模型 [15]的降级。我们的编译器基于 MLIR 框架 [11]，构建，通过一系列结构化编译通道，充分利用 NPU 的架构特性来加速 AI 工作负载。它支持针对我们的目标平台，将 Triton 内核（手工编写或来自 PyTorch 2.0 [2] 的子图）自动编译为二进制文件，从而加快新内核的部署。通过接收 Triton 内核，我们生成大型内核，以最大限度地提高数据在 NPU 紧耦合内存（TCM）中的局部性，减少基于库的方案固有的带宽瓶颈。这项计划补充了我们的商业工具链，为开发者提供基于 MLIR 的开源编译栈，使他们能够通过更灵活的方式推进 AI 编译能力。Hexagon-MLIR 仍在开发中，我们将继续为此项目添加更多优化和功能。

1 引言

Hexagon-MLIR 是一个用于在高通 Hexagon 神经处理单元（NPU）[10]上编译并执行 Triton 内核 [24] 和 PyTorch [15] 计算图的技术栈。该技术栈的部分内容最近已开源[17, 18]。基于 MLIR 框架 [11]，构建，我们的编译器应用一系列结构化的转换、优化和降级遍，以利用神经处理单元架构特性——包括硬件向量扩展（HVX）；利用硬件线程和多个 HVX 上下文的多线程；包含紧耦合内存（TCM）的内存层次结构；DMA 传输与计算的交错；经过优化的向量化数学库；以及高吞吐矩阵乘法引擎——从而加速 AI 工作负载。Hexagon-MLIR 采用生成式方法：通过将融合视为一等编译通道，我们可以为任意长度的算子链生成专用内核。

传统的机器学习编译方法通常依赖算子库，它们能够提供峰值性能，但需要耗时且谨慎的手动实现 [4] [21]。随着新的算子和优化不断使用 Triton 及其他领域专用语言编写

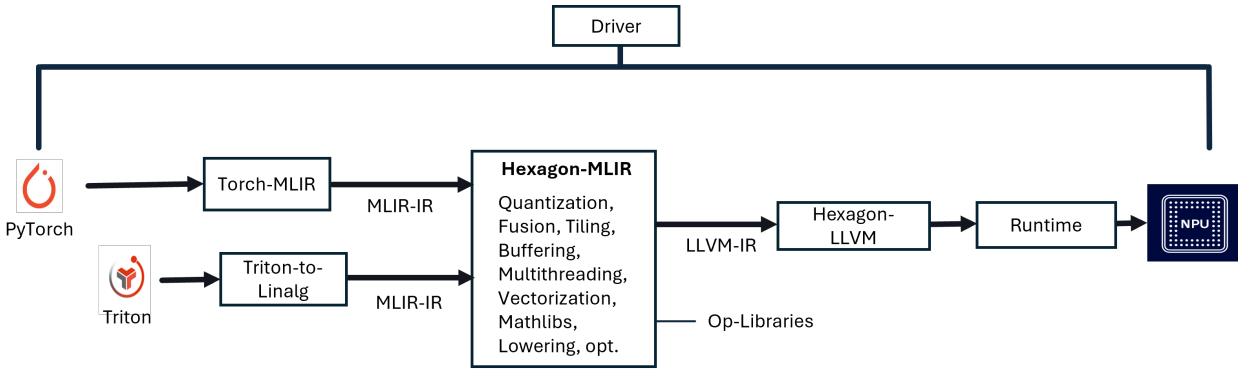

图 1：Hexagon-MLIR——AI编译栈

[19, 25] ，从 PyTorch算子、手写或由 Inductor 生成的 Triton 内核到经过合理优化的二进制文件，提供了一条持续可用的快速自动化路径，从而显著提升部署速度。此外，算子库 是阻碍优化的不透明屏障。这种碎片化迫使数据在调用之间往返于 DRAM，造成带宽瓶颈。与 20 世纪 90 年代能够利用局部性和并行性的区域编译 [7, 9], 类似，通过融合 [28]实现的 Triton 内核也提供了利用局部性和并行性的机会，而这些机会可能被算子库错过。手写汇编（峰值性能代码）在单项操作上速度更快，但自动代码生成才是适应 AI模型快速演进的唯一可扩展解决方案。此外，Triton生态系统持续发展，涌现出更高层次的抽象，例如 Helion [16], 。它提供以 PyTorch为中心的DSL，并将其编译为 Triton，进一步降低了内核开发的门槛。Hexagon-MLIR 提供了架构基础，可接收这些大型区域（例如通过 PyTorch Inductor [2, 15]），并将其编译为二进制文件。

高效编译新型大型语言模型（LLM）和近期的混合专家架构 [6] 仍是一个活跃的研究领域。近期研究表明，Triton 在注意力机制等关键 LLM 组件上能够实现最先进的性能 [5, 20, 26]。此外，研究人员提出了多项式组合激活函数（PolyCom）等新型激活函数，旨在增强 Transformer 的表达能力和动态特性 [31] ，这进一步凸显了对灵活编译框架的需求，以便高效支持不断涌现的架构创新。通过 PyTorch 2.0 Inductor 路径，可以自动生成 Triton 内核。或者，也可以使用 Triton语言快速、高效地编写新算子或算法。因此，手动优化的算子库路径无法扩展。

我们的编译器基于 MLIR [11]构建。MLIR 并非某个特定的编译器，而是一个可用于构建编译器的框架。IREE [27], 是使用 MLIR 构建的一种 AI 编译器，它通过硬件抽象层（HAL）提供可移植性。另一种基于 MLIR 的编译器是 TPP（Tensor Processing Primitives） [8], ，它面向英特尔和 AMD 处理器；据其作者称，“其性能可达到手工优化的等效程序的 90%。”基于编译器的方法也已应用于分布式训练优化，例如 DeepCompile [23], ，它利用剖析引导优化遍来改善分布式环境中的通信-计算重叠。尽管封装手工优化的库（例如英特尔采用 libxsmm 的方案）能够确保标准操作达到峰值性能，却无法应对现代 Transformer 中大量存在的长尾融合模式，而且还面临所需内核数量的组合爆炸问题。

2 组织结构

本文其余部分安排如下。第 3 节概述 Hexagon-MLIR 编译流程，并对语义保持和通道流水线架构等关键概念进行形式化定义。第 4 节详细介绍核心编译器遍的技术细节，包括 Triton 到 Linalg 的转换、`linalg.generic` 抽象、算子融合、内存层次结构分块、多线程、双缓冲、数学库集成以及其他优化。我们在全文中通过 softmax 和 GELU 等具有代表性的内核示例，说明代码如何在编译流水线中逐步转换。第 5 节展示实验结果并进行性能分析，说明各项优化（向量化、多线程、双缓冲）的效果及其相互作用，以及内核在计算受限和内存受限状态之间转换时的表现。第 6 节讨论深度学习编译器领域的相关工作和优化技术。最后，第 7 节总结本文的贡献并展望未来工作方向。

3 Hexagon-MLIR：概述

图 1 概述了编译流程。PyTorch 模型通过 Torch-MLIR [1] 转换为 Linalg 和其他 MLIR 核心操作。同样，Triton 内核通过 triton-to-linalg [13]转换为 MLIR。

领域。 令 $\mathcal{M}$ 为 PyTorch 模型的集合，$\mathcal{K}$为 Triton 内核的集合，$\mathcal{L}$ 为 MLIR Linalg 操作的集合。那么：

$$f : \mathcal{M} \longrightarrow \mathcal{L} \quad \text{(Torch-MLIR conversion)}$$
$$g : \mathcal{K} \longrightarrow \mathcal{L} \quad \text{(Triton-to-Linalg conversion)}$$

语义保持。 对于所有 $m \in \mathcal{M}$ 和$k \in \mathcal{K}$，降级到 Linalg 的路径应保持指称语义。

$$[\![m]\!] = [\![f(m)]\!] \qquad \text{and} \qquad [\![k]\!] = [\![g(k)]\!]$$

虽然某些与精度相关的选择（例如量化）会削弱严格的语义保持，但机器学习模型通常能够容忍此类变化。此外，如果 MLIR 图中出现自定义命名操作，我们会将其转换为语义等价的 MLIR 核心操作。

Hexagon-MLIR 通道流水线。 编译过程可以建模为中间表示（IR）之间的一系列态射，其中每个编译通道都将一种中间表示转换为另一种：

$$\begin{aligned} \mathrm{Obj}(\mathbf{IR}) &= \{\mathrm{IR}_\alpha \mid \alpha \text{ is an IR abstraction level}\} \\ \mathrm{Hom}(\mathbf{IR}) &= \{p : \mathrm{IR}_\alpha \mapsto \mathrm{IR}_\beta \mid p \in \text{pass-pipeline}\} \end{aligned}$$

编译通道至少可以分为两类：

- **转换：** 重写中间表示以简化代码（例如，公共子表达式消除、规范化）或提升性能。例如，向量化编译通道：

$$p : \mathrm{IR}_{scalar} \mapsto \mathrm{IR}_{vectorized}$$

- **降级：** 逐步降低抽象层次，使其接近目标。例如，将结构化控制流转换为非结构化控制流：

$$p : \mathrm{IR}_{\mathrm{scf}} \mapsto \mathrm{IR}_{\mathrm{cf}}$$

Hexagon-MLIR 通道流水线包含许多编译通道，但本文聚焦于其中几个主要通道

符号	说明
C	规范化，CSE，或常量传播
F	算子融合
L	布局变换与传播
M	多线程
p_i	一种优化或降级遍
T	内存层次结构分块
Q	量化
V	向量化及相关编译通道

Hexagon-MLIR 流水线是应用于输入中间表示的编译通道的组合：

$$\Pi = p_n \circ \cdots \circ T \circ F \circ \cdots \circ p_0 : \mathrm{IR}_0 \longrightarrow \mathrm{IR}_{\mathrm{final}}.$$

本文其余部分将详细介绍一些重要的编译通道。

4 Hexagon-MLIR：详细信息

本节将深入探讨其中一些编译通道，并通过几个示例来帮助说明。

4.1 Triton 到 Linalg

Softmax 函数是机器学习中广泛使用的函数。数学上，对于给定的 $\mathbf{z} = (z_1, \ldots, z_K)$，softmax 为：

$$\sigma(z_i) = \frac{\exp(z_i - \max_j z_j)}{\sum_{k=1}^{K} \exp(z_k - \max_j z_j)}.$$

减去 $\max_j z_j$ 可以提高数值稳定性，同时不会改变结果。

```
@triton . jit
    定义 softmax_内核（输出_指针，输入_指针，... 行_减
```

图 1：Triton Softmax 内核摘录

Triton 内核。 图 1 展示了 Triton 内核的一个片段，该内核实现了数值稳定的逐行 softmax。内核从每行的所有元素中减去该行的最大值，以防止求幂过程中发生溢出。然后，它计算`tl.exp`，并使用逐行求和`tl.sum`进行聚合，以构成归一化因子。最后，将每个指数值除以该归一化因子，得到 Softmax 概率。

清单1中的 Triton 内核经 Triton 编译器降级为 Triton-IR，然后由 Triton 到 Linalg 转换器转换为 Linalg。清单2展示了该中间表示，其中包含一系列`linalg.generic`。这里稍作说明，解释一下`linalg.generic`，因为它是编译流程的重要组成部分。

4.2 Linalg-Generic

`linalg.generic` 是 MLIR 的 Linalg（线性代数）方言中的关键算子。作为一种操作，它包含若干构造：索引映射、迭代器类型、输入和输出，以及 Linalg 主体。

索引映射： 设 $\mathcal{I} \subseteq \mathbb{Z}^d$ 表示迭代域，它是受仿射约束限制的整数格。`linalg.generic` 操作指定了一组有限的仿射映射

$$\phi_\ell : \mathcal{I} \longrightarrow \mathbb{Z}^{n_\ell},$$

其中每个映射都定义了输入或输出张量 X_ℓ坐标空间中的一个视图。这些映射共同决定了如何将公共迭代域中的点投影到相应操作数的索引集上。

迭代器类型： 标记为并行 或归约的迭代域，编码了通用操作主体中沿各个轴进行计算的方式。linalg 主体必须与此声明一致。

载荷区域： `linalg.body`是一个基本块，对标量进行操作，并根据标量输入生成标量结果（输出）。该主体定义了在该`linalg.generic`迭代空间中的单个位置要执行的计算。每个标量都通过索引映射从张量操作数中提取，并作为块参数传入`linalg.body`。

`linalg.generic` 是一种结构化操作，它将迭代的几何信息（索引集、仿射映射和迭代器类型）封装起来，同时将计算委托给 linalg 主体。这种设计将数据遍历的控制与数据上的计算语义分离开来，从而能够在降级为显式循环之前进行有原则的转换。例如，每个命名操作（如 `linalg.add`、`linalg.conv_2d`）都会唯一地降级为 `linalg.generic`。这正是它实用性的关键，确保了面对机器学习领域大量操作时，能够采用统一的转换、优化和降级机制。虽然 linalg.generic 可以对缓冲区进行操作，但最佳实践是使用 基于张量的 linalg,

```
%9 =Linalg.通用
        {indexing_映射    = [仿射_映射 <(d0)     -> (d0)>,    仿射_映射 <(d0)     -> (d0)>,
         仿射_映射 <(d0)     -> (d0)>],
         迭代器_类型        = ["并行"]}
        输入(%3,  %8 : 张量<16384xf32>,      张量<16384xf32>)
        输出(%4   : 张量<16384xf32>) {
    ^bb0(%输入: 32位浮点数,    %输入_5: 32位浮点数,    %输出: 32位浮点数):
        %17 = 算术.subf %输入 , %in_5 : f32
        Linalg.生成    %17 : f32
}->张量<16384xf32>
%10 =张量.空张量():       张量<16384xf32>
%11 = Linalg.通用
         {索引_映射         = [仿射_映射 <(d0)     -> (d0)>,    仿射_映射 <(d0)     -> (d0)>],
          迭代器_类型        = ["并行"]}
         输入(%9   : 张量<16384x32位浮点数32>)   输出(%10   : 张量<16384x32位浮点数32>) {
    ^bb0(%in: 32位浮点数,    %输出: 32位浮点数):
        %17 = math.exp   %位于: f32
        linalg.产出    %17 : f32
}->张量<16384xf32>
```

清单2：Softmax 片段的 Linalg中间表示。

我们正是如此，因为它能提供 SSA 表示的优势。

总的来说，必须指出，许多具有以下特征的操作——输出形状不一致、数据依赖流、非仿射映射等——无法用单个`linalg.generic`表示。重要的机器学习操作，例如`softmax`、前缀和、`TopK`等，需要配合其他控制流操作和`linalg.generic`才能表示。此外，还必须指出，Linalg 增加了一个中间抽象层，形式为`linalg.contraction`和`linalg.elementwise`。它们也会分别降级为`linalg.generic`。

4.3 算子融合

算子融合是一种保持语义的转换，它将多个结构化操作（例如`linalg.generic`）合并为一个操作，从而消除中间结果物化并改善数据局部性。

考虑以下两个操作：

$$X \xrightarrow{P} Y \xrightarrow{Q} Z,$$

其中 P和 Q 是迭代域分别为 $\mathcal{I}_P$和 $\mathcal{I}_Q$ 的`linalg.generic`操作。**生产者** P使用以下方式写入 Y 的元素：

$$\phi^P_{\text{out}} : \mathcal{I}_P \to \text{dom}(Y),$$

而**消费者** Q使用以下方式读取 Y 中的元素：

$$\phi^Q_{\text{in}} : \mathcal{I}_Q \to \text{dom}(Y).$$

一种融合方式是将这两个`linalg.generic`合并为一个，这样就不必将整个张量 Y写回，而是在融合后的 generic 操作体中读取 X 的标量值x，生成 Y 的标量值 y，并立即使用该值生成 Z 中的 z。因此，如果

$$P = \texttt{linalg.generic}\{\mathcal{I}_P,\ \{\phi^P_\ell\},\ \mathbf{y} \coloneqq f(\mathbf{x})\},$$
$$Q = \texttt{linalg.generic}\{\mathcal{I}_Q,\ \{\phi^Q_\ell\},\ \mathbf{z} \coloneqq g(\mathbf{y})\},$$

那么，融合后的操作为：

$$\texttt{linalg.generic}\{\mathcal{I},\ \{\tilde{\phi}_\ell\},\ \mathbf{z} \coloneqq g\big(f(x)\big)\}.$$

清单3展示了清单1中融合的`subf` 与`exp`通用操作。清单3中的示例相当简-

```
...
%7 =Linalg.通用
          {indexing_maps    = [仿射_映射 <(d0)     -> (d0)>,
                               仿射_映射 <(d0)     -> (d0)>],
           迭代器_类型        = ["并行"]}
      输入（%3   :  张量 <16384xf32>)      输出（%4    :  张量 <16384xf32>){
    ^bb0(%在:  32位浮点数,   % out:   32位浮点数):
        %11 = 算术.subf %in    ,  % 已提取      : f32
        %12 = 数学.exp   %11  :  f32
        Linalg.产生      %12  :  f32
    } ->张量 <16384xf32>
...
```

清单3：融合后 Softmax 片段的中间表示

单（恒等映射），仅作示意；但一般来说，融合必须考虑广播、转置以及重新计算。

4.4 面向内存层次结构的分块

分块会将通常位于速度较慢但容量较大的共享内存（DDR）中的大型张量划分为较小的分块，并将其复制到容量较小但速度较快的本地紧耦合内存（TCM）中。随后，诸如向量化 HVX 操作之类的计算会针对 TCM 中的数据进行。充分复用存入或在 TCM 中生成的数据，对于抵消 DDR 与 TCM 之间的数据传输延迟至关重要。上一节介绍的算子融合有助于提高向量寄存器级别的数据复用。保留 TCM 中的张量结果，可以促进融合算子之间的数据复用。

下面详细介绍 Hexagon-MLIR 中的分块机制。给定一个迭代域为 $\mathcal{I} \subseteq \mathbb{Z}^d$的`linalg.generic`、一个分块向量$\mathbf{t} = (t_1, \ldots, t_d)$以及一个可选的置换向量，分块会引入两个层级的迭代：外层循环由`scf::for` 或`scf::forall`构成，而循环体仍是一个`linalg.generic`，但现在它会对 TCM 中的一个分块进行操作。

为充分利用内存层次结构，Hexagon-MLIR 中的分块遍次引入了两种不同的内存空间：DDR 和 TCM。DDR 与 TCM 之间的数据传输通过 DMA 操作完成。分块应用于Linalg-on-Tensor 层面，此时遍次会使用带有内存空间属性的`bufferization.alloc_`张量插入显式的数据移动操作。这些属性使编译器能够在后续生成 DDR 与 TCM 之间的 DMA 传输。此外，分块遍次还会进行循环置换，将可向量化维度置于最内层，并为分块循环添加注释，以便后续进行软件流水线等优化。我们稍后会再讨论 softmax，现在先来看另一个有趣的内核——高斯误差线性单元 (GELU)，以探索分块、向量化和双缓冲。

$$\mathrm{GELU}(x) \approx 0.5x \left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)\right)\right)$$

融合与分块后的 GELU 内核如清单 4 所示。缓冲区化尚未发生，因此分块后的通用操作仍作用于张量。请注意 `scf.for` 循环：在每次迭代中，都会通过 `tensor.extract_slice` 计算较大输入张量的一个切片。随后，该切片会从默认内存空间复制到 TCM。`linalg.generic` 操作的主体对载入 TCM 的数据进行计算。该通用操作的主体目前虽然看起来像标量代码，但经过进一步变换后，它将成为多线程 HVX 操作。接下来，我们将对该分块后的通用操作进行多线程化。

4.5 多线程

```
# 映射  = 仿射_映射 <(d0) -> (d0)>
模块属性 {llvm. 目标平台_三元组  =                  " Hexagon "} {
函数 . 函数   @gelu_内核 (% arg0: 内存引用 <* xf32>{tt. 可整除性              = 16 : i32 },  ..) {
  % cst _2 =算术 . 常量 7.978850e -01              : f32
  . .
  %4 =结构化控制流 .循环 %arg8 =%c0 到 % c1048576 步长     % c262144
        迭代_参数 (% arg9 =%0)  ->(张量 <1048576xf32>) {
      % 提取_切片          =张量 . 提取_切片             %3[% 参数8] [262144] [1]
           : 张量 <1048576xf32> to      张量 <262144xf32>
      %5 =缓冲区化 . alloc_张量 ()
           复制 (% extracted_slice ) {memory_space      = 1 : i64 } :   张量 <262144xf32>
      . .
      %7 =Linalg . 通用
           { indexing_映射    = [#映射 , # 映射 ], 迭代器_类型       = [" 并行 "]}
           输入 (%5    : 张量 <262144xf32>)    输出 (%6    : 张量 <262144xf32>) {
      ^bb0 (% 在:   32位浮    % 输出:  32位浮
        . .
        %13 = 算术 .mulf %12  , % cst  : f32
        %14 = math . exp  %13 : f32
        %15 = 算术 .addf %14  , % cst _0 : f32
        %16 = 算术 . subf %cst  _0, %14 : f32
        %17 = 算术 . divf   %16, %15 : f32
        . .
        Linalg .产出 %20      : f32
      } -> 张量 <262144xf32>
      % 插入_切片          =张量 . 插入_切片           %7 到    % arg9[% arg8] [262144] [1]
           : 张量 <262144xf32> into      张量 <1048576xf32>
      结构化控制流. yield % inserted_slice :  张量 <1048576xf32>
    } {all_并行 ,          分块_通用 }
    . .
  }
}
```

清单 4：融合和分块后的 GELU

高通神经处理单元具备硬件多线程，以支持 Hexagon Vector eXtensions（HVX） 指令的并行执行。向量上下文是一组支持 HVX 指令独立执行的资源，包括向量寄存器文件和向量谓词文件。多个硬件线程可以并行运行，每个线程使用不同的向量上下文。在本节中，我们将展示上一节中已分块的 GELU 内核的多线程化。我们有两种多线程方案。一种使用`scf::for`起点，另一种使用 `linalg.generic`。我们将在此展示后者。我们的多线程

使用 MLIR 的 Async 方言，它提供了一种结构化的分叉-汇合中间表示：

- `async.execute` 会创建一个异步区域，并通过 `async.yield` 返回一个令牌（以及可选的值）。
- `async.await` 通过挂起当前协程来等待令牌/值。

•`async.create_组`，`async.add_至_组`，并且 `async.await_`所有通过多个令牌实现屏障。异步表示会降低为

```
1 ... 2 %12 =async . create_组 %11 : ! async . group 3 scf . for % arg9 =% c0_10至 % c262144_11步长 % c8_12{ 4 %
令
内
14
ac
24
用
1as
1
1
1
1
1
1
1
1
2
2
2
2
2
```

清单 5：创建 async-threads 后的 GELU

LLVM 协程（llvm.coro.id、llvm.coro.begin、llvm.coro.suspend、llvm.coro.end）与 MLIR 异步运行时协同工作。Hexagon-MLIR 利用 HVX 多线程和两阶段流水线：

1. 形成虚拟线程：使用 scf.forall对工作进行分区，并编码并行结构。2. 形成异步线程：将该结构降级为 MLIR 的 Async 方言。这种方法使并行语义保持声明式且可分析。

阶段 1： 分析 linalg.generic，并使用多面体大小启发式判断并行化是否有利。它对并行迭代器进行分块，以生成 scf.forall（虚拟线程）；当范围可以均匀整除时采用块分配，并通过块循环分配实现线程规模均衡。

阶段 2： 每个 scf.forall都会被重写为显式的异步分叉-汇合模式。首先，async.create_组(N) 为N个线程建立一个屏障。外围的 scf.for 随后遍历分块，并且每次迭代都会生成：

async.execute { /* forall 分块的主体 */ } → 令牌生成的令牌通过以下方式添加到组中：async.add_to_group(group, token)

最后，async.await_all(group)确保所有操作完成后，才会继续执行任何依赖工作。这种方法保留了原有的并行语义，同时为基于协程的代码生成做好准备。

清单 5 展示了经过多线程处理后的 GELU 内核中间表示。其输入中间表示是清单 4 中的分块 GELU 内核。

4.6 双缓冲

软件流水线中的双缓冲是一种将计算与数据传输重叠起来的技术，从而减轻分层内存系统中的内存延迟。该方法利用两个缓冲区（ping 和 pong）：一个缓冲区正在执行计算时，另一个缓冲区同时填充数据（随后二者交换角色），从而使数据传输和计算能够并行进行。

在 AI/ML 工作负载中，张量运算占主导地位，且数据规模往往超过高速本地内存的容量，因此双缓冲至关重要。通信与计算的交错可将内存受限内核转变为能够容忍延迟的流水线，从而利用内存单元与计算单元之间的并行性。类似的重叠技术也已在分布式 AI 系统中得到探索，其中基于编译器的方法可为分布式工作负载实现原生的重叠优化 [29]。

hexaogn-mlir 中的实现通过两个分层编译通道引入双缓冲，将结构化软件流水线与异步数据移动分离。

第 1 阶段：结构变换第 1 阶段会查找处于“范式”中的规范单缓冲分块循环`scf.for`。这些循环带有属性，表明它们源自分块的 `linalg.generic`。分块会为每个分块留下调度三元组

内存引用子视图 → 内存引用分配 →内存引用.co py，

以及计算区域和回写序列。双缓冲编译通道会合成 ping-pong 流程：受保护序言将第一个分块预取到 ping 缓冲区中，并重建内核，其中包含两个互斥的子内核（标注为`db_ping_kernel`/`db_pong_kernel`）。每个子内核都会为下一次迭代发起预取，将计算克隆并把`alloc` 重新映射到当前的 ping/pong 缓冲区，然后在当前归纳变量处重新生成子视图，以重建存储操作。驻留在内存中的布尔值`toggle`控制 ping↔ pong 的选择，并在每次迭代时更新；而`db_generic`（唯一 ID）和`db_prologue`/`db_prefetch`等属性则为后续编译通道提供 IR 锚点。这样可以确保转换具有确定性且不会产生危险，同时保留原有的分块语义，并建立通信（预取）与计算之间的重叠，而无需预先采用特定的异步传输方式。

第 2 阶段：异步 DMA 集成第 2 阶段会具体生成真正的异步传输和同步操作。它使用注入的属性解析第 1 阶段 IR，以恢复序言、内核和 ping/pong 子调度。随后，该编译通道会将所有预加载和预取`memref.copy`操作重写为 `memref.dma_start`，并分配独立的 ping/pong 标签，以便分别等待每个阶段；同时，在相应的预取块之后立即插入`memref.dma_wait`，确保计算开始前数据已驻留。对于写回路径，它同样会将每个 `memref.copy`替换为`dma_start`，随后插入`dma_wait`（使用单独分配的存储标签），并在内核之后为所有标签生成配对的释放操作。这种设计将合法性与调度（第 1 阶段）同传输语义（第 2 阶段）解耦，通过显式的 DMA 启动/等待隐藏延迟，同时不扰动计算 IR。

清单 6 展示了双缓冲后的中间表示。中间表示中添加了 C++ 风格的注释，并为一些变量指定了自定义名称，以便于理解。

我们一直用 GELU 中间表示来说明整个流程，下面来回顾一下。清单3展示了较早阶段的内容，即融合之后。融合发生在Linalg-on-Tensor上。随后，我们在清单 4 中进行了分块，将切片从 DDR 移至 TCM。分块后的代码在张量与内存引用（缓冲区）之间搭起了桥梁。之后还会执行缓冲区化等其他编译通道。接着，我们在清单 5 的中间表示中展示了针对 HVX 的多线程。之后是本节展示的双缓冲。此后还会进行更多降级，最终生成带有运行时调用的 LLVM-IR。

4.7 数学库

在 AI/ML 工作负载中，使用低阶多项式近似替代`exp` 等超越函数，是一种常见做法，可以用可控的数值

误差换取更高的吞吐量。我们调用高通 Hexagon 库（QHL）——一组针对向量化计算预先优化的数学库。

MLIR 还提供数学运算的多项式逼近展开，用于不依赖任何自定义库函数的快速近似。一般方法例如要近似 e^x，首先将其表示为

$$e^x = e^{\epsilon+b} = e^{\epsilon+n\log 2} = e^{\epsilon}\cdot 2^n.$$

然后，由于将 ϵ 选取得较小，可以使用若干项泰勒级数来计算 e^{ϵ}。霍纳公式常用于减少计算该多项式所需的乘法与加法次数。

除此之外，还有其他用于生成数学运算快速实现的优化与算法。一个例子是**快速逆平方根**。

4.8 优化与降级

上一节讨论的流程使我们的工具能够成功编译 PyTorch 与 Triton 内核。我们并未涵盖所有已有或正在开发的编译通道与优化。我们也没有讨论所使用的 MLIR 核心编译通道，例如缓冲区化、释放以及降级到 LLVM，因为它们已相当为人熟知。我们在流程较早阶段识别卷积与矩阵乘法，通常它们会进入专用的矩阵乘法引擎。数据需要按特定方式布局，我们使用 `linalg.pack/unpack` 进行数据空间变换。为减少 pack/unpack 的数量，我们使用布局传播。

5 结果与分析

本节的结果旨在将前文介绍的编译器优化与随附表格和柱状图中展示的性能提升对应起来。我们在 Hexagon-MLIR 流水线中分别启用各个编译通道，也会将它们组合启用：融合（F）消除中间结果的物化并改善局部性；分块（T）将工作集移入 TCM；HVX 向量化（V）利用宽 SIMD 执行；多线程（M）将分块后的工作分配到多个硬件 HVX 上下文；双缓冲（DB）则使 DMA 传输与计算重叠。这种结构化的呈现方式使技术章节中介绍的每项优化都能与观测到的运行时效果对应起来，同时也突显了这些编译通道之间如何相互作用——有时相辅相成，有时彼此竞争——以及内核如何从计算受限逐渐转为内存受限。

5.1 向量化加速比

表 1 总结了若干内核通过 HVX 向量化获得的加速比。GELU 在 `float16`（16,384 个元素）下达到 63.9×，在 `float32` 下达到 16.1×，这表明 HVX 宽度利用率和数据类型打包是吞吐量的主要驱动因素。

RMS 归一化向量化后，在 127×513 形状下的加速比达到 46.5× ，证实以 stencil linalg.elementwise 为主的计算即使包含一些归约，也能从向量化中获益。回顾一下，RMS 归一化的定义如下：

$$\mathrm{RMSNorm}(x) = \frac{x}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}} \cdot g$$

其 Triton 内核中的计算部分如代码清单 7 所示。

```
平方 =沿求和(x * x，轴 =0) / 列数_列均方根
```

清单 7：Triton RMS Norm内核摘录

表 1 还列出了**Sigmoid 线性单元（SiLU）**激活函数的结果。SiLU 定义如下：

$$\mathrm{SiLU}(x) = x\cdot\sigma(x) = x\cdot\frac{1}{1+e^{-x}},$$

我们获得了不错的 $5-8$ 倍提升。计算主要受 e^x 项影响。对于 `float16`，当运算在内部被提升为 `float32`时，性能会下降。

表 1：向量化带来的加速比。

Test	加速比	备注
高斯误差线性单元（GELU）	63.9x	float16，元素 = 16384
高斯误差线性单元（GELU）	16.1x	float32，元素 = 16384
双缓冲 GELU	15.4x	float32，元素= 16x 16384，双缓冲
闪电注意力	4.7x	float32，N_CTX=1024，DIM_HEAD=64，BLOCK_N=64
均方根归一化（RMS-norm）	46.5 倍	float16，= 127 × 513 个元素，
Sigmoid 线性单元（SiLU）	4.8	float16，= 16384 个元素
Sigmoid 线性单元（SiLU）	7.1	float32，= 16384 个元素
二维向量加法	40.6 倍	float32，元素 = 64 x 16384

5.2 单线程与多线程

图2比较了随着问题规模逐渐增大（8K → 1M 个元素），单线程执行与跨 HVX 上下文的 HVX 多线程执行情况。在较小的问题规模（≤ 16K）下，由于线程启动、令牌管理和屏障开销，多线程运行时间更长。超过 32K 个元素后，多线程执行始终优于单线程执行，加速比从 2.28×（32K）到 3.95×（512K）不等，在 1M 个元素时达到 3.40×（见加速比柱状图）。当并行负载足以摊销`async.execute`和`async.await`全部的固定开销时，就能获得这些收益。在 1M 个元素时，加速比相比 512K 略有下降，反映出 TCM 和 DMA队列面临的压力不断增大。在这种情况下，重新调优可能会有所帮助。图3以更直观的形式展示了加速比，其定义为单线程执行时间与多线程执行时间的比值。

5.3 双缓冲

图5展示了向量化、多线程（MT）和双缓冲（DB）带来的性能提升。向量化带来了显著收益，但若将其视为既定优化（基准情况），那么多线程和双缓冲还能进一步提升性能。图7展示了双缓冲及其他优化为 GELU 带来的收益。同样，图6展示了计算密集型指数级数内核的性能提升。

5.4 多遍交互

现在我们来讨论多遍交互。向量化和多线程等技术主要用于缩短计算时间，这会使内存传输开销更加突出。双缓冲通过让数据移动与计算重叠来提升整体性能。不过，最终可能导致优化后的内核受内存限制。图4展示了一个理想化情况：当内存传输完全占主导地位（100%）时，双缓冲毫无收益；同样，如果计算完全占主导地位，这项技术也无效。在实践中，大多数工作负载介于这两个极端之间，因此双缓冲能够带来收益，而开销和调度等次要因素也会发挥重要作用。选择性地应用优化遍，有助于进一步了解内核特性以及这些次要因素之间的相互作用。

6 相关工作

Li等人 [12] 全面综述了深度学习（DL）编译器，分析其设计架构和优化策略。尽管这篇论文发表于五年前，但其中的概念至今仍然适用。论文阐述了 DL 编译器如何将来自各种框架的模型转换为适用于异构硬件的优化代码，重点介绍多级中间表示（IR），以及前端和后端优化。文中还讨论了算子融合、内存规划、布局变换、硬件映射和自动

调优。

Yifan 等人 [28], 提出了 Neptune，这是一种张量编译器，支持针对归约算子的循环融合算法；它通过重新计算来满足数据依赖，并自动推导所需的代数修正。其中，滚动更新融合和 Split-K 融合这两种方法尤其适用于优化类注意力算子。他们基于 Apache TVM 调度张量编译器和 Triton 分块张量编译器实现了 Neptune。同一作者较新的一篇论文 [30] 《深度学习编译器中的图优化策略》深入探讨了前端优化技术。作者认为，这些技术对于优化计算图结构至关重要，进而显著提升神经网络训练和推理阶段的效率。

Chin 等人在论文 [14] *HloEnv*：深度学习编译器优化研究的图重写环境中，从两个角度分析了某个优化遍或优化流水线对 HLO 数据集性能的影响：该优化遍影响的数据集比例，以及该优化遍带来的平均性能变化（启用与未启用该优化遍时的运行时比率）。

对于 AI 编译套件而言，优化计算速度、内存容量和传输需求至关重要。文献 [3]讨论了一种 XLA 编译器扩展，它可以根据用户指定的内存限制调整算法的计算数据流表示。Benoit 等人 [22]提出了 MODeL，这是一种整数线性规划方法，通过优化用于训练神经网络的张量的生命周期和内存位置，降低模型的内存使用量。

7 结论

本文介绍了 Hexagon-MLIR，这是一个基于 MLIR 构建的编译工具栈，专为高通 Hexagon NPU 上的 AI 工作负载而设计。通过结合 MLIR 的可扩展基础设施和硬件感知型编译通道，Hexagon-MLIR 为 PyTorch 模型和 Triton 内核提供了系统化的降级流水线。算子融合、分块、HVX 向量化、多线程执行和双缓冲等关键优化，可高效利用紧耦合内存（TCM）、异步 DMA 传输和专用数学库。实验结果表明，代表性内核的性能均有所提升，验证了我们方法的有效性。除性能之外，这项工作还展示了 MLIR 的设计原则——结构化中间表示、声明式转换和分阶段降级——如何为构建面向 AI 工作负载的编译器奠定坚实基础。尽管当前实现已取得可喜成果，我们仍在持续扩大对新兴模型的覆盖范围、完善调度启发式方法，并提升特性支持和性能。最终，Hexagon-MLIR 旨在为在 Hexagon NPU 上部署下一代 AI 工作负载提供可扩展的高性能解决方案。

参考文献

[1] Torch-MLIR：连接 PyTorch 与 MLIR 生态系统。GitHub 代码库，2021–2025。https://github.com/llvm/torch-mlir.[2] J. Ansel 等。PyTorch 2：通过动态 Python 字节码转换和图编译加速机器学习。载于第 *29* 届 *ACM* 编程语言与操作系统架构支持国际会议论文集，2024 年。[3] Artem Artemev、Yuze An、Tilman Roeder 和 Mark Van der Wilk。基于 XLA 编译器的内存安全计算，2022 年。[4]陈天奇、Thierry Moreau、Ziheng Jiang、Lianmin Zheng 等。TVM：一种用于深度学习的自动化端到端优化编译器，2018 年。[5] Tri Dao、Daniel Haziza、Francisco Massa 和 Grigory Sizov。面向长上下文推理的 Flash-decoding。
h
t
tps://crfm.stanford.edu/2023/10/12/flashdecoding.html，2023 年。

[6]DeepSeek-AI。DeepSeek-V3 技术报告，2024 年。模型检查点：https://github.com/deepseek-ai/DeepSeek-V3.[7] J. A. Fisher。轨迹调度：一种全局微码压缩技术。*IEEE* 计算机学报，1981 年。[8] Renato Golin、Lorenzo Chelini、Adam Siemieniuk、Kavitha Madhu、Niranjan Hasabnis、Hans Pabst、Evangelos Georganas 和 Alexander Heinecke。迈向基于上游 MLIR 的高性能 AI 编译器，2024 年。[9] W. M. Hwu 等。超块：一种高效的超长指令字和超标量编译技术。*The Journal of Supercomputing*，1993 年。[10]Qualcomm Technologies Inc. 借助 NPU 和异构计算释放设备端生成式 AI 的潜力。https://www.qualcomm.com/processors/ai-engine，2024 年。[11] Chris Lattner、Mehdi Amini、Uday Bondhugula、Albert Cohen、Andy Davis、Jacques Pienaar、River Riddle、Tatiana Shpeisman、Nicolas Vasilache 和 Oleksandr Zinenko。MLIR：面向摩尔定律终结的编译器基础设施，2020 年。[12]Mingzhen Li、Yi Liu、Xiaoyan Liu、Qingxiao Sun、Xin You、Hailong Yang、Zhongzhi Luan、Lin Gan、Guangwen Yang 和 Depei Qian。深度学习编译器：全面综述。*IEEE* 并行与分布式系统汇刊，32(3)，2021 年。可查阅：https://arxiv.org/abs/2002.03794.[13]Microsoft。triton-shared：用于 Triton 编译的共享中间层。GitHub 代码库，2019–2025。https://github.com/microsoft/triton-shared.[14] Chin Yang Oh、Kunhao Zheng、Bingyi Kang、Xinyi Wan、Zhongwen Xu、Shuicheng Yan、Min Lin 和 Yangzihao Wan。HloEnv：深度学习编译器优化研究的图重写环境。载于第 *36* 届神经信息处理系统大会（*NeurIPS 2022*）机器学习系统研讨会，2022 年。[15]Adam Paszke、Sam Gross、Francisco Massa、Adam Lerer 等。PyTorch：一种采用命令式风格的高性能深度学习库，2019。[16] PyTorch Team。Helion：用于 Triton 内核开发的高级领域特定语言。https://pytorch.org/blog/helion/，2025。[17]高通技术公司。使用开源 Hexagon-MLIR 为 Hexagon NPU 编译 Triton 和 PyTorch。https://www.qualcomm.com/developer/blog/2026/02/，2026年2月。[18]高通技术公司。Hexagon-MLIR：基于 MLIR 的 Hexagon NPU 开源编译栈。https://github.com/qualcomm/hexagon-mlir，2026。[19] J. Ragan-Kelley、C. Barnes、A. Adams、S. Paris、F. Durand 和 S. Amarasinghe。Halide：一种用于优化图像处理流水线中的并行性、局部性和重新计算的语言与编译器。*ACM SIGPLAN Notices*，48(6)，2013。可查阅：https://arxiv.org/pdf/1802.04730。[20]Burkhard Ringlein、Jan van Lunteren、Radu Stoica 和 Thomas Parnell。Triton 注意力内核剖析，2025。[21] Nadav Rotem 等。Glow：一种面向神经网络的图降级编译器技术，2018。[22] Benoit Steiner、Mostafa Elhoushi、Jacob Kahn 和 James Hegarty。MODeL：深度学习的内存优化。载于第 *40* 届 *International Conference on Machine Learning*（*ICML'23*）论文集，2023。[23] Masahiro Tanaka、Du Li、Umesh Chand、Ali Zafar、Haiying Shen 和 Olatunji Ruwase。Deep-compile：一种由编译器驱动的分布式深度学习训练优化方法，2025。[24]P. Tillet、H.-T. Kung 和 D. D. Cox。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（*MAPL@PLDI*）论文集，第 10–19 页，美国亚利桑那州菲尼克斯，2019。ACM。

[25] Nicolas Vasilache 等。张量综合：与框架无关的高性能机器学习抽象，2018。[26] Ashish Vaswani 等。Attention is All You Need。载于*Neural Information Processing Systems 30*（*NIPS 2017*）论文集，美国加利福尼亚州长滩，2017。[27]Hanhan Wang。IREE 中的数据分块：通过编译器设计实现高性能。LLVM 开发者大会，2025年6月。可查阅：
h
t
t
p
s://
l
l
v
m.
o
r
g/
d
evmtg/2025-06/slides/technical-talk/wang-data-tilling.pdf。[28] Yifan Zhao、Egan Johnson、Prasanth Chatarasi、Vikram Adve 和 Sasa Misailovic。Neptune：面向图形处理器局部性与并行性的高级机器学习算子融合，2024。[29] Size Zheng、Wenlei Bao、Xin Liu 等。Triton-distributed：基于编译器的分布式 AI 系统重叠优化，2025。项目页面：
https://github.com/ByteDance-Seed/Triton-distributed。[30] Yijian Zheng。深度学习编译器中的图优化策略。载于*2024*年国际交互智能系统与技术会议（*IIST*），2024。[31] Zhijian Zhuo、Ya Wang、Yutao Zeng、Xiaoqing Li、Xun Zhou 和 Jinwen Ma。多项式复合激活函数：释放大语言模型的动态特性，2024。北京大学数学科学学院。

```
1 // -----// 中间表示转储：Hexagon - 双缓冲 - 通用 -s2 // ---------- // 2 func . func @gelu_内核 (% arg0 : 内
```


代码清单 6：双缓冲后的 GELU

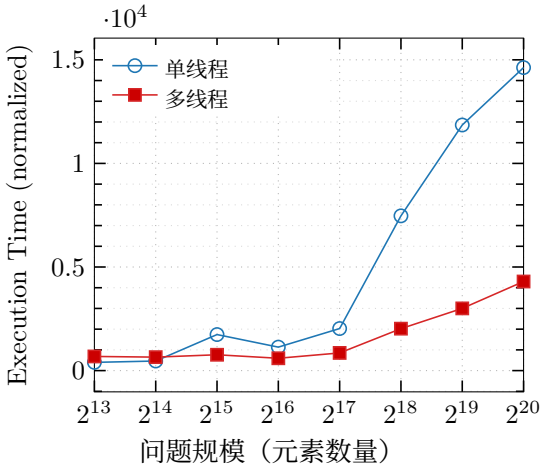

图2：单线程与多线程 GELU 内核实现的执行时间。

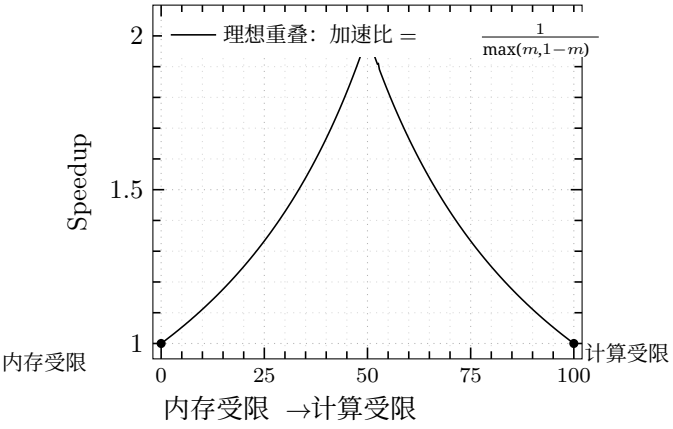

图4：随着内存占比变化，双缓冲在完全重叠时的加速比。

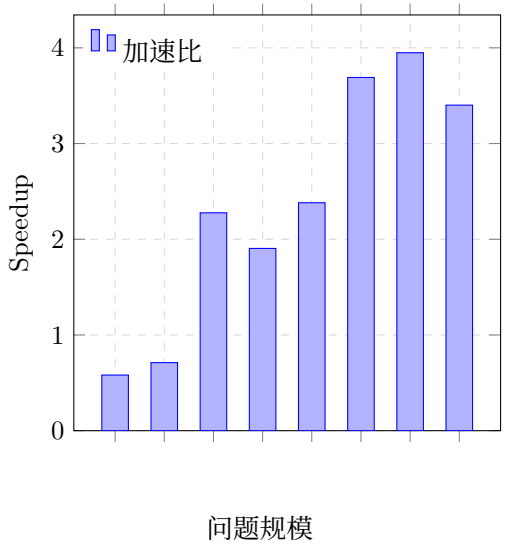

图3：随着问题规模不断增大，多线程 GELU 的加速比。

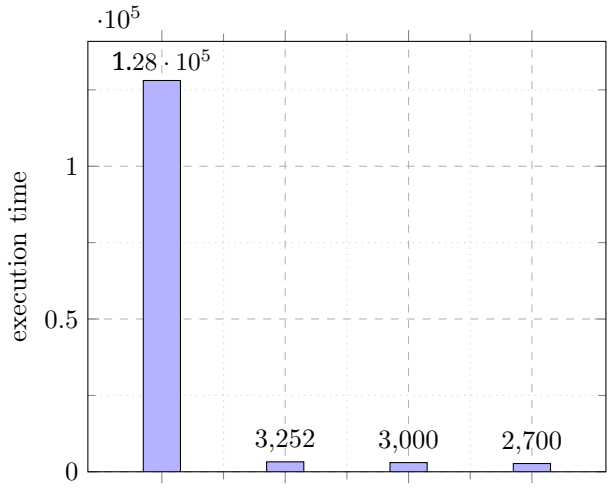

图5：二维向量加法在各优化编译通道中的性能。

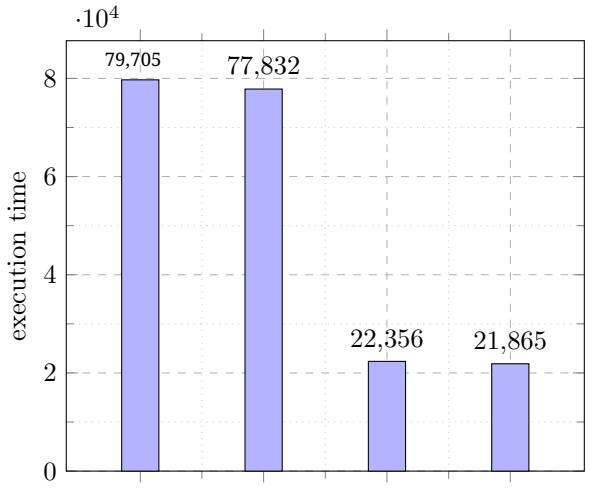

图 6：指数级数的多遍分析。

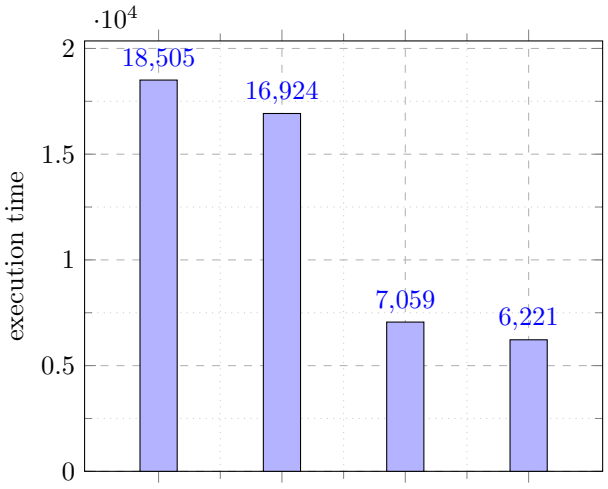

图 7：GELU 的多遍分析。