Astitch（缝纫）: Enabling a New Multi-dimensional Optimization Space for Memory-Intensive ML Training and Inference on Modern SIMT Architectures

Asplos 2022

Session 4A - Systems for Machine Learning

第一遍梳理：逐段梳理流水账

第二遍梳理：加入辅助讲解的文字和我的思考

写在最前面：我为什么要大花力气精读顶会论文?

- **本报告形式（字多、细节太多）不适用于给老板进行工作汇报、学位答辩、项目申请、大同行交流；仅适用于内部论文分享**
- Asplos的论文到底好在哪里?
  - 学习写作的逻辑性：如何建立几个关键词之间的逻辑性
  - 看别人的科研过程，思考自己应该如何思考问题：问题与问题之间的呈上启下➔研究有深度，没有只停留在表面问题上
  - 学习如何讲故事：用能够吸引读者眼球的地方（关键句），给出新概念（新概念要尽量少，但是如果没有就不好讲你的故事）
  - 学会举例子：简单、清晰、明确地给出case study➔说明关键的新概念、形象化地表示你方法之所以有效的原因或原理、形象地表明出现性能问的原因
  - 学习应该八股的地方：套路化的实验设计➔有限的测试和实验环境，呈现出更多实验➔不是仅仅体现单点技术在特定场景的性能优势，而是证明你工作整体能达到的能力（不同硬件间的可迁移性、不同负载间的适用性）
  - 你第一次读论文时，脑子里出现的疑问要记录下来。（为什么会有这样的技术选型）
  - 学习“大师级”的架构师思维：技术选型、tradeoff、case study、验证手段

两个读和分享论文的两个误区

- **如果你看不出一篇顶会到底哪里好**，说明。。。。。。（说明，要么您是已经是武林高手；要么，你的武功还不行，跟这个作者的差距可能不只是一个数量级）
- **如果你看完一篇顶级文章（无论精读还是粗读）后就完了**，对你手头的工作完全没有帮助，无法引起你对现有设计或实验的思考，没有学到任何套路，说明。。。。。。你读的不是跟你个人利益相关的文章，论文分享仅仅是混了个热闹

梳理全文流水账关键句

- **论文细节完整的整理，就是要像论文作者写论文过程一样，详细梳理全文细节**
- 先按照二级标题（如Sec 2.2）为PPT每一页
- 要点要以**每一段文字为单位**梳理关键句
  - 什么是关键句：起到承上启下的句子、引入新概念的句子、总结性句子、整段文字代表性句子、任何存疑问的句子
  - 注意尽量把与关键句在这一段里具有相同逻辑的句子去掉
  - 关键词➔这些词会吸引读者的注意
    - insight
    - observation
    - However
    - Key *Inefficiency*
    - Challenge
- 作者在论文中画的图都是重点，需要详细讲解其中的细节
- 在梳理流水账的时候，把与前文内容呼应的细节找出来

全文结构说明与注意事项

- INTRODUCTION 往往都有“由大到小”、“由远及近”的逻辑线，通过这种方式，将“大”的技术趋势与“小”的技术瓶颈和技术缺陷建立逻辑关系。最后看完能让人有“以小见大”的感觉。
  - 大的趋势和问题严重性很容易写的虚，需要有量化统计的论据或者前期验证性实验进行佐证。 (fig.1)
  - 小的技术点很容易写的十分零碎，难以建立跟大问题逻辑关联，这时候需要有能够体现技术点的形象化图示进行说明。(fig.2)
- BACKGROUND AND CURRENT CHALLENGES
  第二章承接第一章，要把技术背景与趋势➔关键核心概念➔难点问题➔性能挑战点，梳理出来。
  - 第二章写作的难点就是建立这四者的关联，应该先独立写出这四点。但是把这四部分很独立地写出来，容易写的比较晦涩。
  - 先写出这四点➔通过过渡句建立四点联系➔通过举具体例子的方式说明这四点，或者说明联系。

- KEY DESIGN METHODOLOGY 和COMPILER DESIGN AND OPTIMIZATIONS
- 第三和四章都是写自己技术点和方案设计，就是要讲自己的故事
- 先建立自己建立的基本概念和关键逻辑（比如Table.1)
- 尽量避免制造很多彼此无关联的新概念
- 建立的新概念要么加粗，单独给一段进行描述，并引起读者注意

- EVALUATION 、RELATED WORK 、CONCLUSION 的逻辑结构大多都是八股结构，详细学习一篇后可作为下一次写作的结构模板进行仿写。

1 INTRODUCTION

1 INTRODUCTION

- 首先引入本文Memory-intensive op这一核心，体现本文现有围绕*compute-intensive* op开展工作的区别。Machine learning models usually consist of two types of operations: *compute-intensive* operations and *memory-intensive* operations. Memory-intensive operations account for an even more significant portion of execution time than their compute-intensive counterparts.
- 用Memory-intensive op量化占比与统计数据表明热度。With an average ratio of 63.2% in execution time and 89.6% in total kernel numbers, memory-intensive computation has already become a dominating factor that significantly impacts the training/inference efficiency of many recent DNN workloads. Moreover, the ratio of computing power to memory bandwidth has also drastically increased on the recent generations of GPU architectures. 说明mem-intensive op随加速器的演变潮流发展，会越来越重要【十分美妙的写法，通过硬件的技术发展趋势，表名软件问题会越来越重要】

$$\frac{\text{computing power}\uparrow}{\text{memory bandwidth}}$$

- 【全文的第一章图很重要，用数字佐证全文核心观点的重要性，一上来就要用“硬证据” 立住你的题】the average portion of the execution time contributed by the memory-intensive operations from the five models of Figure.1 increases to as high as 76.7% on A100.

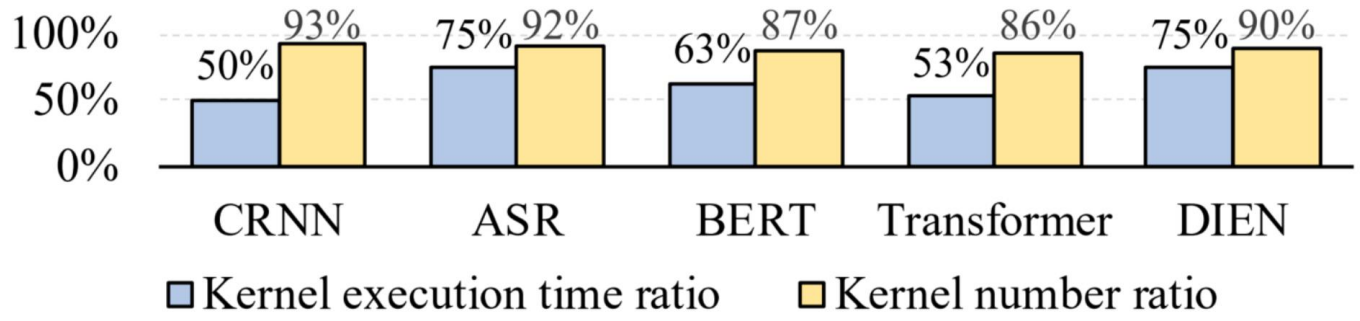


Figure 1: Ratio of memory-intensive computations. The ratio is the proportion of memory-intensive ops' metrics to those of all GPU kernels under study. Statistics on execution time and kernel count are collected from TensorFlow (v1.15) execution.

1 INTRODUCTION

- Memory-intensive computations的kernel launch表面问题: The overhead caused by memory intensive computations mainly comes from intensive off-chip memory access, severe CPU-GPU context switch and high framework scheduling cost due to the large amount of kernels required to be launched and executed.
- ML模型结构多样化引出JIT编译的需求：there are various ML model structures and innumerable customized variants, requiring just-in-time (JIT) optimizations for a given arbitrary model structure rather than ad-hoc solutions.
- （重点凝练句）现有方案的不足：However, a unique set of new challenges emerge from executing these memory-intensive ML models in production. （两个挑战（定位在第二章）为提出的两个性能瓶颈和技术点埋伏笔）
  - *First*, complex two-level dependencies **（大概念）** combined with just-in-time demand exacerbates training/inference inefficiency (Sec.2.3.1) (redundant computation, or skipping fusion) （子问题）
  - *Second*, irregular tensor shapes **（大概念）** lead to poor parallelism control and severe performance issues(Sec.2.3.2)

1 INTRODUCTION

- （重点凝练句）承接上文，由第二章的挑战引入本文第三章的两个技术点To address these limitations, we propose *AStitch*, a machine learning optimizing compiler that opens a new multi-dimensional optimization space（自我评价） for memory-intensive ML computations by supporting efficient just-in-time（呼应前文JIT）operator stitching（与标题呼应） for arbitrary memory-intensive subgraphs.
  - We propose *hierarchical data reuse* technique (Sec.3.2) to address the complex two-level dependencies and enlarge fusion scopes.
  - An *adaptive thread mapping* technique (Sec.3.3) is also proposed to adapt different input tensor shapes and generate proper thread mapping schedules for maximizing hardware utilization and parallelism.
  - Finally, we make several key design observations (Sec.4), and by leveraging them we design a compiler to enable the proposed optimizations automatically.

1 INTRODUCTION

- Contributions【八股】
  - **揭示了非CV范畴的ML模型的性能问题**：reveals performance-critical factor in non computer vision machine learning models
  - **解决了两个性能问题**：tackles two major performance issues: inefficient fusion and inputs with irregular tensor shapes
  - **访存op为核心**：first work to optimize memory-intensive ML computations
  - **整体软件形态为JIT编译器**：designs a compiler, just-in-time, both training and inference
  - **经过产品化考验**：production-ready ML compiler

1 INTRODUCTION

- （重点图示）基本思想就是"stitch" many small and basic fusions
- 【值得学习】如何用一张图把你这篇论文的基本思想表达出来？
  - 先用文字表示出来
  - 把你文字的“对象”用图形表示出来
  - 用对比的方式，将你方法和baseline方法对“对象”影响得到的不同结果表示出来
  - We use “stitching” in this paper to differ our advanced fusion techniques from the existing ML compiler fusion approaches: our expansion of the current fusion scope is to "stitch" many small and basic fusions enabled by the current works into much larger and broader fusions.

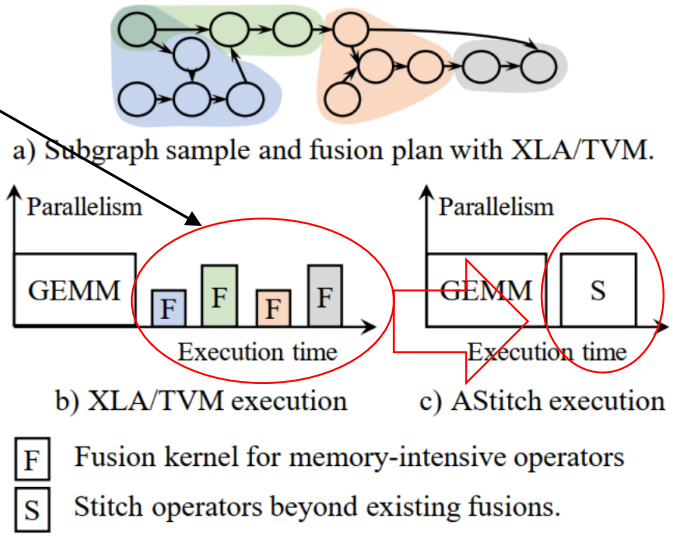
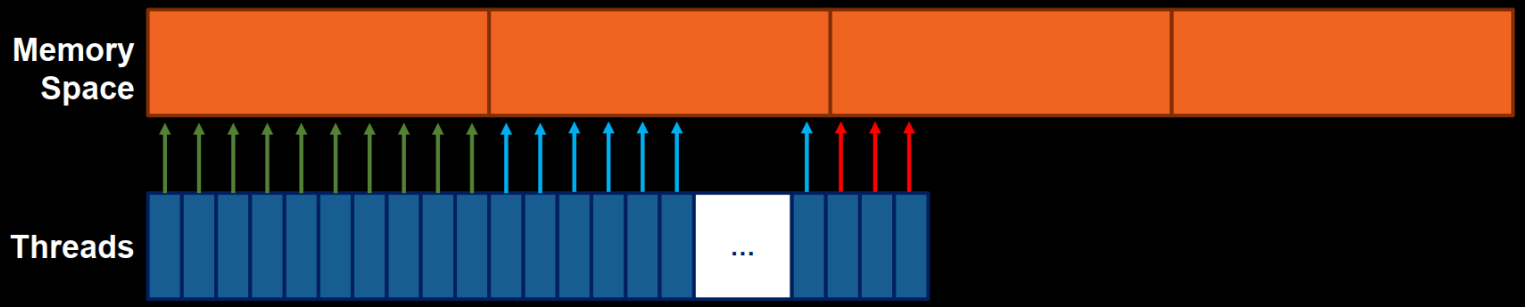


Figure 2: Conceptual illustration of how *AStitch* outperforms XLA and TVM for processing memory-intensive subgraphs. *AStitch* stitches large scope of memory-intensive operators together to reduce non-computation overhead and increase parallelism.

AMD材料中有提到按照cache line进行访存行为的融合，减少访存的次数

Memory Access Coalescing

- Combine memory access to the same cache line
  - Increase effective memory throughput



AMD

2 BACKGROUND AND CURRENT CHALLENGES

2 BACKGROUND AND CURRENT CHALLENGES

- 2.1 Essential Memory-Intensive Ops in Current Models
  - 先交代几个名词之间的关联关系，引出两个关键的op（由大到小，引出关键细节）A machine learning workload => graph => subgraphs with a set of tens or even hundreds of memory intensive operators=> *element-wise ops* and *reduce ops*
  - There are two types of widely adopted operators that cover the majority of the memory-intensive computations in modern machine learning models: *element-wise ops* and *reduce ops*.
  - 下面介绍细节，用一张图的case study说明为什么这两类op是访存密集
  - As shown in Figure.3, the elements in an element-wise op are processed independently in an element-wise manner.=> *light* & *heavy*
  - A reduce op takes a tensor as input and reduces its one or more dimensions. =>
    - *Row-reduce*: *Row-reduce* for one row is usually organized within a thread block, in which adjacent threads read continuous memory addresses for efficiency.
    - *Column-reduce* => one or several GPU thread block

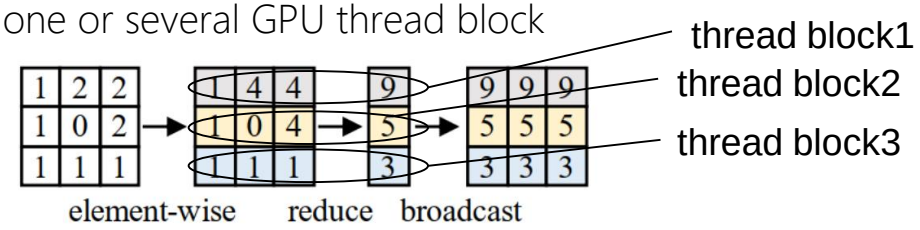


Figure 3: Typical memory-intensive operations.

2 BACKGROUND AND CURRENT CHALLENGES

- 2.1 Essential Memory-Intensive Ops in Current Models
  - Due to the high frequency of *reduce* and *broadcast* applied in modern machine learning computation graphs, tensor shapes between operators become increasingly diverse.

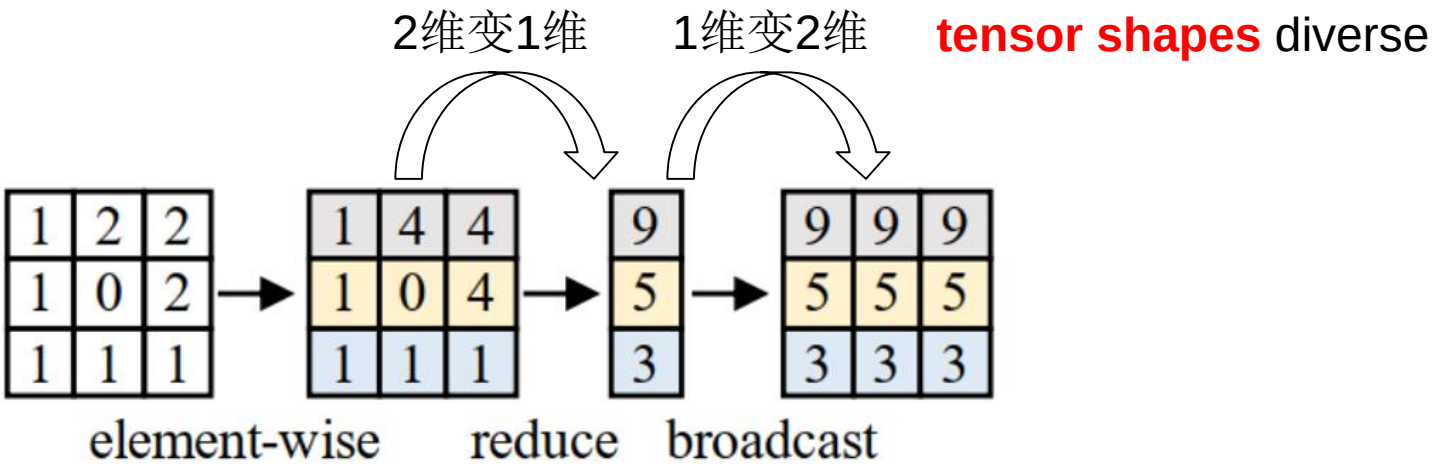


Figure 3: Typical memory-intensive operations.

2 BACKGROUND AND CURRENT CHALLENGES

- 2.2 Memory-Intensive Op Fusion
  - 先介绍已有工作kernel fusion工作能达到的表面效果（减少mem access）：State-of-the-art works(XLA and TVM)=> kernel fusion=> reduce off-chip memory access, CPU-GPU context switching, framework-level operator scheduling overhead induced by frequent kernel launching.
  - Fusion技术定位（基础支撑性技术）是代码生成过程：One of the most fundamental factors of fusion is code generation ability.
  - 目前融合的基本思路（fusion decisions）是模式匹配（pattern matching process）
  - 现有方法的基本思路TVM/XLA's code generators deal with all data dependencies with per-element input inline to merge producer with consumer together.
  - 现有限制引出2.3内容：limitations for such code generation approach in Sec.2.3.

2 BACKGROUND AND CURRENT CHALLENGES

- 2.3 Major Limitations of the State-Of-The-Arts
  2.3.1 挑战一：*看似在讲挑战，实际是想通过"Two-Level Dependencies"的概念引出 "Element-level" Challenge I: Complex Two-Level Dependencies Combined With Just-In-Time Demand Exacerbates Training/Inference Inefficiency.*
  - Operator level dependency describes operator connection represented in a subgraph, e.g., operator *B* and *C* depends on operator *A* in Figure.4.
  - Element-level dependency indicates the dependency between elements within tensors, such as the element 9 depends on elements 1, 4, 4 for *reduce* in Figure.3.
    （这才是重点概念，配合图3实际的例子，详细到具体的element依赖）
  - 强调JIT，区别定制优化there are various machine learning model structures and innumerable customized variants, demanding just in-time (JIT) optimizations for a given arbitrary model structure rather than ad-hoc solutions.
  - 看似再讲“observation”，实际在自卖自夸：Constrained by these unique challenges, we make a key observation that current machine learning optimization compilers (e.g., XLA[11], TVM[18]) cannot perform efficient fusion under such two-level dependencies.

瓶颈关键词➔背后技术原因（自造概念）➔引出具体原因

- root causes behind these limitations as follows.
- ***引出瓶颈关键词（fusion strategies）Key Inefficiency:*** *large number of kernels generated by the ineffective fusion strategies for memory-intensive subgraphs.*
- 瓶颈关键词背后的技术原因（one-to-many elment-level dependencies）State-of-the-art compilers (e.g., XLA and TVM) cannot efficiently fuse two common memory-intensive patterns due to the inability to deal with one-to-many elment-level dependencies
  - (1) reduce ops with its consumers (e.g., orange circles in Figure.4),
  - (2)costly element-wise ops followed by *broadcast* ops (e.g., blue and green circles in Figure.4).

elment-level dependencies是作者提出的概念，不是普适性的概念。那出现inefficiency的原因到底是什么呢?

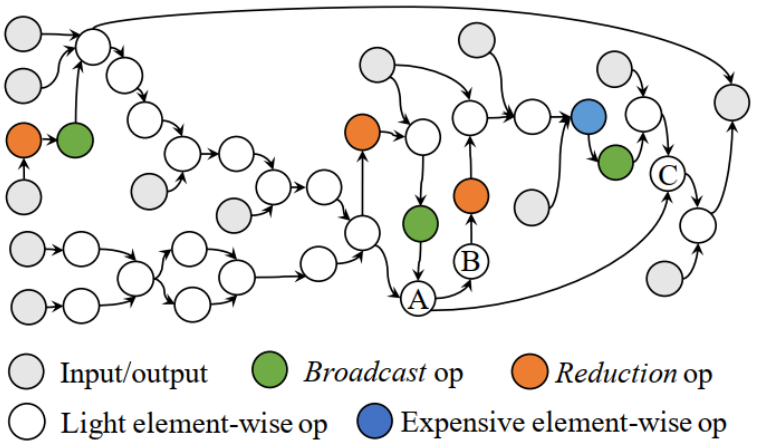


Figure 4: A typical subgraph in a Transformer model.

吊完胃口后➔真正的原因1（基本原理一定要简单易懂，一说就懂，特别是“真正的原因”）

某项能力的缺失

- *大量重复性计算Heavy redundant computation.*
  - neither XLA nor TVM communicates intermediate results between threads; they only leverage per-thread registers to fuse ops in the compiler.
  - one-to-many element-level dependencies
    - one element generated by the producer is required by multiple elements of the consumer(s)
    - each thread of the consumer will independently compute this common element

多生产者➔单个消费者

  - However, *power* will recompute 128 times for the same value in 128 different threads because the compiler cannot cope with the one-to-many dependency effectively with its automatic strategy.
  - *Power* is an expensive element-wise op and requires a large number of cycles to produce data.
  - When the tensor shape is large, it requires several waves of threads, causing notable waste of GPU resources caused by redundant computation of *power*.
  - It is quite tricky to decide how to organize intermediate data for reuse automatically

补齐能力的缺失

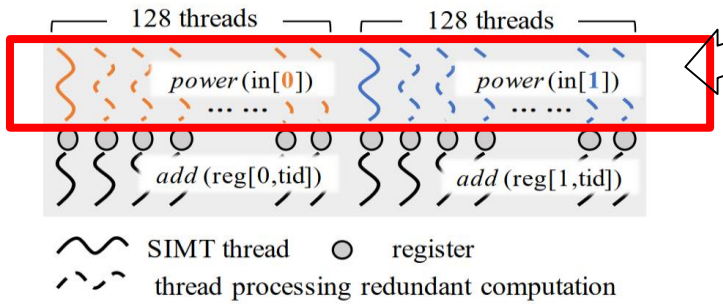


Figure 5: Redundant computation in TVM when attempting to fuse *power<2>* - *broadcast<2,128>* - *add<2,128>*[1] automatically with compiler. Different colors for *power* represent threads that process different elements in the input tensor.

真正的原因2（以承上启下的方式体现研究的深度）

- （承上“原因1的临时方案”，启下“带来的新问题”）执行的核函数数量多
  More kernels are generated for execution.
  - 针对原因1的“临时性” 解决方案：To avoid these redundant computations, state-of-the-art designs tend to skip fusion when encountering one-to-many dependencies from the two patterns.
  - “临时性” 解决方案带来的新问题skips fusion => a large number of kernels for memory-intensive operators =>kernel launching overhead and off-chip memory access
  - 解决问题的关键思路：effective fusion should merge all the memory-intensive operators between two the compute-intensive operators
  - 现有方案的这这个问题的不足：TVM =>skips fusion upon *reduce* ops (1) =>fuse for pattern (2)
  - 略写operator level：at operator level, one-to-many dependencies, in which an op is the producer of multiple ops, may also lead to redundant computations.

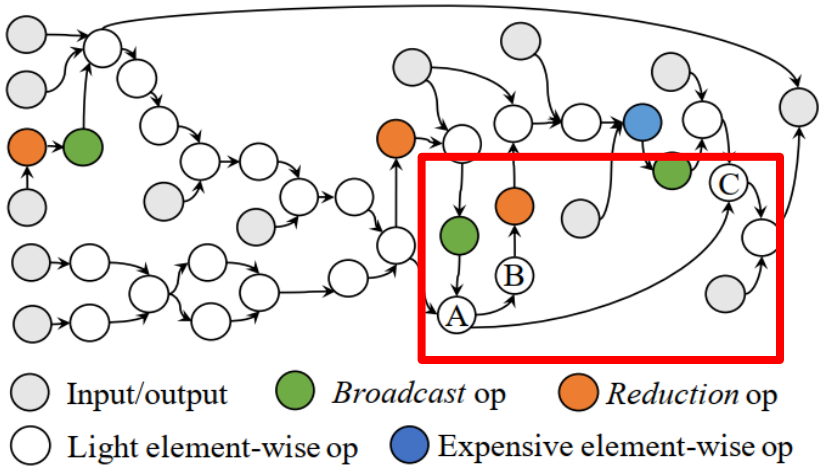


Figure 4: A typical subgraph in a Transformer model.

2 BACKGROUND AND CURRENT CHALLENGES

- 2.3 Major Limitations of the State-Of-The-Arts
  2.3.2 挑战二，张量数据维度的变化对并行的影响： *Challenge II: Irregular Tensor Shapes in Real-World Production Workloads.*
- 【“临时性” 解决方案的不足】确定性*Tensor Shapes**的优化虽然能有好的并行性，但是实际场景无法确定**Tensor Shapes**，所以还需要**JIT**的能力才行*： This makes it challenging for compiler to generate kernels with good parallelism because it demands JIT optimization under the given tensor shapes that are not known in advance. ➔ lack of adaptive designs
- **【进一步点破**Irregular Tensor Shapes**影响性能的原因——**thread-blocks**的**partition】*Key Inefficiency: irregular tensor shapes lead to either too many small partitions or too few large partitions on GPU.*
-

2 BACKGROUND AND CURRENT CHALLENGES

- *原因一：small block size issue* （DIEN model）
  - This is because there is an upper-bound number of thread-blocks that GPU can concurrently execute; when the thread-block size is too small, the concurrency at any given time is also low.一堆小size的block要同时运行➔一个GPU能够同时运行Block的数量有限(没有那么多寄存器)➔图6a的waste
- *原因二：small block count issue* （Transformer model）
  - 编译器(XLA➔64x1024)无法产生能够占满整个GPU（V100➔160x1024）的任务
    - a V100 GPU can concurrently schedule 160 thread-blocks for the same block size.
    - XLA auto-generates 64 thread blocks with the size of 1024
  - Serious hardware under-utilization

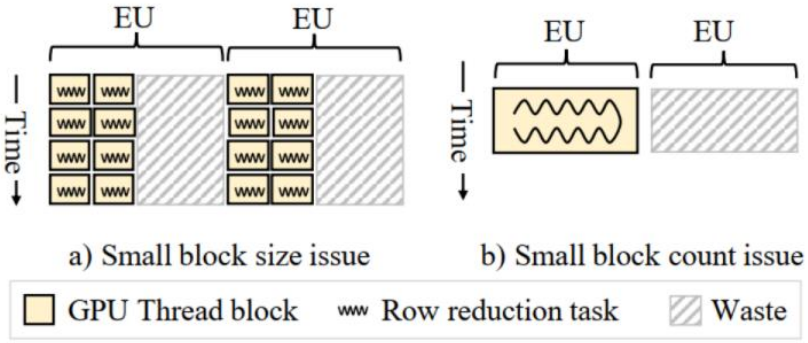


Figure 6: Typical poor parallelism issues in existing works.

3 KEY DESIGN METHODOLOGY

3 KEY DESIGN METHODOLOGY

- *先承上启下，过渡一下*
  - *hierarchical data reuse* (Sec.3.2) => challenge I (Sec.2.3.1)
    - *Two-Level Dependencies=>**large number of kernels generated***
  - *adaptive thread mapping* (Sec.3.3) to address challenge II (Sec.2.3.2) =>*irregular tensor shapes* .
- 3.1 Operator-Stitching Scheme Abstraction**（讲故事前，先介绍角色，介绍我定义的新概念）**
- four types of stitching schemes**（四个新概念）** =>（关联到不同**技术选型**的影响）dependency, memory hierarchy and parallelism

Scheme	Dependency	Memory Space	Locality v.s. Parallelism
Independent	None	None	-
Local	one-to-one	Register	-
Regional	one-to-many	Shared memory	CAT locality first
Global	Any	Global memory	Parallelism first

用Tradeoff展开剧情，增强设计章节的可读性

- **【值得学习：**Trade-off**是读者想看的剧情，过犹不及，中庸之道好】**However, sometimes optimizing for block locality hurts parallelism, leading to poor overall performance.
  - Block locality ➔使用share mem，仅block内可见➔single thread block ➔那就不好并行了，可利用的计算资源受限于max block size(1024)的限制
  - prevents parallelism ➔mapping threads onto more thread block➔使用global share mem来交换中间数据的速度要比share mem来的慢
  - tradeoff between locality and parallelism (i.e., Regional vs Global)

Scheme	Dependency	Memory Space	Locality v.s. Parallelism
Independent	None	None	-
Local	one-to-one	Register	-
Regional	one-to-many	Shared memory	CAT locality first
Global	Any	Global memory	Parallelism first

3 KEY DESIGN METHODOLOGY

- 3.2 **【第一个技术点】**Hierarchical Data Reuse Illustration
- *Data Reuse.* ***(用什么存储介质传递数据是关键)***
  - two-levels of data reuse across memory hierarchies (i.e.,registers, shared memory and global memory)
  - *Element-level data reuse.*
    - The result can be maintained on GPU shared/global memory buffer for its consumer(s) to reuse in regional/global stitching scheme.
  - *Operator-level data reuse.*
    - For local stitching scheme, the to-be-reused data is maintained on register, while the data is maintained on shared/global memory for regional/global memory.
  - **感觉这两个层做的事情是相同的事情——用不同类型的片上显存传递中间结果**
- *Global Barrier.——**限制**Constraints*
  - 1. Block**总数限制**total number of GPU thread-blocks [50]
  - 2. block**死锁：** dead-locks between active and inactive thread-blocks (Sec.3.3) =>limit the thread-block number => still retaining high parallelism.
  - *global* stitching inlines the implicit global thread barriers between kernel calls into a single kernel. inline**是**C++**关键字,函数指定为内联函数解决频繁调用的函数大量消耗栈空间（栈内存）的问题**

Case study显示技术选型以及效果（选型原因未谈）

- *Kernel Form Illustration.*
  - *regional* scheme for *reduce.1*,
  - *global* scheme for *power.1* and *reduce.2*,
  - *independent* scheme for *multiply.1*,
  - and *local* scheme for other operators
- *AStitch* eliminates 3 kernel launches with fine-grained data management and multi-level thread barriers, which reduces CPU-GPU context switch and framework scheduling overhead (with extra lightweight thread barriers in a kernel).
- XLA forms 4 kernels
- TVM will form 3 kernels with redundant computations
  - *power.1* and *reduce.2* are merged into one kernel

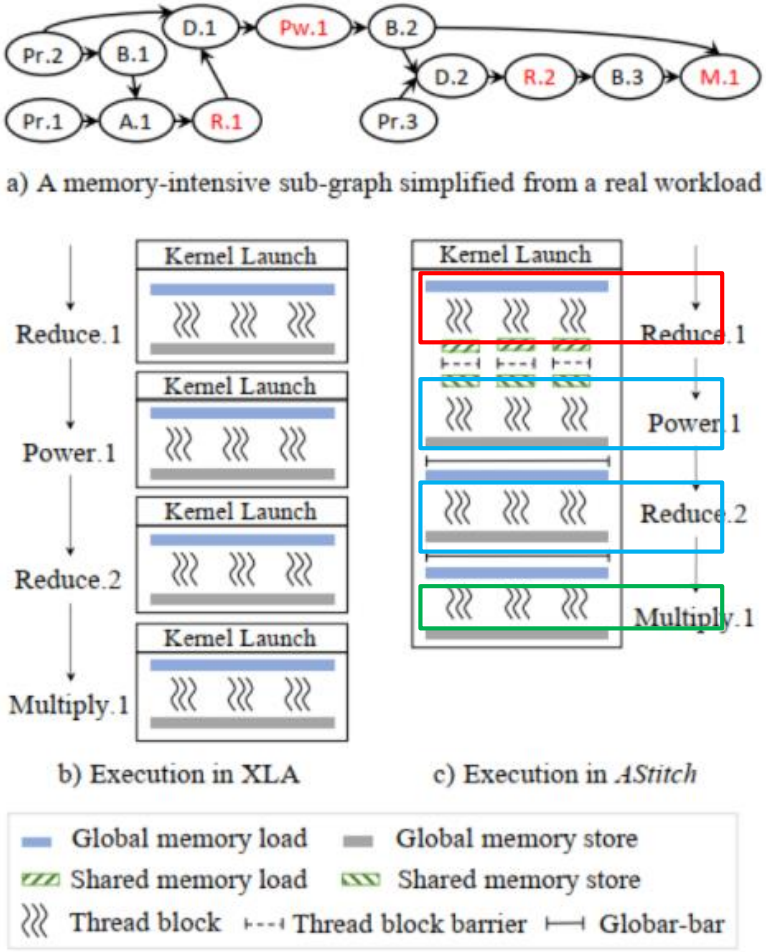


Figure 7: Execution scheme of a memory-intensive sub-graph. *AStitch* reduces kernel launches and off-chip memory access with hierarchical data reuse. Pr: parameter. A: add. B: broadcast. R: reduce. D: divide. Pw: power. M: multiply.

读到这里我有了疑问：为什么 *regional* scheme for *reduce.1*, *global* scheme for *power.1* and *reduce.2* ?

- At *element level*, during tensor processing, the frequent occurrence of *reduce* and *broadcast* may also incur many one-to-many dependencies.
- 是否跟这个op后边的broadcast相关?

Scheme	Dependency	Memory Space	Locality v.s. Parallelism
Independent	None	None	-
Local	one-to-one	Register	-
Regional	one-to-many	Shared memory	CAT locality first
Global	Any	Global memory	Parallelism first

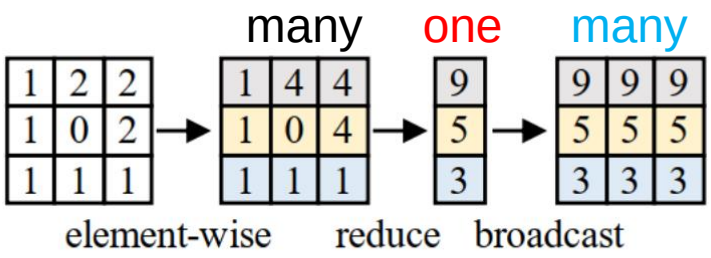
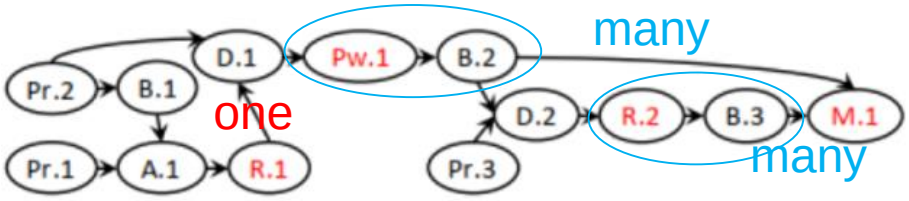


Figure 3: Typical memory-intensive operations.



a) A memory-intensive sub-graph simplified from a real workload

Figure 7: Execution scheme of a memory-intensive sub-graph. *AStitch* reduces kernel launches and off-chip memory access with hierarchical data reuse. Pr: parameter. A: add. B: broadcast. R: reduce. D: divide. Pw: power. M: multiply.

3 KEY DESIGN METHODOLOGY

- 3.3 **【第二个技术点】** Adaptive Thread Mapping(block size**大了不行，小了也不行**)
  - As discussed in Sec.2.3.2, state-of-the-art ML compilers lack of designs to support irregular tensor shapes presented in production workloads (Challenge II), causing significant performance issues. (**写作：承上启下**)
- Note that what suffer from irregular tensor shapes are mainly *reduce* ops, which are the most time-consuming operators among memory-intensive computations. **缩小优化目标到**reduce op**一人身上**
- Task packing **（增大**block size，**降低**block count**）**
  - *Horizontal packing （空间维度）*
    - pack multiple small blocks, each of which processes the reduction of one row, into one large thread block. => small block-size issue shown in Figure.6-(a)
    - H-packing: enlarge block size **原来：**32**个**block(32thread)=>**现在：**1**个**block (32x32=1024thread)
    - Every 32 threads in the thread-block process a row
      原来：1**个小**size block**处理一个**element=>**现在：**1**个大**size block**处理一个**row(**包含**32**个** thread)

Warp and wave的定义(摘自Real-Time Rendering 4th Edition)

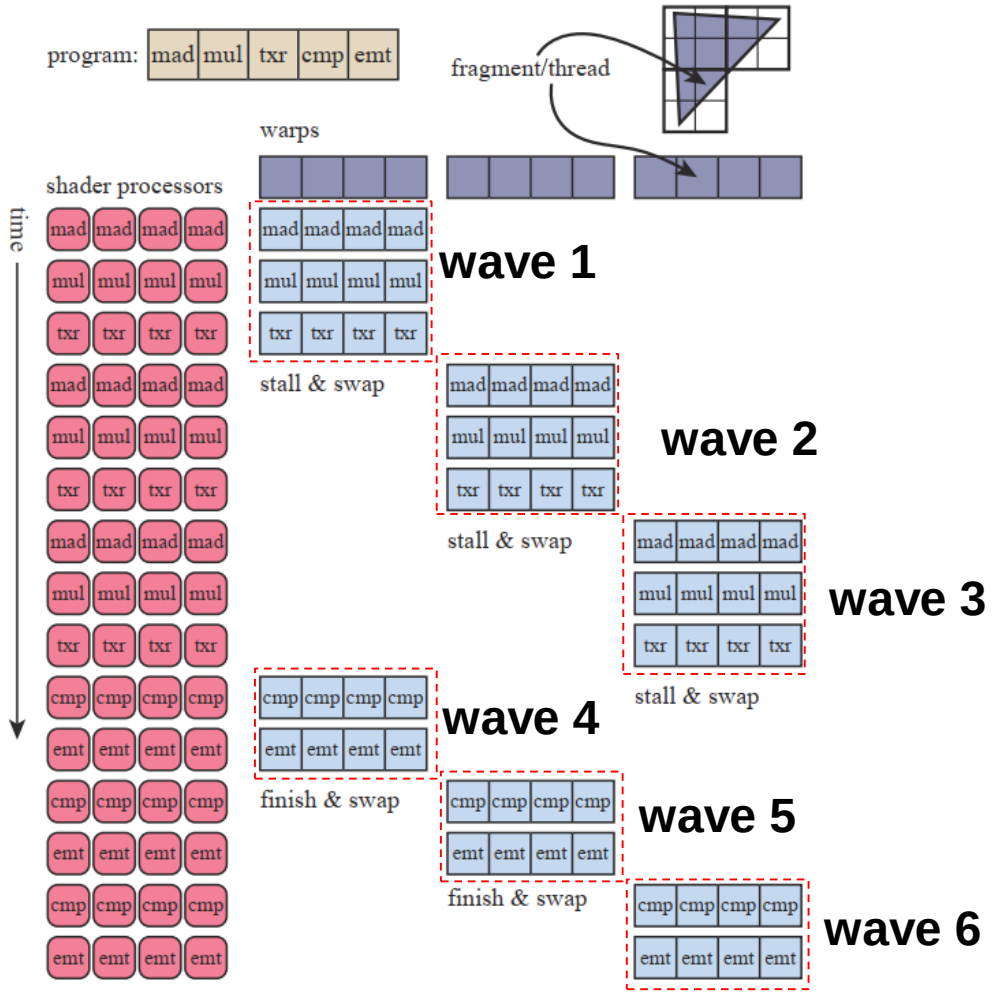
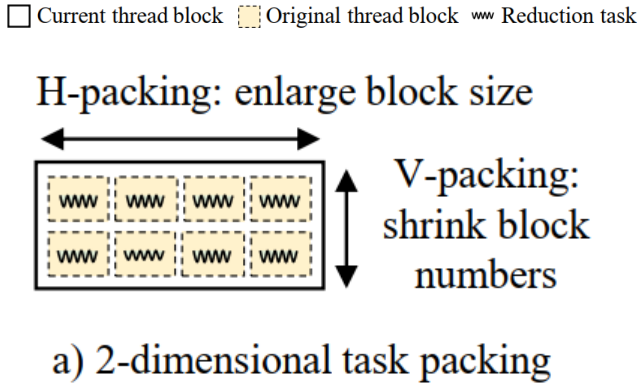


Figure 3.1. Simplified shader execution example. A triangle's fragments, called threads, are gathered into warps. Each warp is shown as four threads but have 32 threads in reality. The shader program to be executed is five instructions long. The set of four GPU shader processors executes these instructions for the first warp until a stall condition is detected on the "txr" command, which needs time to fetch its data. The second warp is swapped in and the shader program's first three instructions are applied to it, until a stall is again detected. After the third warp is swapped in and stalls, execution continues by swapping in the first warp and continuing execution. If its "txr" command's data are not yet returned at this point, execution truly stalls until these data are available. Each warp finishes in turn.

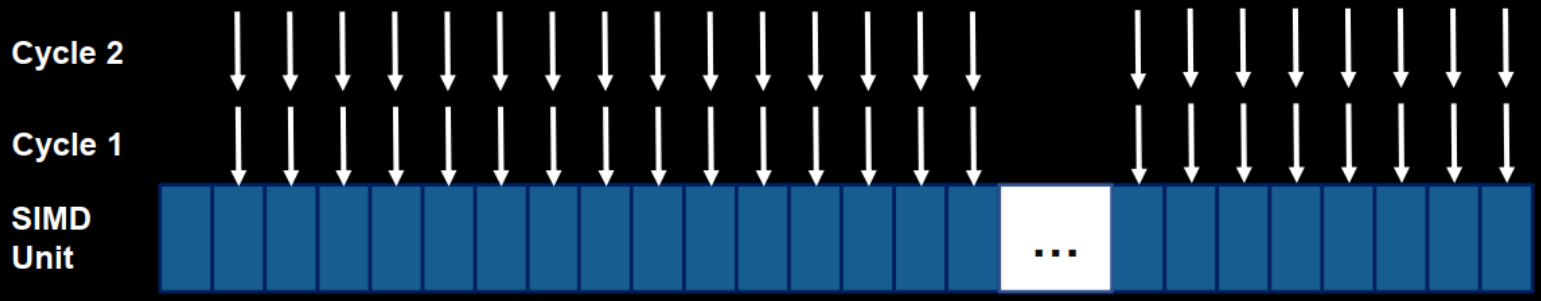


- *Vertical packing* (时间维度????，在一个block中执行3个wave????)
  - pack multiple waves of thread-blocks into one wave to meet the requirement of a global barrier.
  - V-packing: shrink block numbers
  - The block size is unchanged in the vertical packing and each thread processes elements from multiple tasks in order.

AMD的Warp and wavefront

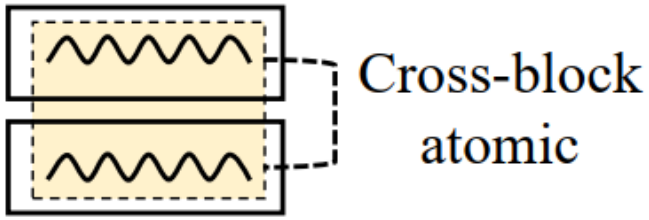
SIMD Instruction Execution

- 64 threads in a block are scheduled together
  - Warp / wavefront
  - Single instruction multiple data



3 KEY DESIGN METHODOLOGY

- Task splitting
  - *task splitting* is to split the task within one thread-block into several thread-blocks to increase block count, in case there is under utilization problem caused by small block count (Figure.6-(b)).
  - a cross-block atomic to increase parallelism（第一次出现，且唯一一次出现的概念）
  - （到底是如何增加block count的还是有点不太理解，细节太少）



b) Task splitting

Current thread block Original thread block Reduction task

4 COMPILER DESIGN AND OPTIMIZATIONS

4 COMPILER DESIGN AND OPTIMIZATIONS

- 基本思路，就是把第三章的两个技术（stitching scheme，thread mapping）进行组合的过程：It is required to determine the stitching scheme along with thread mapping for all the operators just-in-time.
- 但是，组合的过程也有讲究。即，实现过程会遇到的问题。通过简单枚举过程实现会有参数空间爆炸问题：However, enumerating the schemes for each operator results in combinatorial explosion. （避免直接讲你实现了哪些模块具体咋实现的。而是**用实现过程中的科学性问题来吊起读者胃口**，这种写法不枯燥，能够引起读者思考）
- **看似在讲**observation，**其实在讲自己的结题思路：**a key observation: only need to determine the scheme for several key operators but not all ops. key operators propagate to all the other operators in the subgraph.
- **用研究问题的方式来写**design&optimization**章节，值得学习**

4 COMPILER DESIGN AND OPTIMIZATIONS

- 4.1 Stitching Scope Identification**先建立了一个“scope”的新概念**
- ***方法的指导思想：****AStitch* stitches as large scope of memory-intensive operators together as possible (under resource constraints) for a given ML computation graph.
  - BFS algorithm to identify memory-intensive sub-graphs
  - replaces each of the sub-graphs with *stitch op*.
  - To enlarge stitching scope: *AStitch* groups disconnected *stitch ops* together and forms a larger *stitch op*, which we call *remote stitching*.
  - merge two *stitch ops* together if they have no data dependency
  - A constraint to form a *stitch op* is that no cyclic dependence is allowed.
  - One stitch op=>one CUDA kernel

4 COMPILER DESIGN AND OPTIMIZATIONS

- 4.2 Key Design Observations
- **看似是现象，其实是指导原则（融合策略与线程映射的选型）**
- *Observation-A:* If an operator is of *local* scheme, its thread mapping can be determined by propagating from its consumer's thread mapping.
- *Observation-B:* The patterns of *reduce* and expensive element-wise ops followed by *broadcast* ops need to be supported by either *regional* or *global* scheme because the two patterns induce complex element-level *one-to-many* dependencies (Table.1).

指导原则A:
stitching scheme➔thread mapping

指导原则B:
（红蓝）两类mem-intensive负载的stitching scheme，但是依然没有解答之前想到的这个问题

为什么*regional* scheme for *reduce.1*, *global* scheme for *power.1* and *reduce.2*?

- At *element level*, during tensor processing, the frequent occurrence of *reduce* and *broadcast* may also incur many one-to-many dependencies.
- 是否跟这个op后边的broadcast相关?

Scheme	Dependency	Memory Space	Locality v.s. Parallelism
Independent	None	None	-
Local	one-to-one	Register	-
Regional	one-to-many	Shared memory	CAT locality first
Global	Any	Global memory	Parallelism first

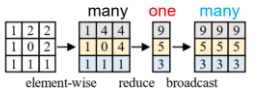
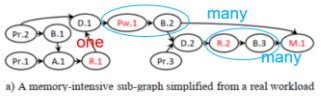


Figure 3: Typical memory-intensive operations.



a) A memory-intensive sub-graph simplified from a real workload

Figure 7: Execution scheme of a memory-intensive sub-graph. *AStitch* reduces kernel launches and off-chip memory access with hierarchical data reuse. Pr: parameter. A: add. B: broadcast. R: reduce. D: divide. Pw: power. M: multiply.

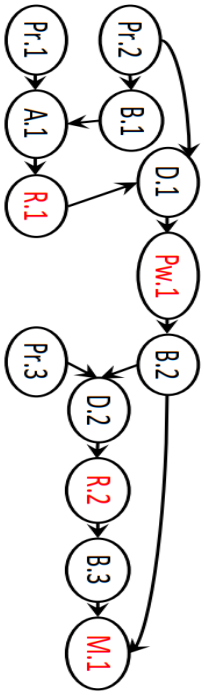
4 COMPILER DESIGN AND OPTIMIZATIONS

- 4.3 Automatic Compiler Optimization Design
- **【值得学习的**case study**设计方法】复用前文**fig.7 case study**的**DEG**图，即跟前文建立联系，承上启下；又能减少引入新的**case**信息量，避免绕晕读者**

2) 老大线程映射策略扩散到其他op

1) 先分组圈地，确定老大老二op

3) 确定老大和老二的stitch策略（其他op都是local）



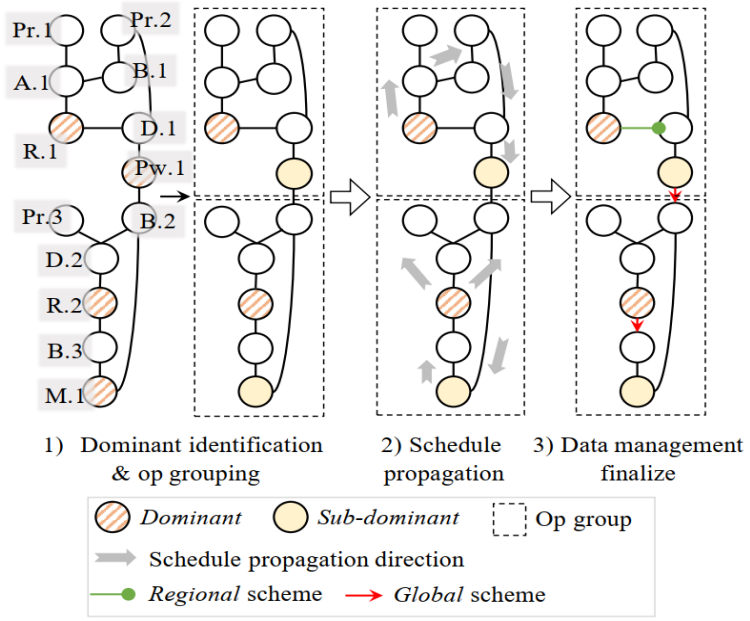


Figure 9: Schedule propagation and data management planning for the graph topology shown in Figure.7-(a).

4 COMPILER DESIGN AND OPTIMIZATIONS

- 4.3 Automatic Compiler Optimization Design
- Step 1: dominant identification and op grouping.
- key operators => *dominant ops*.

1) 先分组圈地，确定老大老二op

基本思路是“抓大放小”，确定dominant op的策略后，inline到同一scope范围内的其他op上

- *reduce* and expensive element-wise ops followed by broadcast are candidates for *dominant ops*.
  - Under this rule, *reduce.1*, *power.1* and *reduce.2* are candidates for *dominant ops* in Figure.7-(a).
  - If two candidate *dominants* connect with each other through ops of only *local* scheme, *AStitch* chooses one as final *dominant op*, and regards the other as sub-dominant.
  - only need to determine the thread mapping schedule for the *dominant op*
  - *AStitch* prefers to choose reduction(reduce) as the final *dominant op* when merging.
  - different groups are connected with each other through only dominant and sub-dominant ops.

4 COMPILER DESIGN AND OPTIMIZATIONS

2) 老大线程映射策略扩散到其他op

- 4.3 Automatic Compiler Optimization Design
- Step 2: adaptive thread mapping and schedule propagation.
  - generates the parallel code
  - propagates the thread mapping schedule within the corresponding group
- *Tensor Shape Adaptation*.
  - Task packing and splitting **跟前文一样**
- *dominant merging* in Step-1 can enable more operator-level data reuse (Sec.3.2.1).
- irregular tensor shapes->many operators as consumer dependence on one operator as producer -> the producer's result cannot be reused. -> redundant computations (operator-level)

4 COMPILER DESIGN AND OPTIMIZATIONS

3) 确定老大和老二的the stitching schemes策略（其他op都是local）

- Step 3: Finalization.
  - determines the stitching schemes for the *dominant* and *sub-dominant* ops
  - We apply locality check to identify the stitching scheme between *regional* and *global*.
- *Passive block-locality checking*
  - Check if its consumers read the same range of data within the same block
  - *reduce* ops ➔因为require more computation ➔ 所以prioritize parallelism
- *Proactive block-locality adaptation*
  - For an op group that only contains element-wise ops, *AStitch* proactively adjusts the thread mapping schedule of this group to match the block-locality with its producer group
  - element-wise ops ➔因为low-cost computation ➔所以 prioritize locality
- 技术选型与决策没有过多繁琐的解释，简明扼要地给出原因。
- ***【写作方法】越到末端的技术细节，解释的越少。越是根部的内容越要详细解释***

4 COMPILER DESIGN AND OPTIMIZATIONS

- 4.4 Memory Usage Optimization
- **其实还是再解释**regional**和**global**两种策略的**tradeoff：high shared memory usage（regional） hurts kernel parallelism
- **基本原则是最大化的发挥硬件（**share mem**）的最大容量：** consideration of memory resource limitations and their impacts on performance
- AStitch uses dominance tree algorithm[19] for memory data-flow analysis to maximize the memory reuse for operators.
- If a shared memory request exceeds the hardware limit for a thread-block，*AStitch* alters the stitching scheme of *dominant* and *sub-dominant* ops from *regional* to *globa*l

思考tradeoff**带来的启示：**

- **过度的并行➔浪费硬件资源，浪费数据交换快的**share mem➔block**间数据交互量会与通信开销会增加➔不适用于计算量小但访存密集的任务**
- **过度追求局部性➔降低并行效率➔不适用与计算量大的任务**

4 COMPILER DESIGN AND OPTIMIZATIONS

- 4.5 Resource-Aware Launch Configuration $C_{blocks-per-wave}$
- GPU kernel launch dimension ➔ thread-block count does not exceed the max block count per-wave (*C blocks-per-wave*)**一个**kernel**的范围，要符合**GPU**能够同时启动**block**数量的上限** ➔ thread mapping plan (Task packing and splitting optimization)➔ 编译之后才能知道
- 思考：（*C blocks-per-wave*随加速器不同而不同，是体现不同异构加速器差异特性的关键参数，面向异构加速器运行时优化应该应该好好发掘这个参数*C blocks-per-wave*➔*Max block size*，max register usage）
  - Assume register usage bound
  - calculate *C blocks-per-wave* (register usage bound, planned shared memory usage (Sec.4.4) and specified block size)**如何计算，本文并未详细展开**
  - Max block size (1024)**根据**size**上限走**➔ smaller global barrier overhead
  - as larger block size results in smaller*C blocks-per-wave*,
  - deduce the max register usage allowed according to *Cblocks-per-wave*.
- **得到的经验：**The insight is that, the parallelism may be bounded by shared memory usage but not register usage（block count）, for which we can relax the register usage bound.

5 IMPLEMENTATION

- Currently, we implement *AStitch* as a TensorFlow add-on.
- *AStitch* leverages TensorFlow built-in XLA engine for the system implementation.
- *跟现有软件栈的关系与接口AStitch* accepts the computation graphs represented in XLA but replaces XLA's fusion and codegen passes.
- rewrite computation graph and generate GPU code for *stitch ops*.
-

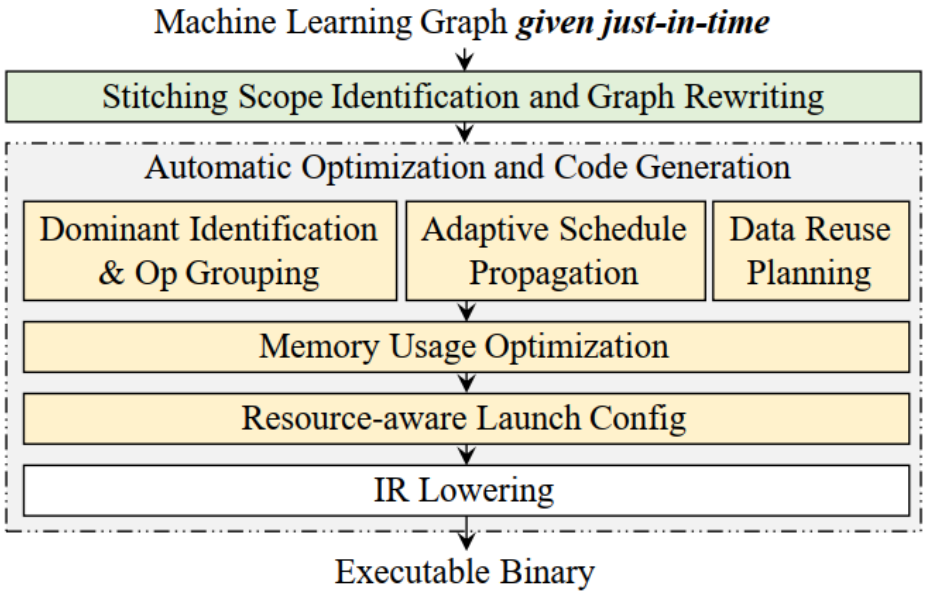


Figure 10: *AStitch* System Overview.

6 EVALUATION

6 EVALUATION（八股写法）

- NVIDIA V100 GPU with 16GB device memory
- CUDA toolkit 10.0 and cuDNN 7.6.

- 6.1 End-to-End Evaluation
- *Workloads*. We use a set of representative memory-intensive machine learning applications as our evaluation workloads

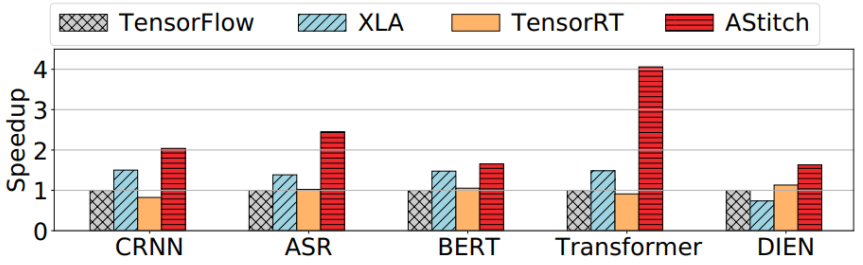
Table 2: Workloads for evaluation.

Model	Field	Train batch-size	Infer batch-size
CRNN	Images	-	1
ASR	Speech	-	1
BERT	NLP	12	200
Transformer	NLP	4,096	1
DIEN	Recommendation	256	256

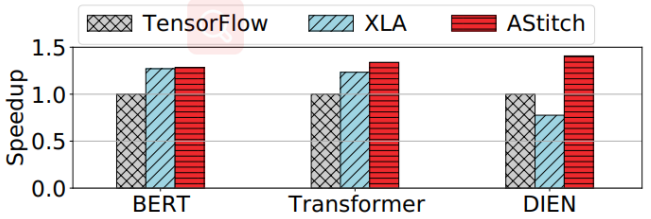
- *Baselines* : TensorFlow (v1.15), TensorFlow XLA (v1.15) and TensorRT (v7.0)
- 两个细节说明
  - We evaluate the speedup of *AStitch* by comparing inference time or the training time of one iteration.
  - We repeat 10 times and use the average performance to validate speedup.

6 EVALUATION（八股）

- 6.1 End-to-End Evaluation
- *6.1.1 Overall Results.*
- **【八股的“四句”段落写法】Fig.11:坐标说明、整体结论、分实验表面数据的现象、个别特殊数据说明与分析**
- *整体结论写开头AStitch* outperforms all other techniques we compare.
- Fig.12:inference 换了一种GPU（NV T4）补充实验
  - For the inference workloads, we have evaluate *AStitch* on NVIDIA T4 GPU
  - we also evaluate *AStitch* along with auto mixed precision (AMP) optimization[2] (Figure.12)
  - 自动混合精度(AMP)



(a) Speedup of inference tasks.



(b) Speedup of training tasks.

Figure 11: End-to-end performance speedup.

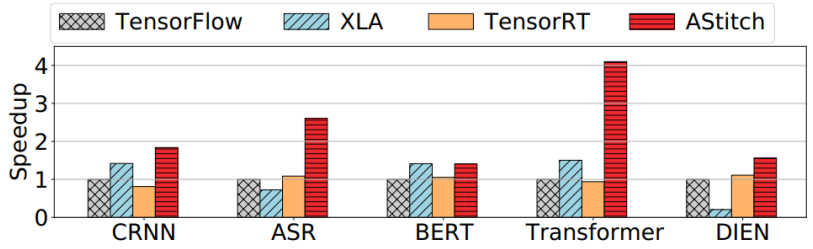


Figure 12: Inference speedup, for which baselines and *AStitch* are all with AMP optimization.

6 EVALUATION

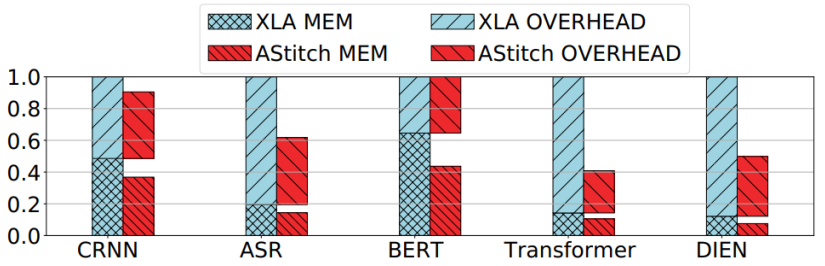


Figure 13: Performance breakdown, without showing the time of compute-intensive ops.

- 6.1 End-to-End Evaluation
- *6.1.2 Performance Breakdown.*
- 【值得学习，细分一下油水来源的“细分项”】
  - memory-intensive op execution (*MEM*)
  - compute-intensive op execution and
  - non-computation overhead(*OVERHEAD*).
- Note that we do not explore multi-stream execution (multi-stream会有什么额外的时间？？？)
- *Fig.13:AStitch* significantly reduces both the execution time of memory-intensive ops (*MEM*) and non-computation overhead (*OVERHEAD*).
- **【八股段落】**Fig.13:**实验目的说明与坐标说明、整体结论、现象举例、取得效果的原因说明**
- 【过渡句】The decrements of *OVERHEAD* mainly comes from kernel call decrements, and *MEM* benefits from parallelism increment. ➔用取得效果的说明引出kernel call decrements**和**parallelism increment**两段文字**

6 EVALUATION

- 6.1 End-to-End Evaluation
- *6.1.2 Performance Breakdown.*
- *Kernel Call Decrements.(**降低内核数量与**CPU-GPU**数据搬移量**)*
  - Thanks to the exhaustive stitching, 65.7% kernel calls of memory-intensive computations are saved on average.
  - reduces 43.2% CUDA memcpy/memset activities

Table 3: Kernel numbers. MEM: kernel of memory-intensive ops. CPY: CUDA memcpy/memset calls.

		CRNN	ASR	BERT	Transformer	DIEN
MEM	XLA	986	496	64	10,132	2,579
	AStitch	297	218	26	2,578	811
CPY	XLA	406	372	25	5,579	628
	AStitch	388	203	10	1,474	422

6 EVALUATION

- 6.1 End-to-End Evaluation
- *6.1.2 Performance Breakdown.*
- *Parallelism Increment*. (**通过**nvprof profiling tool**证明“并行性”的提升**)
- 【值得学习】化“概念” 为具体可测的参数
  - *achieved_occupancy*: *Occupancy*[3] shows whether enough threads are scheduled in parallel , indicating parallelism.
  - *sm_efficiency*: *Sm_efficiency*[10] shows the percentage of elapsed cycles that GPU SM is busy, indicating GPU utilization.
  - top 80% memory-intensive kernels
  - Thanks to the adaptive thread mapping, *AStitch* increases the parallelism and GPU utilization overall.

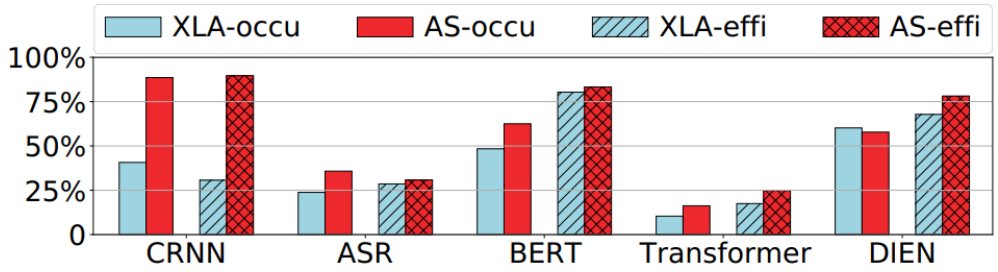


Figure 14: Average parallelism of top 80% memory-intensive computations. AS: *AStitch*. Occu: occupancy. Effi: SM-efficiency.

6 EVALUATION

- 6.1 End-to-End Evaluation

  6.1.3 A Comprehensive Case Study. We make a case study with *CRNN*

- *Ablation Study. (**优化点逐渐增加叠加后，能带来的效果变化过程**)*
- 【值得学习】这种消融实验能够有效与前文的具体优化小点进行一对一的呼应，也能对每一个独立的优化点的效果进行量化
  - We enable the optimizations in *AStitch* one by one to justify and prove the design choice of *AStitch*.
  - XLA： Baseline
  - ATM： + *adaptive thread mapping* (+ 8.9%)
    - increasing the parallelism and GPU utilization
  - HDM： ++ *exhaustive stitching* with *hierarchical data management* onto *ATM* (+ 8.9%+ 8.2%)
    - reducing context switch overhead and offchip memory traffics, and exploring element-level data-reuse.
  - AStitch:： +++ *dominant merging* (+ 8.9%+ 8.2% + 18.7%)
    - enabl

Table 4: Ablation study for *CRNN*.

	XLA	ATM	HDM	AStitch
Time (ms)	23.95	21.98	20.45	17.64

- *Performance Counter Analysis*
- ***通过profile性能计数器的中间参数，能够有效体现优化原理***
  - 1) Parallelism.
    - *achieved_occupancy and sm_efficiency*
    - The top time-consuming kernels shows higher parallelism and hardware utilization
  - 2) Reduced off-chip memory traffic.
    - *GPU global memory load transaction (dram_read_transactions)*
    - *GPU global memory store transaction (dram_write_transactions)*
    - hierarchical data management, which buffers a large portion of intermediate values on-chip
  - 3) Reduced instructions.
    - *fp32 instruction count (inst_fp_32)*
    - reduced redundant computations

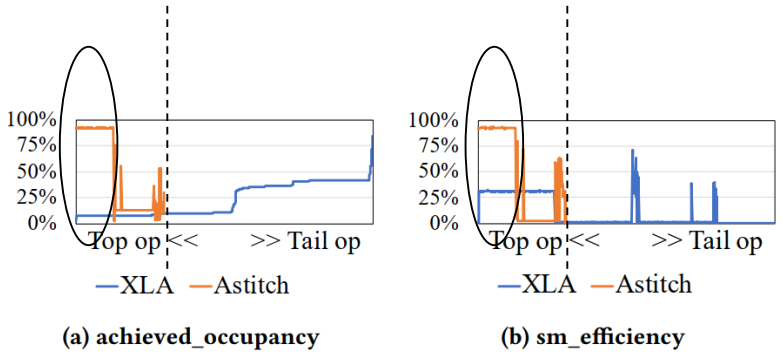


Figure 15: CRNN occupancy and SM efficiency trend. X axis indicates memory-intensive ops in descending order of execution time. Note *AStitch* has less ops.

Table 5: Total performance counters of all memory-intensive ops in *CRNN*. DR_transactions: dram_read_transactions. DW_transactions: dram_write_transactions

	DR_transactions	DW_transactions	inst_fp_32
XLA	104,056,236	63,793,690	1,700,113,391
AStitch	104,022,389	16,302,582	1,675,090,268

6 EVALUATION

- 6.2 Comparing with TVM Ansor: A Case Study 测试方法，观测的性能参数与6.1完全一样就是改变了负载和对比对象
- Ansor[55] on BERT inference case（only support BERT）
- We run Ansor auto-tuning for 2000 measurement trials and choose the best-tuned model for comparison.
- 降低核函数数量方面：Less kernel count:
  - AStitch forms 53% less GPU kernels for memory-intensive ops than Ansor ➔lower context switch overhead
- 提高并行度方面higher parallelism:
  - achieved_occupancy and sm_efficiency
- CPU-GPU间数据传输总量方面：global memory transactions:
  - global memory read/write transactions
  - reducing 40% total off-chip memory transactions

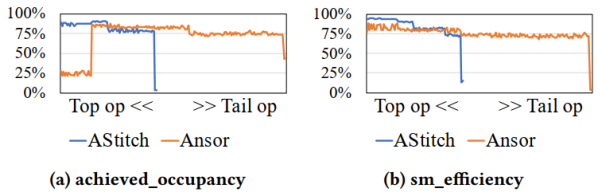


Figure 16: BERT occupancy and SM efficiency trend. Axis has the same meaning with Figure.15. Note *AStitch* has less ops.

6 EVALUATION

- 6.3 Production Evaluation（**产品级的鲁棒性测试**）
- **学术界很少会做这种测试，独具特色**
- *产品收益测试：AStitch* has been deployed into a production cluster and has saved around 20,000 GPU hours on 70,000 tasks within a week. ➔robustness
- 来自产线的负载特征（可供我们之后引用的数据，少数分布式作业占用了超过一半以上的GPU机时）：About 23% jobs are distributed jobs, consuming 56% total GPU time among all machine learning jobs.
- 给出了两点细节
  - The method we estimate the saved GPU hours is that, we run the deployed model with TensorFlow for several iterations at the beginning, and optimize with *AStitch* in later iterations.
  - 以iteration为统计单位：We compute the total time saved by multiplying the number of iterations and time saved per iteration.

6 EVALUATION

- 6.4 Overhead Analysis（**优化设计带来的开销分析**）
- **篇幅虽然不长，但是时堵住审稿人嘴的要点**
- 6.4.1 Optimization Overhead. 编译时间开销
  - We measure the overhead time from accepting the input graph until just before lowering LLVM IR to CUDA binary.
  - Just like compilation overhead, the overhead of *AStitch* is introduced only once for all following iterations of training/inference.
- *6.4.2 Global Barrier Overhead*. 数据传输开销
  - A V100 GPU can accommodate at most 160 such thread blocks concurrently.
  - Thus the overhead of global barrier is no more than 2.72us in *AStitch*, less than the kernel launch overhead on the order of 10 microseconds

7 RELATED WORK

7 RELATED WORK（八股写法）

- 相关工作分成以下四类
- 1）内核融合：There are some works study about **fusion** for machine learning specifically.
  - *AStitch* is orthogonal with the above studies in that it focuses on generating high performance GPU kernels given a large group of memory-intensive operators just-in-time.
- 2）定制优化：There are **ad-hoc optimizations** focusing on specific structures
  - *AStitch* provides a general JIT compiler rather than model-specific optimizations.
- 3）数据预处理：There are some works study about **data preprocessing.**
  - *AStitch* is orthogonal to these data processing works and can be combined with them.
- 4）费计算开销优化：There are some works addressing the issue of **non-computation overhead**.
  - *AStitch* is totally automatic and capable to leverage the highly tuned libraries for compute-intensive computations (cuDNN[8], cuBLAS[7]).

8 CONCLUSION（八股写法：标准的五段话结构）

- 第一段：中心思想句
  - We reveal that memory-intensive computation is a rising performance critical factor in recent machine learning models.
- 第二段：两个技术点
  - 数据重用：We propose *hierarchical data reuse* technique to address the complex dependencies to enlarge fusion scope, reducing non-computation overhead.
  - 线程排布：We propose *adaptive thread mapping* technique to deal with the problem of irregular tensor shapes.
- 第三段：整体软件形态
  - We develop a JIT compiler named *AStitch* integrating the optimizations with high usability.
- 第四段：结果
  - Results show that *AStitch* outperforms state-of-the-art compilers with up to 2.73× speedup.
- 第五段：放大意义
  - We believe *AStitch* fills a long-overlooked gap of machine learning compilers.

总结一下

- 整理一下行文逻辑
  - 表层问题→表层解决（不行）→新技术理念（扩大融合范围；不是最终技术手段）→解决连锁问题（两挑战）
  - 表层问题的表层解决：传统避免融合范围太大的手段，就是很直接的降低融合范围，会造成mem access和上下文Switch开销等问题
  - 解决新理念的连锁问题：融合范围太大会有重复计算、硬件block count上界问题
  - 传统编译器的结题思维：依赖分析→并行效率和局部性原则
  - 我们都知道的任务抽象与我不说你可能不知道的硬件固有限制，两者可能出现的关乎性能的矛盾点
- 思考：
  - 写论文得有自己的“技术理念”，有技术理念才能有影响力，由技术理念牵引出技术点
  - 本文的技术贡献并不是简单的扩大fusion范围，扩大fusion范围是“指导思想”或“技术理念”，而是如何解决融合范围扩大后所带来的连锁问题