AStitch

Enabling a New Multi-dimensional Optimization Space for Memory-Intensive ML Training and Inference on Modern SIMT Architectures

ASPLOS'22

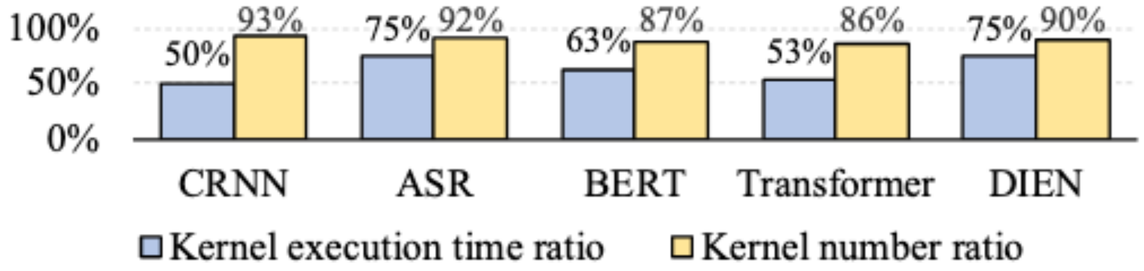
Zhen Zheng,Xuanda Yang,Pengzhan Zhao,Guoping Long,Kai Zhu,Feiwen Zhu,Wenyi Zhao,Xiaoyong Liu,Xiaoyong Liu,Wei Lin,Alibaba Group
Jidong Zhai,Tsinghua University;Shuaiwen Leon Song,University of Sydney

Introduction

Why target at Memory-intensive ML ?

- two types of ops:
  - Comput-intensive(e.g.Convolution) 👍
  - **Memory-intensive**(e.g. element-wise & reduction ops) 👎
- Optimize memory-intensive computations becomes crucial and urgent:
  - **New & various** model feature (software)
  - **GFLOPS / Bandwidth** increase (hardware)

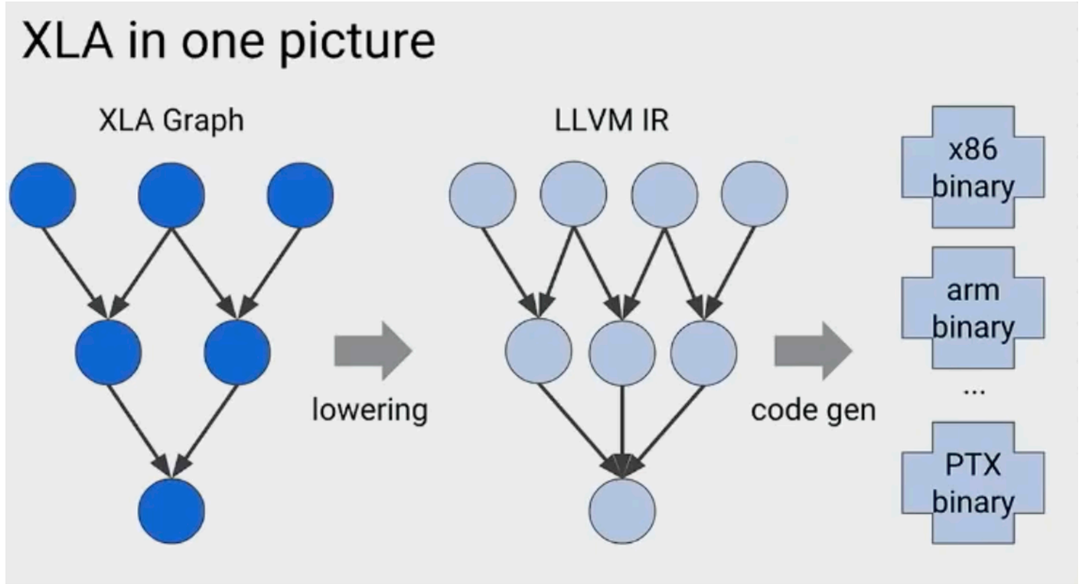
⟶ **Bottleneck: memory**



Introduction

Overhead of memory-intensive OP & compilers

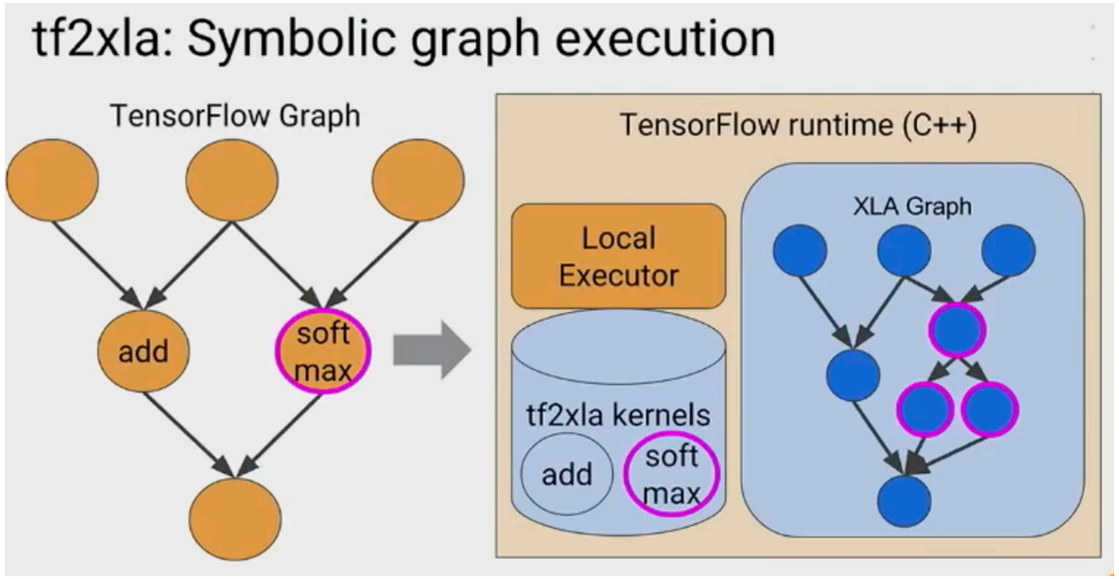
- Overhead of memory-intensive op
  - intensive **off-chip memory access**
  - severe CPU-GPU **context switch**
  - **framework scheduling cost** due to large amount of kernels required to be **launched and executed**
- State-of-art compilers
  - **TensorFlow XLA (JIT & AOT)**
  - **tvm**



Introduction

Overhead of memory-intensive OP & compilers

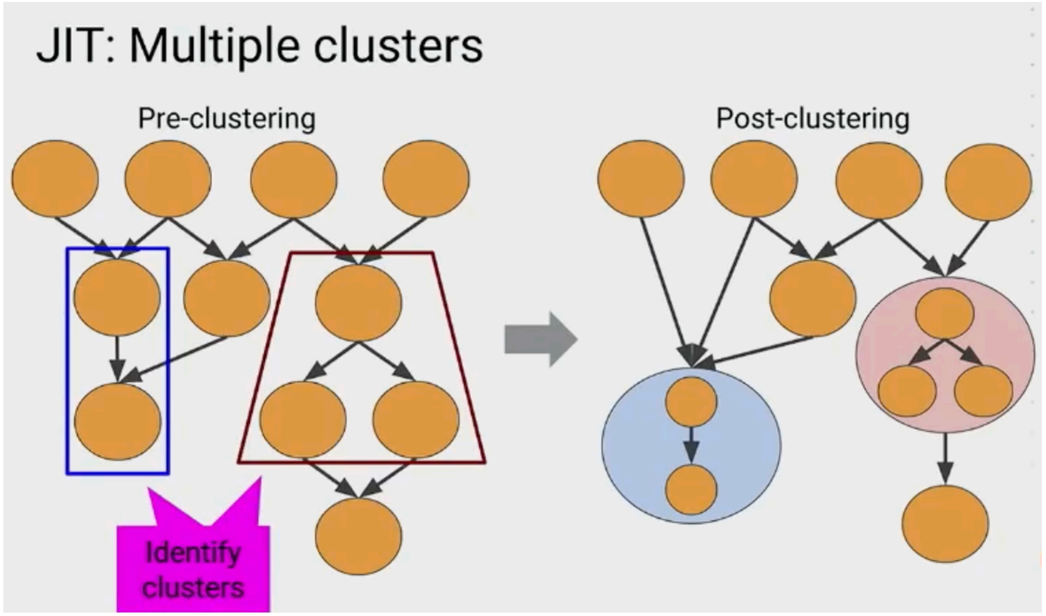
- Overhead of memory-intensive op
  - intensive **off-chip memory access**
  - severe CPU-GPU **context switch**
  - **framework scheduling cost** due to large amount of kernels required to be **launched and executed**
- State-of-art compilers
  - **TensorFlow XLA (JIT & AOT)**
  - **tvm**



Introduction

Overhead of memory-intensive OP & compilers

- Overhead of memory-intensive op
  - intensive **off-chip memory access**
  - severe CPU-GPU **context switch**
  - **framework scheduling cost** due to large amount of kernels required to be **launched and executed**
- State-of-art compilers
  - **TensorFlow XLA (JIT & AOT)**
  - **tvm**



Not good enough

Introduction

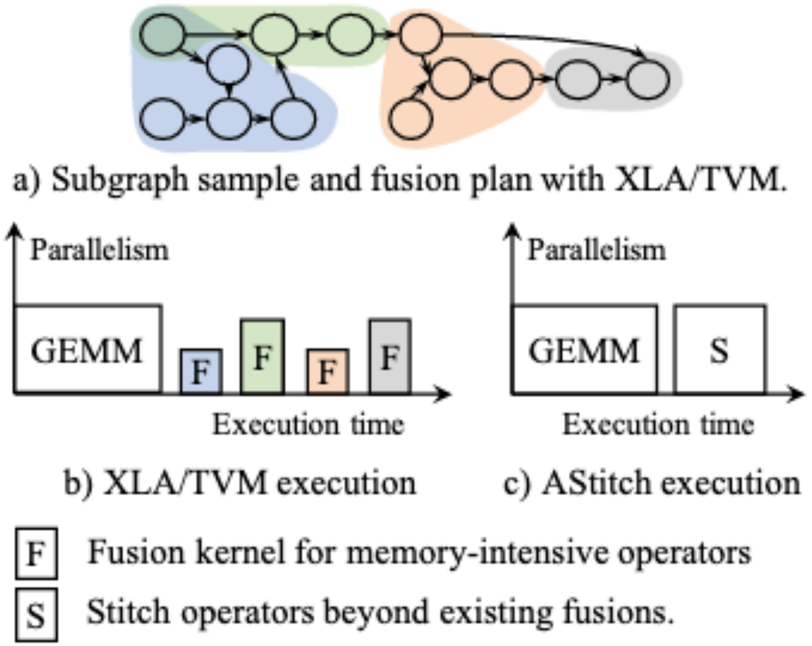
Why current compliers are not good enough?

- Can’t balance between complex **two-level dependencies** and JIT demand
  - many fusion (but too many redundant computation)
  - few fusion (but too many kernels)
- **irregular tensor shapes** often lead to **poor parallelism control** and severe performance issues

Larger fusion scope

Higher parallelism

Lower overhead



Background

Essential Memory-Intensive Ops in Current Models

- Two types (cover the majority of memory-intensive ops)
  - **Element-wise op** (sorted base on **computation strength**)
    - Lighter (e.g. add, sub)
    - Heavy (e.g. tanh, power, log)
  - **Reduce op** (sorted base on **continuity of memory addresses**)
    - Row-reduce ✓
    - Column-reduce ✗
- high frequency of **reduce and broadcast**

tensor shapes between operators become increasingly diverse

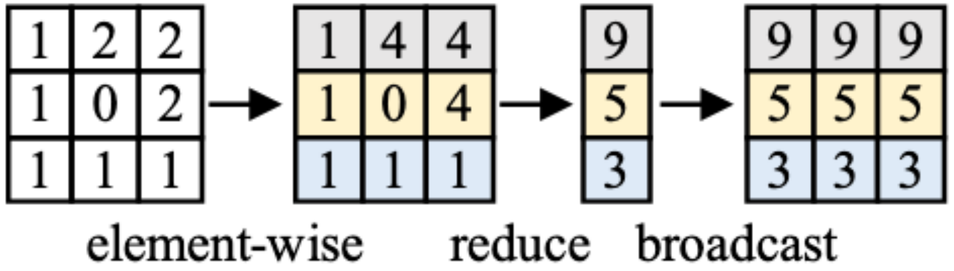
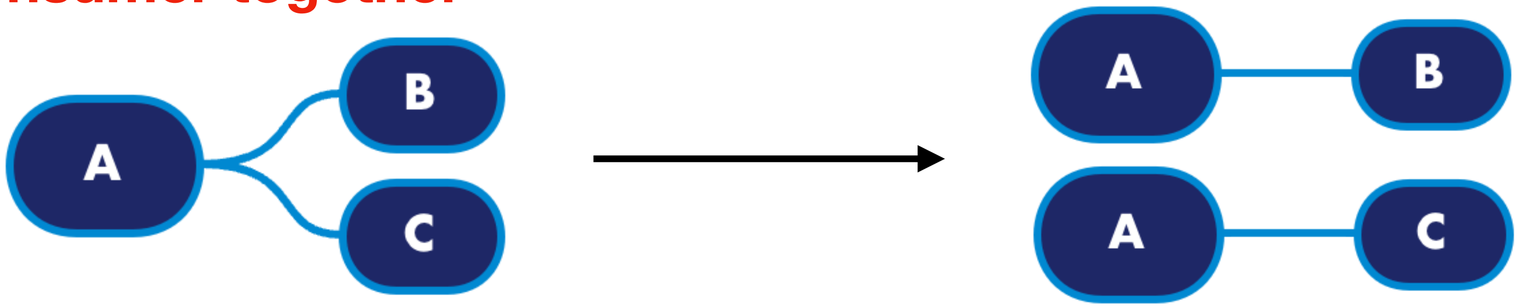


Figure 3: Typical memory-intensive operations.

Background

XLA & tvm ’s problem

- ML compilers often make fusion decisions according to whether they can generate efficient code
- XLA & tvm ’s code generator
- **deal with all data dependencies with per-element input inline to merge producer with consumer together**



Can’t make fusion in larger scale

AStitch's Challenges & Solution 1

Can't balance between complex two-level dependencies and JIT demand

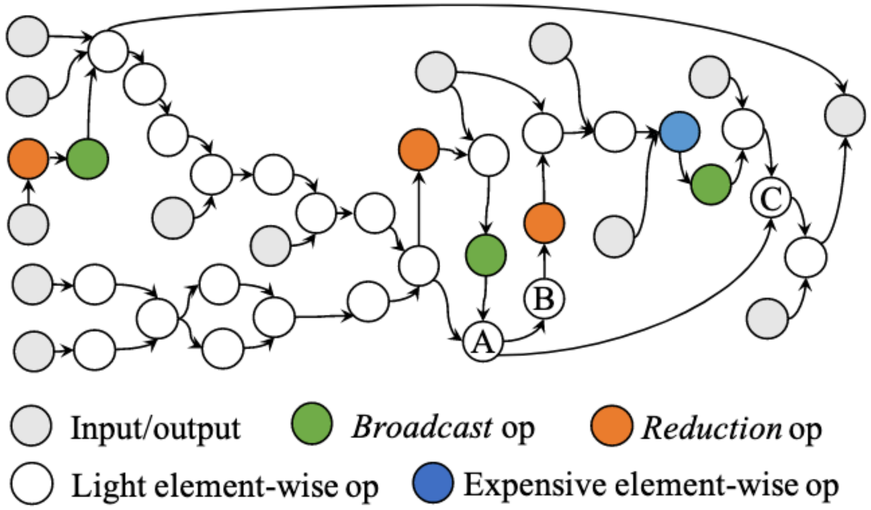
- **two-level dependencies**
  - Operator-level dependency (complexity of model)
  - **Element-level dependency** (feq of reduce & boardcast)

A key observation

XLA & tvm cannot perform efficient fusion under such two-level dependencies.

Two patterns need to be solved :

- **reduce ops** with its consumers
- costly element-wise ops followed by **broadcast ops**



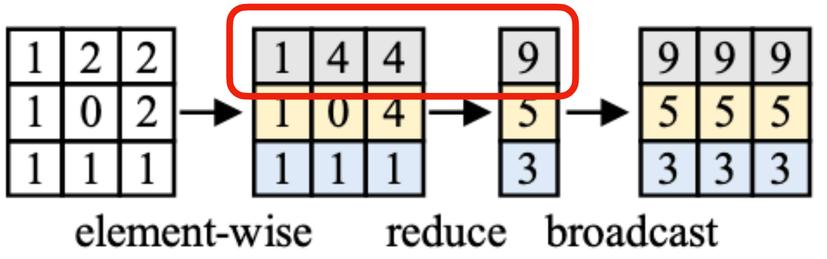


Figure 3: Typical memory-intensive operations.

AStitch's Challenges & Solution 1

Two patterns — fuse or not ?

- **Choice 1 : Fuse** — Heavy redundant computation
  - **Can't communicates intermediate results between threads** (**register only**)
- **Choice 2 : Skip** — More kernels generated for execution.
  - XLA & tvm's choice

How to reuse data efficiently

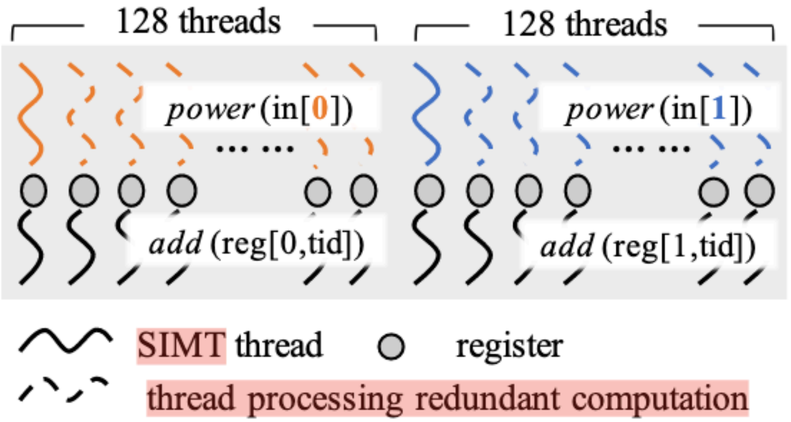


Figure 5: Redundant computation in TVM when attempting to fuse *power<2>* - *broadcast<2,128>* - *add<2,128>*[1] automatically with compiler. Different colors for *power* represent threads that process different elements in the input tensor.

AStitch's stitch scheme

Abstract four types of scheme to cover all dependencies

- abstract four types of stitching schemes, covering all the scenarios of dependencies from the **joint consideration of dependency, memory hierarchy and parallelism**

Table 1: Stitching scheme abstraction with joint consideration.

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

New

AStitch's Challenges & Solution 1

Hierarchical Data Reuse

- **1. Two-level dependencies —> two-level data reuse**
  - **Element-level data reuse**
  - **Operator-level data reuse**
- 2. Kernel Form
  - Less kernel launch
  - More efficient data reuse
- 3. Global Barrier

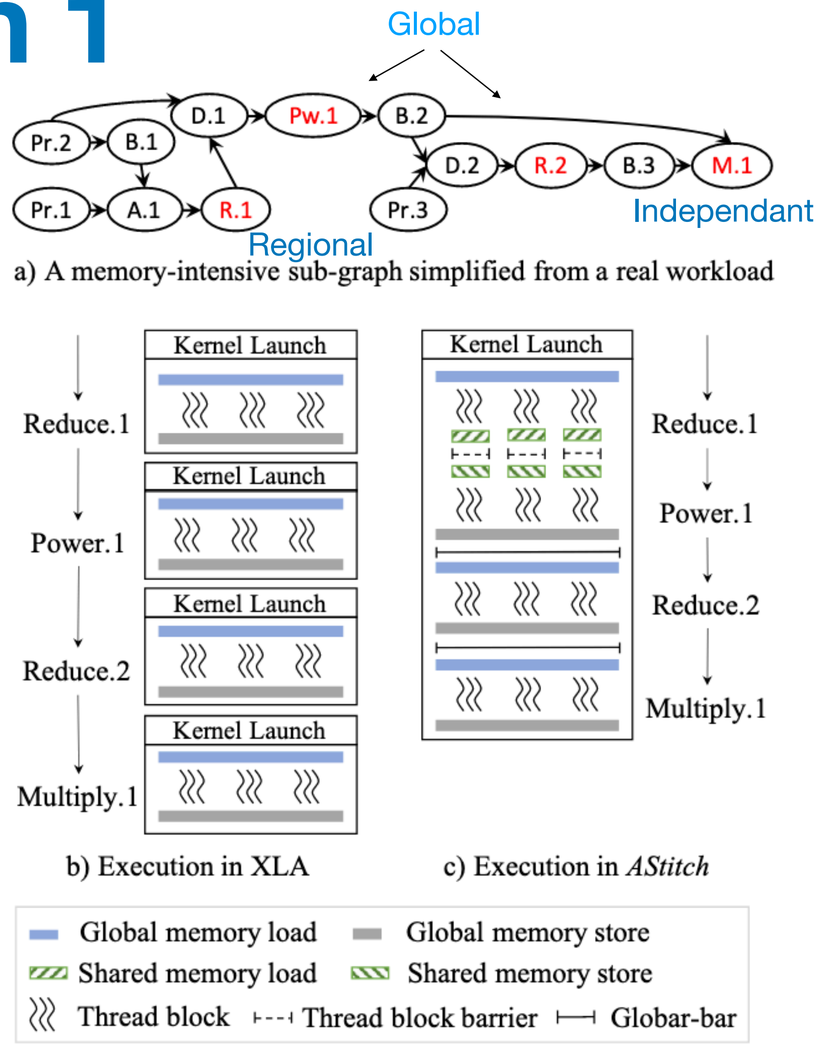


Figure 7: Execution scheme of a memory-intensive sub-graph. *AStitch* reduces kernel launches and off-chip memory access with hierarchical data reuse. Pr: parameter. A: add. B: broadcast. R: reduce. D: divide. Pw: power. M: multiply.

AStitch's Challenges & Solution 2

Irregular Tensor Shapes in Real-World Production Workloads

- Irregular tensor shapes ->
  - **Too many small** partions
  - **Too few large** partions

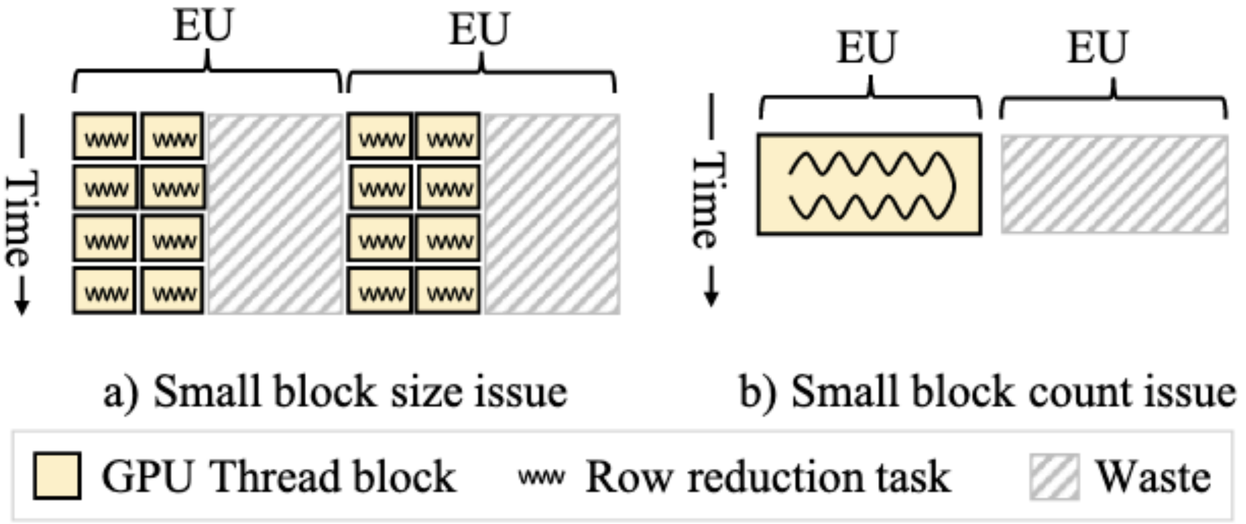


Figure 6: Typical poor parallelism issues in existing works.

How to automatically generate thread mappings ?

<750000,32> —> <750000>

AStitch's Challenges & Solution 2

Adaptive Thread Mapping

- Thread mapping
  - Task **packing (2-dimensional)** — too many small partitions
  - Task **splitting** — too many large partitions

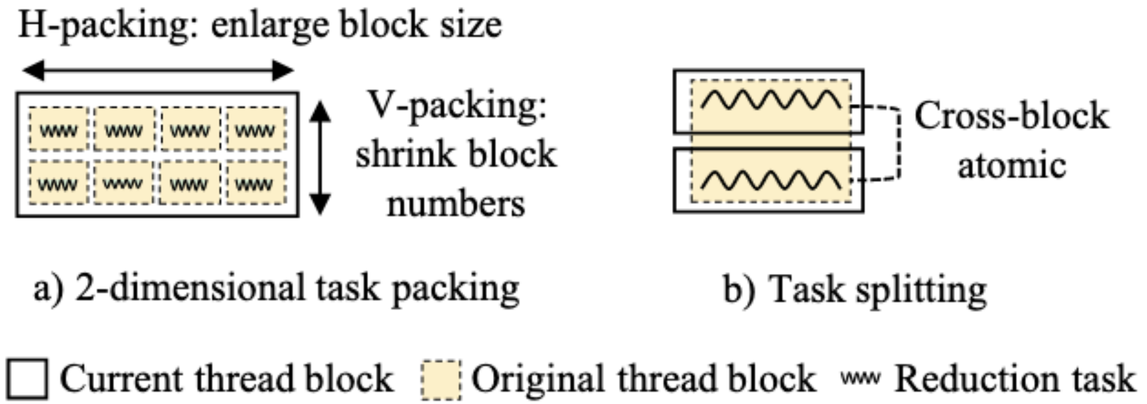
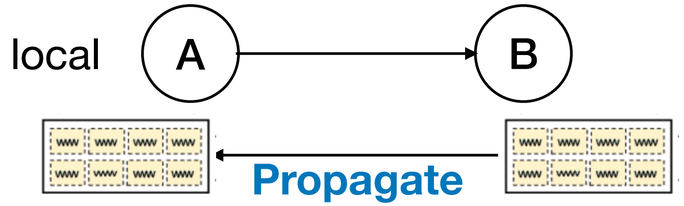


Figure 8: Task packing and splitting optimization for row-reduction on GPU. Task mappings in existing work are in Fig.6.

Automatic Compiler Optimization Design

Observations

- Observation-A : if present op is of *local* scheme, the thread mapping can be determined by **propagating from its consumer**'s **thread mapping**



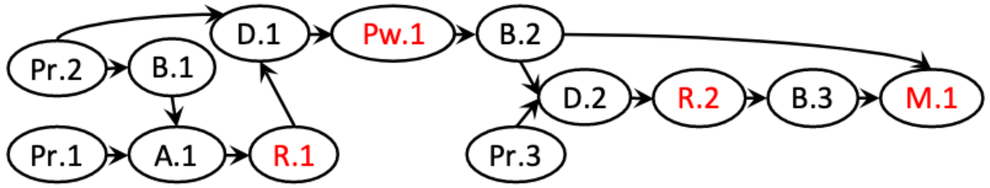
- Observation-B : **patterns of reduce and expensive element-wise ops followed by broadcast ops** must be *regional / global* scheme

Must be regional / global

Automatic Compiler Optimization Design

Step by step

- **Step 1 : dominant identification and op grouping.**
  - 1.1 Identify several **candidates** for becoming dominant ops (two pattern & output op)
  - 1.2 Identify the final ones with **dominant merging (sub-dominant & dominant op (reduce op))**
  - 1.3 op grouping via *local* op **(sub-dominant op as boundary)**



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

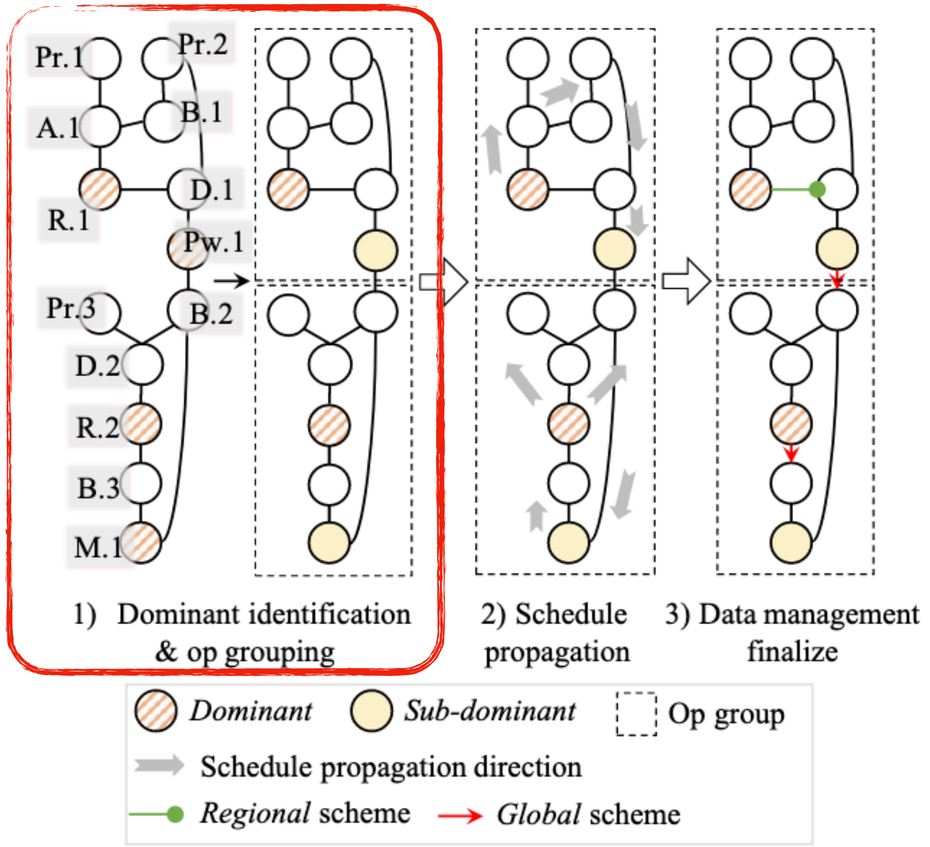
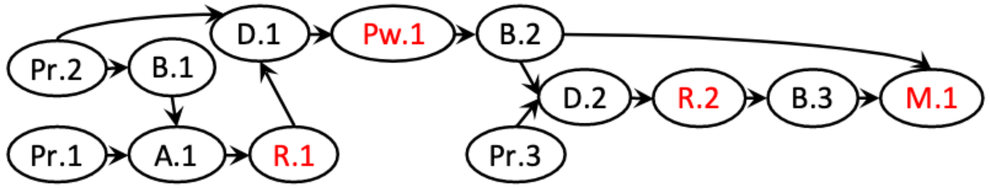


Figure 9: Schedule propagation and data management planning for the graph topology shown in Figure.7-(a).

Automatic Compiler Optimization Design

Step by step

- **Step 2 : adaptive thread mapping and schedule propagation**
  - 2.1 generates the parallel code for each **dominant op** according to **adaptive thread mapping**
  - 2.2 **propagates the thread mapping** schedule within group



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

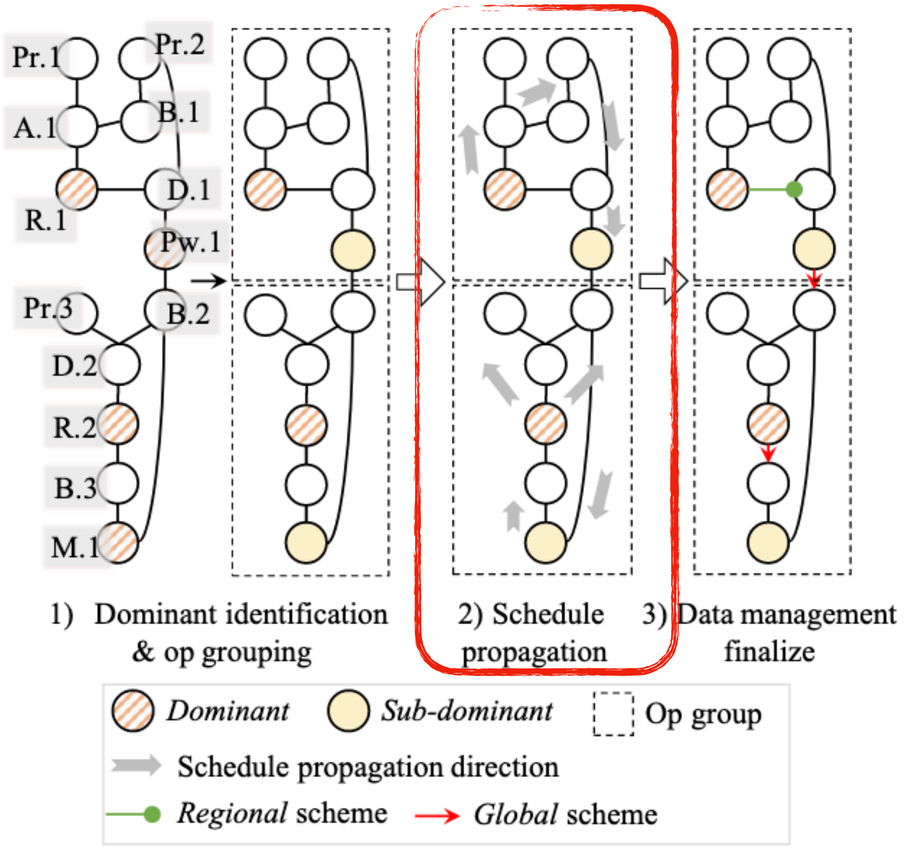


Figure 9: Schedule propagation and data management planning for the graph topology shown in Figure.7-(a).

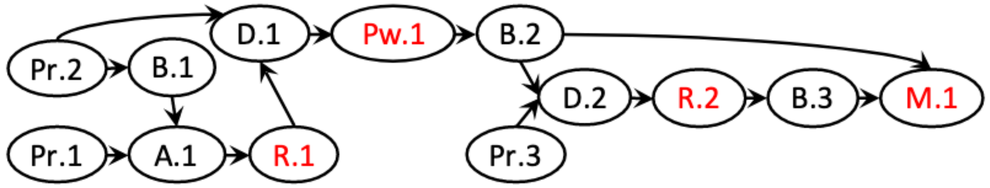
Automatic Compiler Optimization Design

Step by step

- **Step 3 : Finalize stitching schemes for (sub) dominant ops (must be regional / global)**
  - 3.1 identify the stitching scheme between regional and global.**(Passive block-locality checking)**

Thread block locality ?

✓ **Regional** ✗ **Global**



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

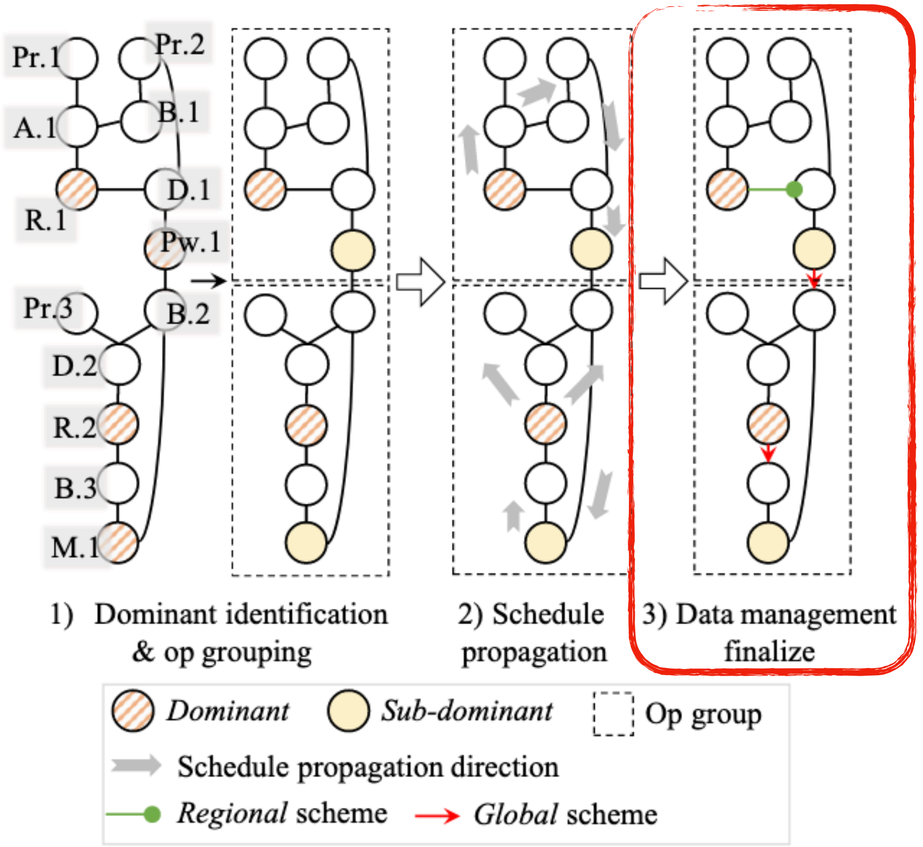
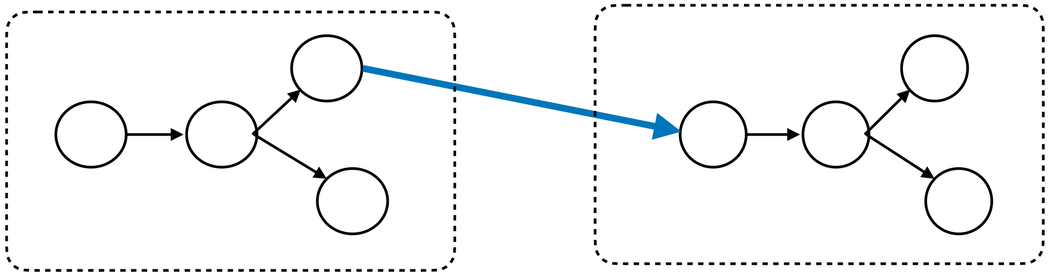


Figure 9: Schedule propagation and data management planning for the graph topology shown in Figure.7-(a).

Automatic Compiler Optimization Design

Step by step

- **Step 3 : Finalize stitching schemes for (sub) dominant ops (must be regional / global)**
  - 3.2 tune between groups for better data reuse **(Proactive block-locality adaptation)**



Adjust thread mapping

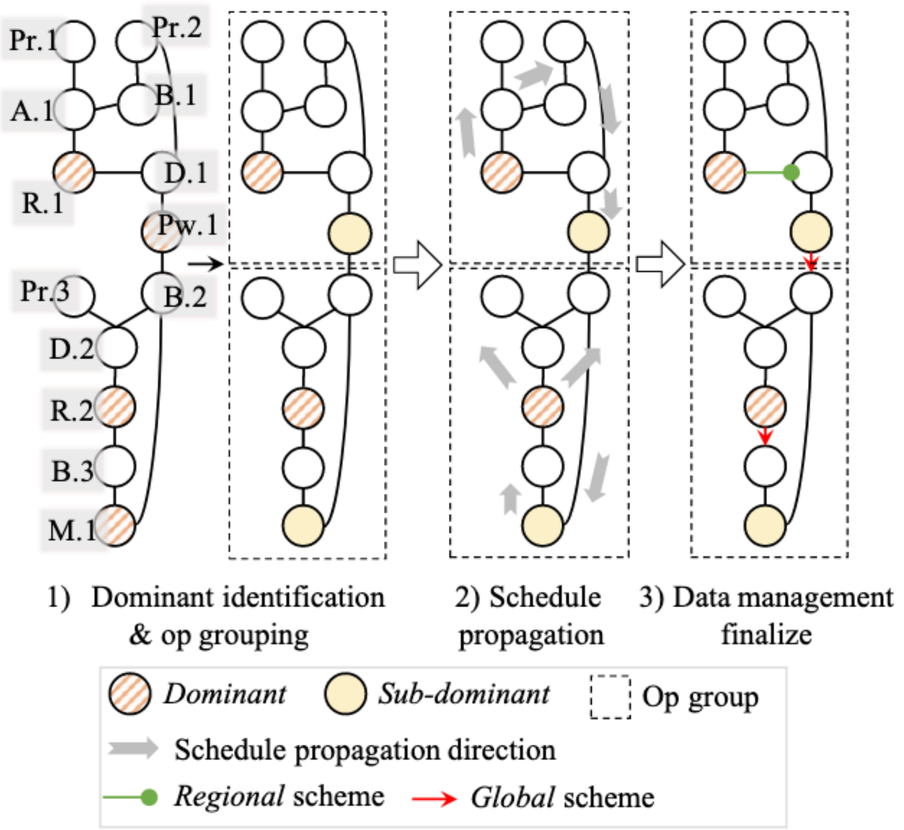
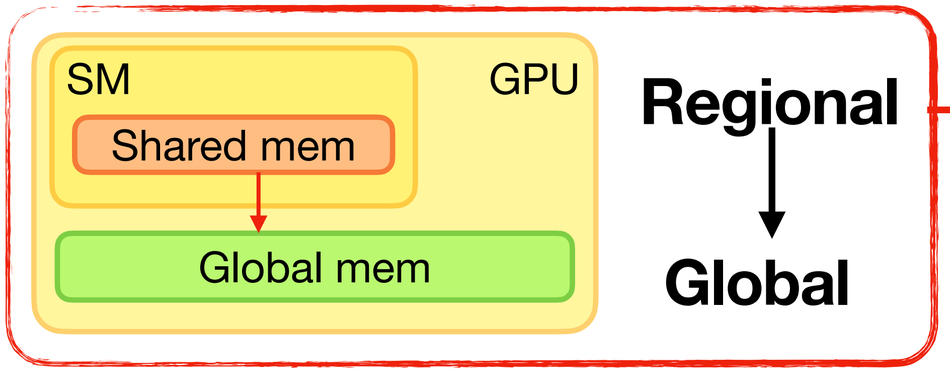


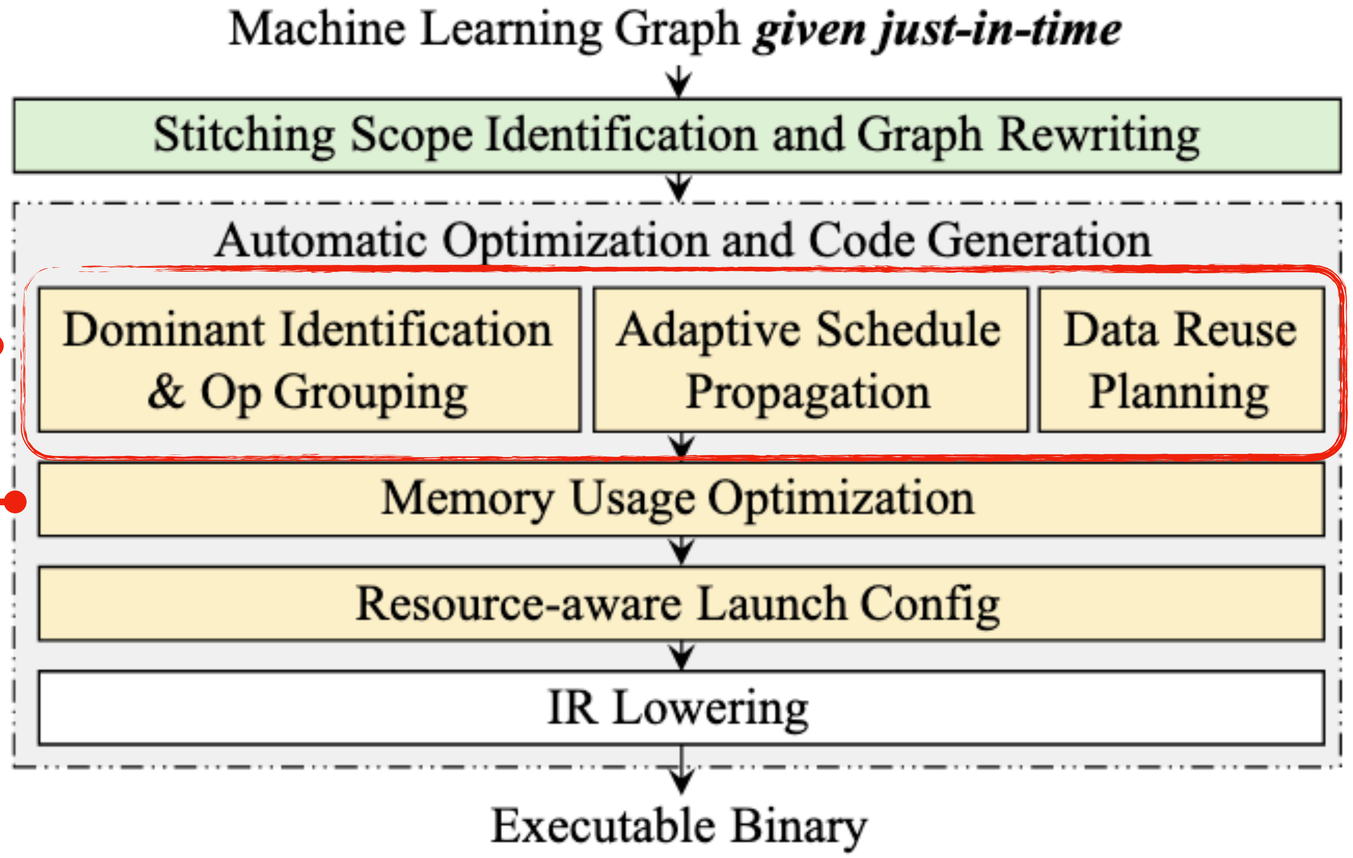
Figure 9: Schedule propagation and data management planning for the graph topology shown in Figure.7-(a).

Implementation

Overview of AStitch

- Adaptive Thread Mapping
- Hierarchical Data Reuse





Evaluation

Workload & baseline

device	NVIDIA V100
graphics memory	16G
CUDA toolkit	10.0
cuDNN	7.6

Workload

Table 2: Workloads for evaluation.

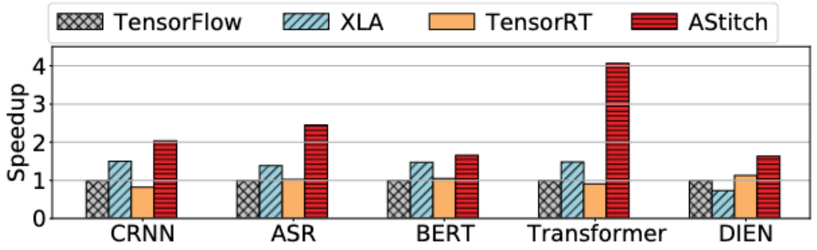
Model	Field	Train batch-size	Infer batch-size
CRNN	Images	-	1
ASR	Speech	-	1
BERT	NLP	12	200
Transformer	NLP	4,096	1
DIEN	Recommendation	256	256

Baseline

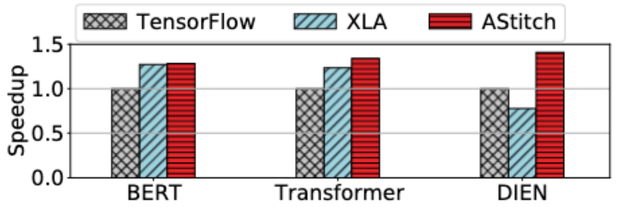
- TensorFlow
- TensorFlow XLA
- TensorRT (for inference)
- TVM Ansor (lack of support)

Evaluation

Overall Results



(a) Speedup of inference tasks.



(b) Speedup of training tasks.

Figure 11: End-to-end performance speedup.

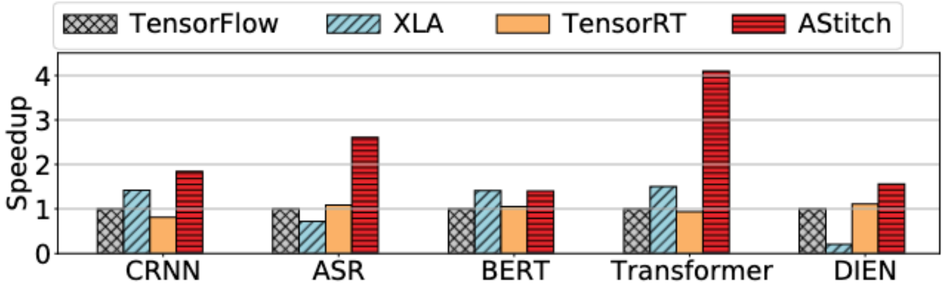


Figure 12: Inference speedup, for which baselines and *AStitch* are all with AMP optimization.

good combination with **AMP**

Speed up in inference & training

Compared with XLA

2.37x & 1.3x

Evaluation

Performance Breakdown

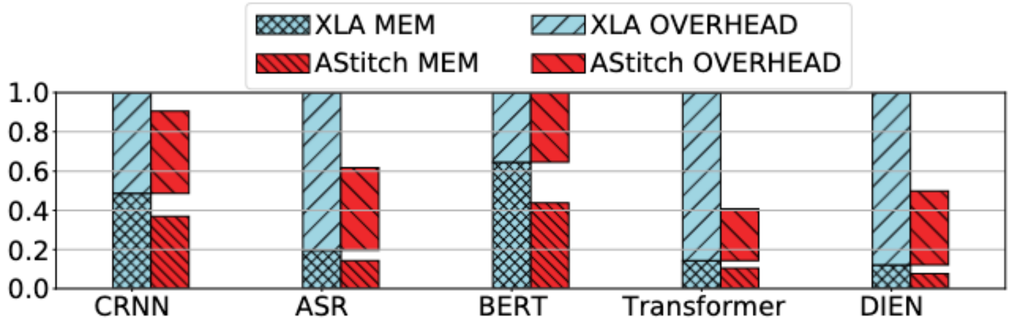


Figure 13: Performance breakdown, without showing the time of compute-intensive ops.

execution time : **memory-intensive op execution (MEM)**, **compute-intensive op execution** and **non-computation overhead (OVERHEAD)**

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

Kernel Call **Decrements**

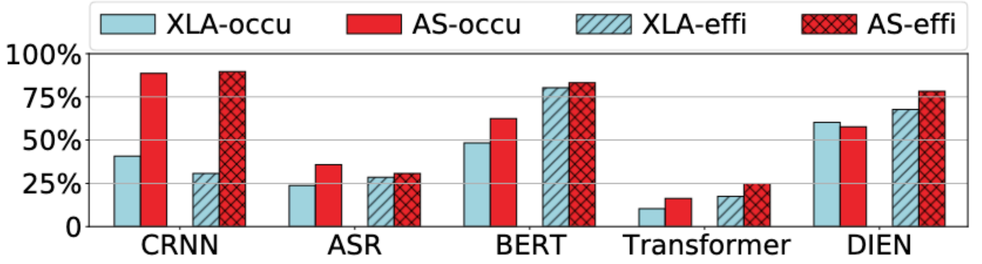


Figure 14: Average parallelism of top 80% memory-intensive computations. AS: *AStitch*. Occu: occupancy. Effi: SM-efficiency.

Parallelism & hardware utilization Increment

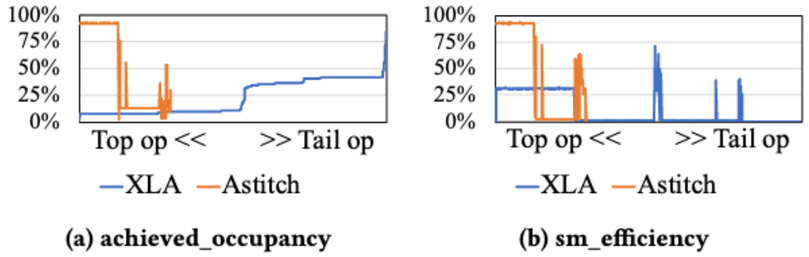
Evaluation

A Comprehensive Case Study

Table 4: Ablation study for *CRNN*.

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

Ablation Study



(a) achieved_occupancy **(b) sm_efficiency**

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.

Parallelism & hardware utilization

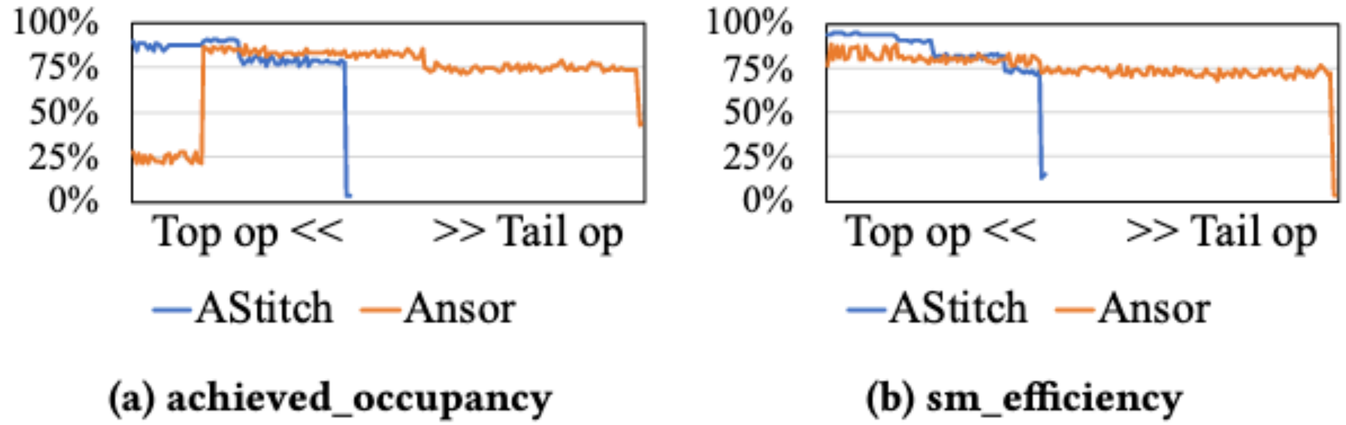
Table 5: Total performance counters of all memory-intensive ops in *CRNN*. DR_transactions: dram_read_transactions. DW_transactions: dram_write_transactions

	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

Reduced off-chip memory traffic & Reduced instructions

Evaluation

AStitch vs tvm (Ansor)



(a) achieved_occupancy

(b) sm_efficiency

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

Conclusion

- The **first work** to thoroughly investigate how to **optimize memory-intensive ML computations** from a **joint aspect** of dependency characteristics, memory hierarchy and parallelism
- Solve two major performance issues of memory-intensive ML computations:
  - **inefficient fusion (two-level dependencies) -> hierarchical data reuse**
  - **inputs with irregular tensor shapes -> adaptive thread mapping**
- **JIT mode** for any given arbitrary machine learning model **for both training and inference**
- Outperform compared with state-of-art compilers **for Memory-Intensive ML**

FusionStitching: Boosting Memory Intensive Computations for Deep Learning Workloads
https://github.com/alibaba/BladeDISC

Thanks

2023-3-30

Presented by Guangtong Li