Optimizing Deep Learning Inference Efficiency through Block Dependency Analysis

Zhanyuan Di, Leping Wang, En Shao, Zhaojia Ma, Ziyi Ren, Feng Hua, Lixian Ma, Jie Zhao, Guangming Tan, Ninghui Sun

SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Hunan University

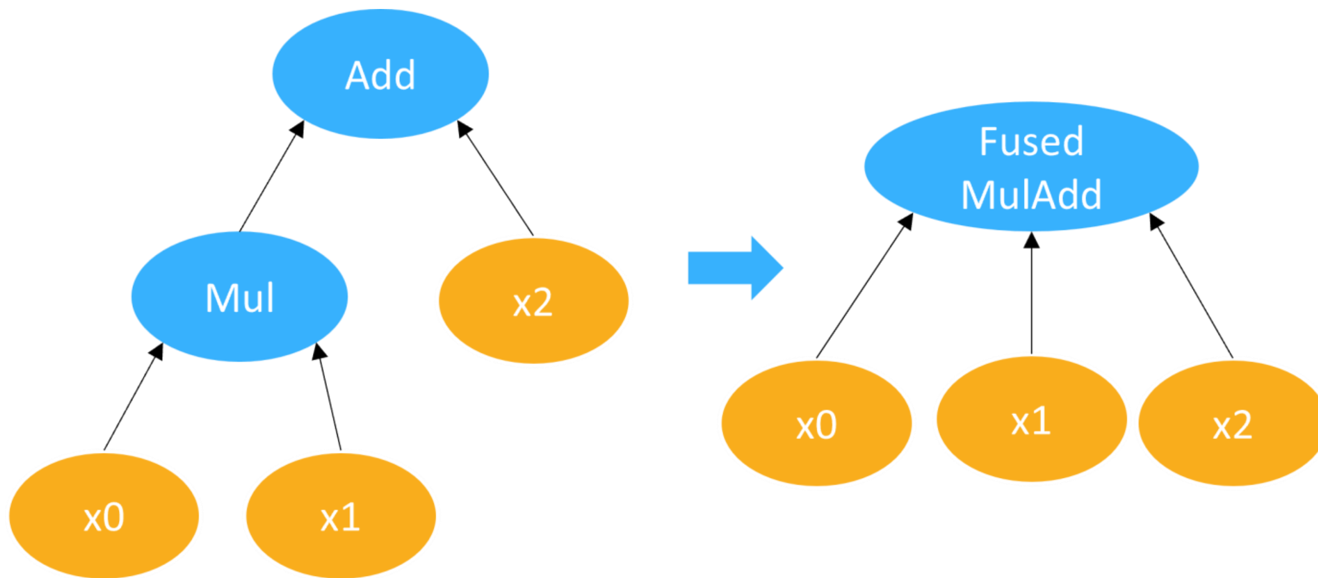
中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

中国科学院大学
University of Chinese Academy of Sciences

湖南大学
HUNAN UNIVERSITY

Background - Operator Fusion

Vectorized operations across multiple operators can be fused into a single vectorized operation.
This improves **GPU utilization**, reduces **kernel launch overhead**, and minimizes **memory access costs**.

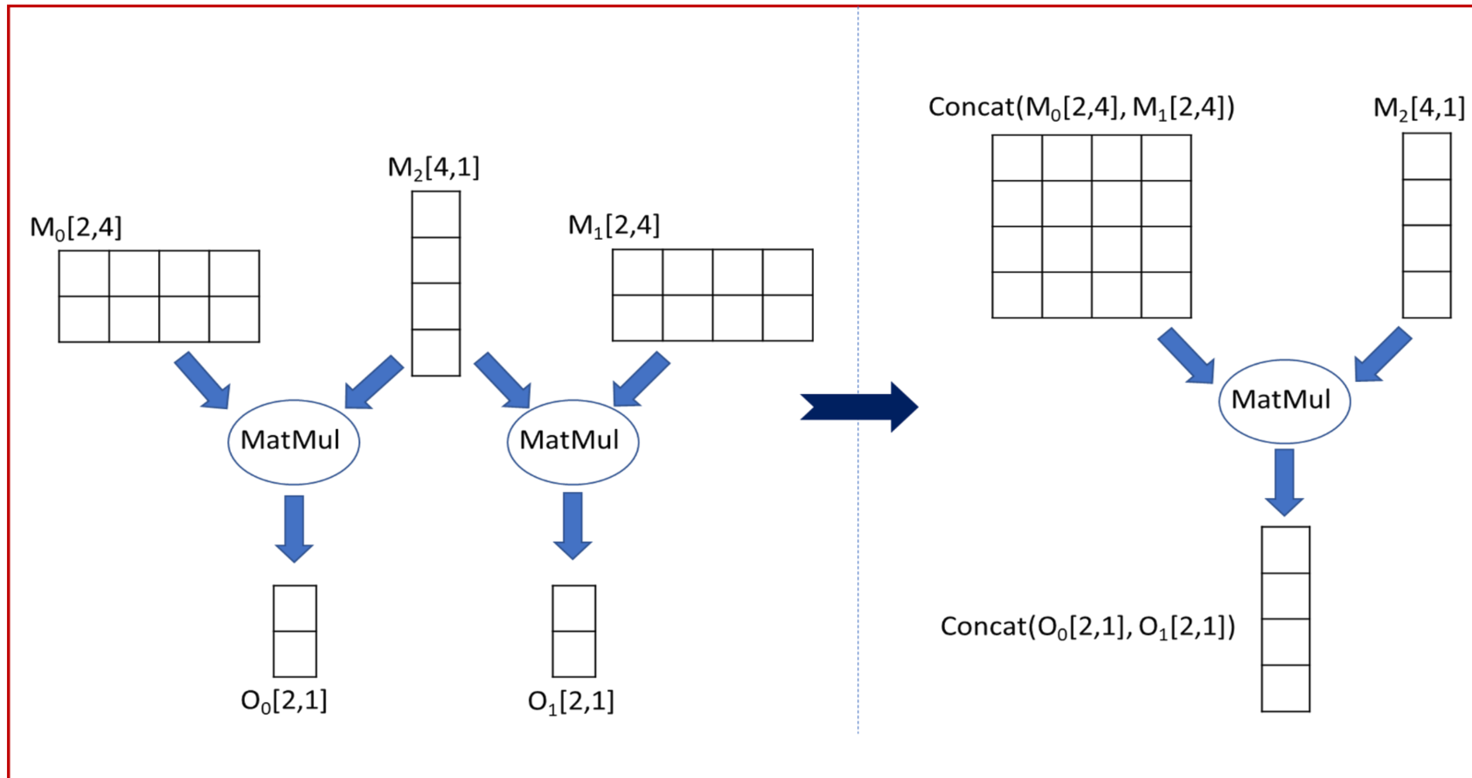


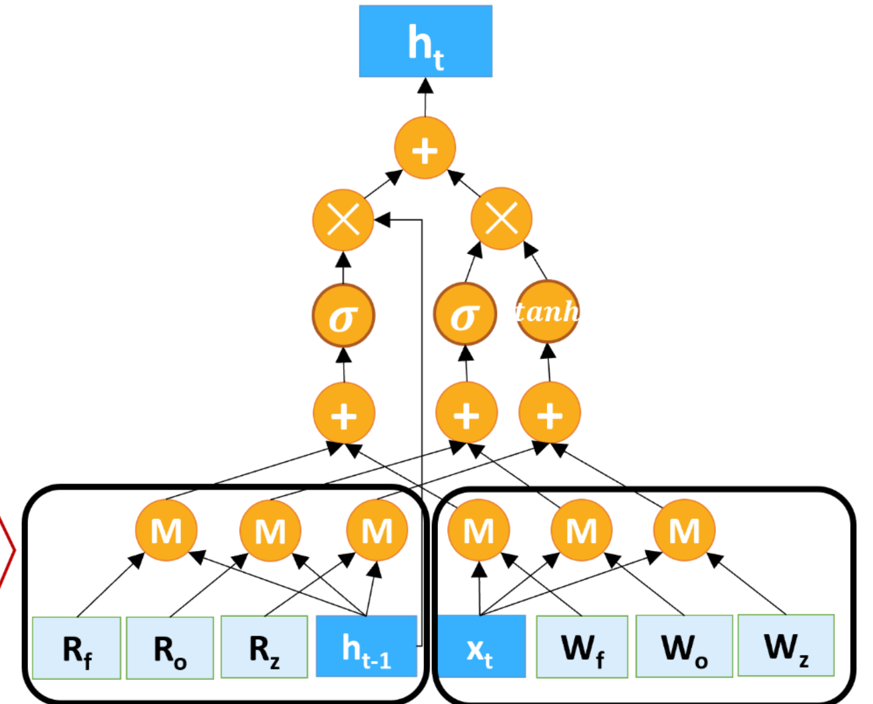

```
__global__ mul(float *x0, float *x1, float *y){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  y[idx] = x0[idx] * x1[idx];
}
__global__ add(float *x0, float *x1, float *y){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  y[idx] = x0[idx] + x1[idx];
}
```

```
__global__ fused_muladd(float *x0, float *x1, float *x2, float *y){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  y[idx] = x0[idx] * x1[idx] + x2[idx];
}
```

Background - Operator Fusion (Sibling)

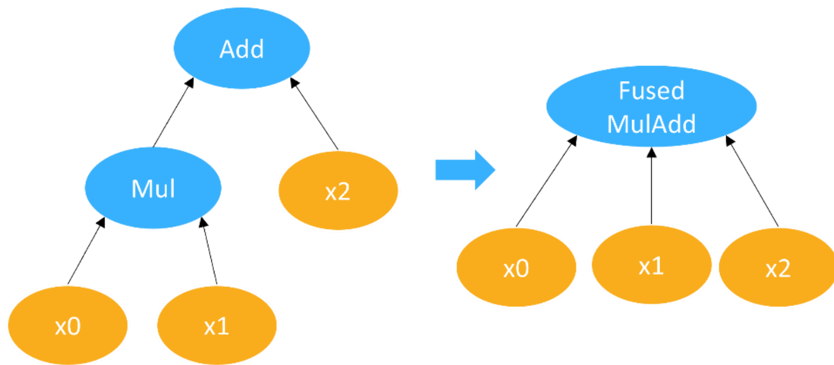
By merging input tensors into a larger tensor, identical operators can be fused into a larger operator, effectively enhancing **hardware parallelism**.



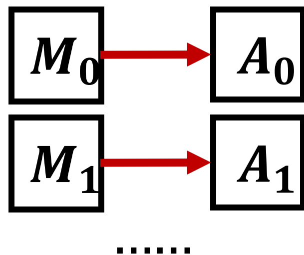


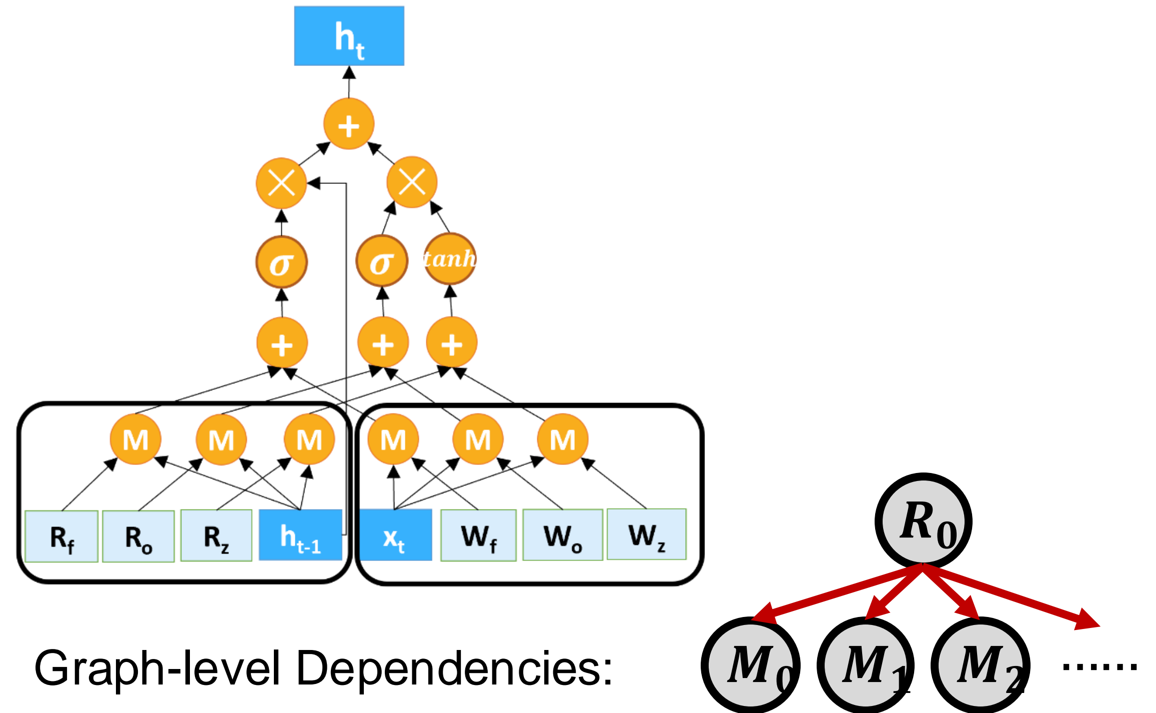
Background - Operator Fusion

ML compilers make fusion decisions (e.g., pattern match) according to whether they can generate efficient code.
For example, TVM/XLA’s code generators deal with all data dependencies with **per-element input inline** to merge producer with consumer together.



Element-level Dependencies:





What if it's non-one-to-one?

What if it's non-sibling?

Motivation

Kernel fusion improves efficiency but struggles with complex dependencies.

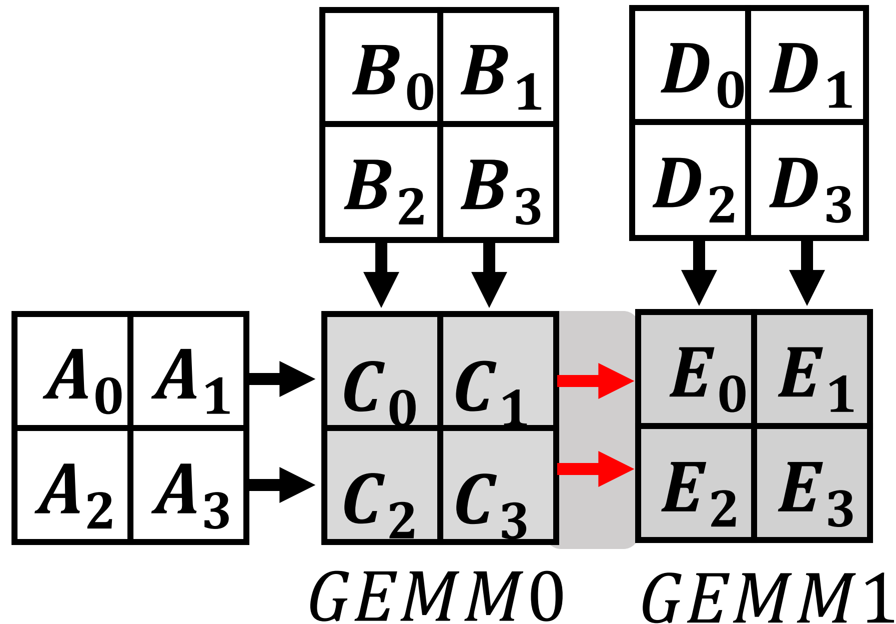
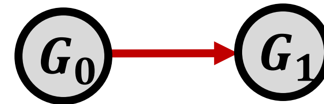
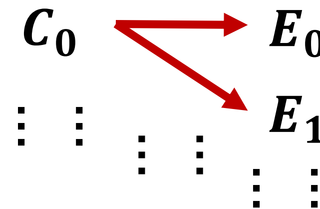


Illustration of two kinds of dependency analysis:

- Operator-Level View:



- Element-Level View:



After inlining:

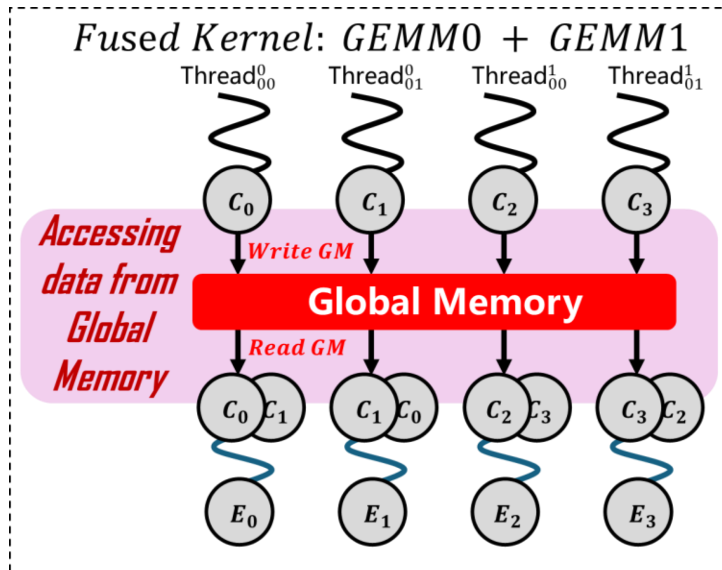
$$E_0 = C_0 \cdot D_0 + C_1 \cdot D_2$$

$$E_0 = \underline{(A_0 \cdot B_0 + A_1 \cdot B_2)} \cdot D_0 + \underline{(A_0 \cdot B_1 + A_1 \cdot B_3)} \cdot D_2$$

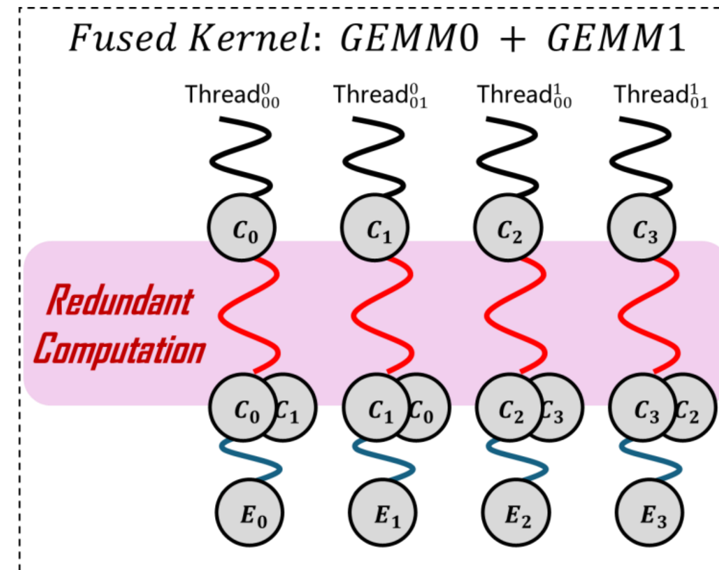
Also inlined by E1

Motivation - What happens after fusion?

Source #1: Inefficient Data Access and Redundant Computation



Typical inefficient data access in AStitch when fusing two GEMMs using global memory



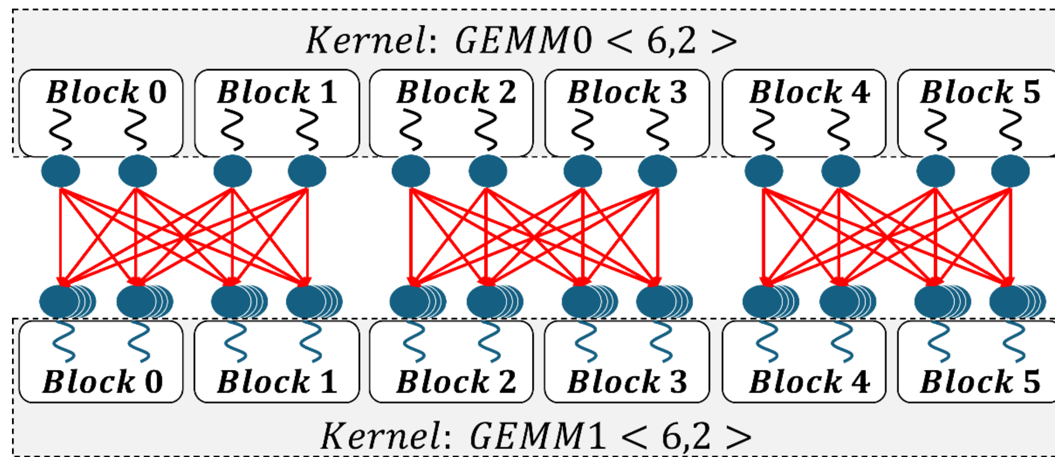
Avoiding low-bandwidth memory access is possible but incurs unacceptable costs due to redundant computation

Two ways to handle data dependency after operator fusion:

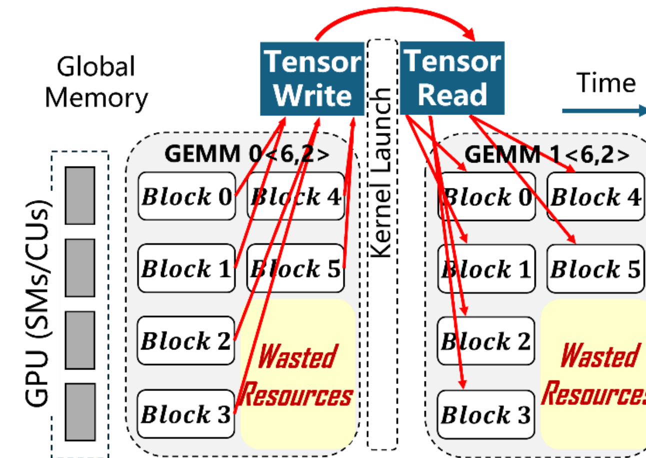
- (1) using global memory
- (2) using redundant computation

Motivation

Source #2: Missed Opportunities for Improving Parallelism



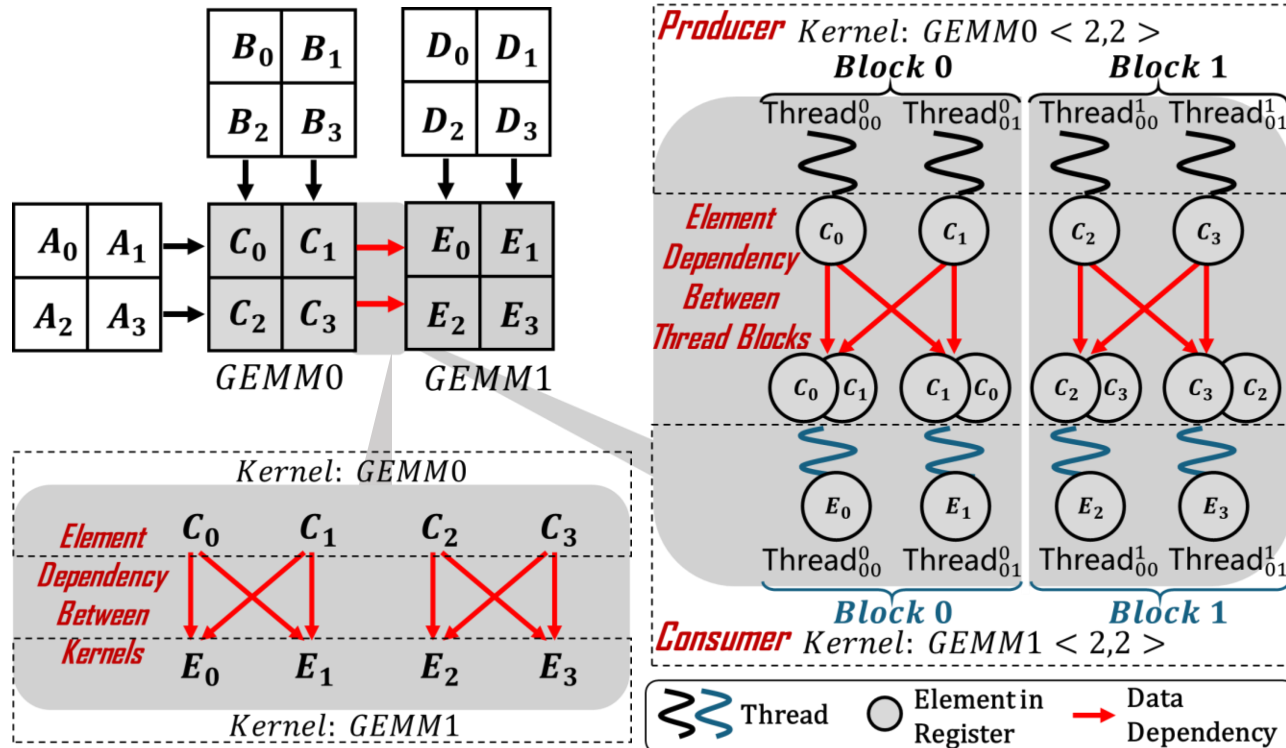
(a) Data Dependency between two sequentially executed GEMMs <6,2>



(b) Wasted resources in Welder and TVM during kernel execution

Typical idle resource issues arise from the tail effect in the execution of compute-intensive operators.

Motivation - Static vs Block Dependency



Static Data Dependency: analysis only based on data flow

Block Dependency: analysis with additional thread block mapping

Existing solutions: Static analysis approach:

Traditional data dependency analyses are based on (1) the study of tensor expressions and (2) computation graphs.

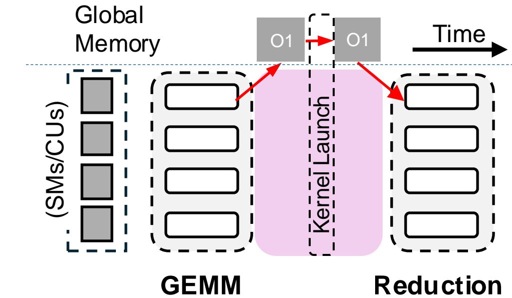
Problem*: Existing static analysis cannot capture block-level dependencies.*

Motivation - Block Dependency Abstraction

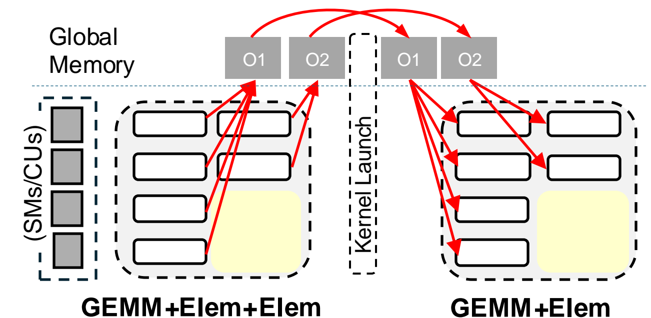
Observations & Potential Optimizations:

- **Source #1 :** One-to-one block dependencies allow fusion with shared memory.
- **Source #2 :** Some blocks in GEMM1 do not depend on all blocks in GEMM0, enabling parallel execution.

These optimization opportunities necessitate block-level dependencies.



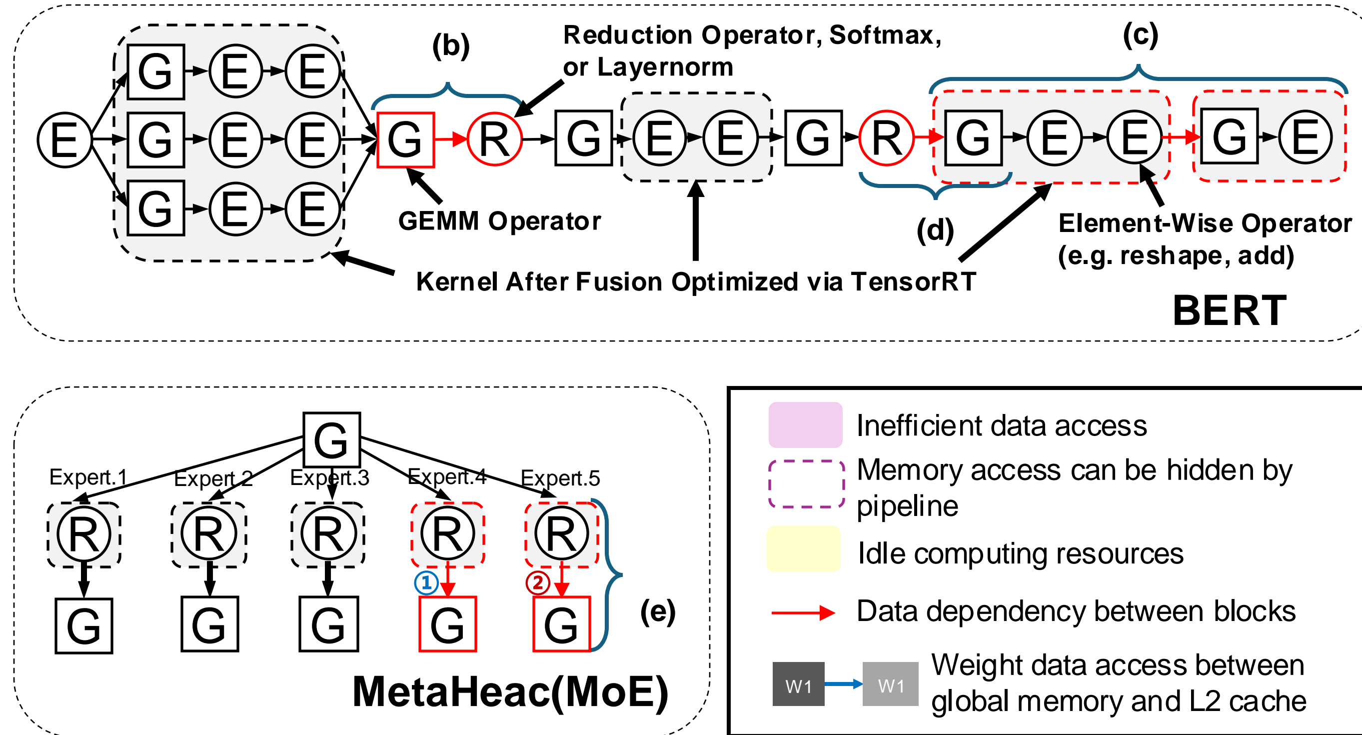
Source #1 Inefficient data access in **One-to-One** Block Dependency



Source #2 Idle computing resources in **Many-to-Many** Block Dependency

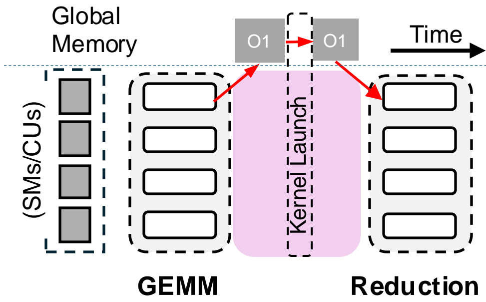
Motivation - Block Dependency Abstraction

Inefficiencies in real-world workloads：

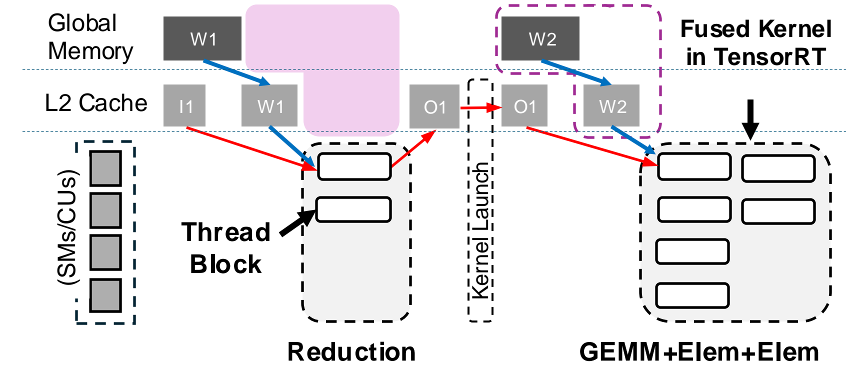


Two subgraphs in BERT and MoE models optimized via TensorRT

Motivation - Block Dependency Abstraction

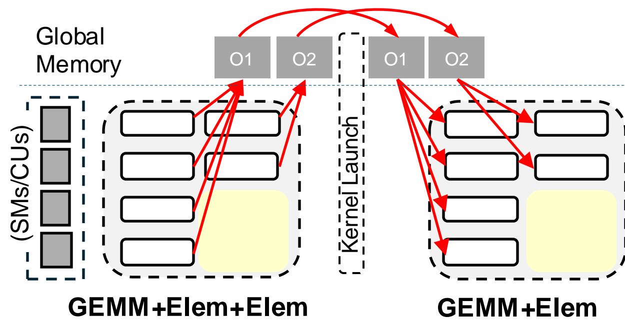


Inefficient data access in **One-to-One** Block Dependency

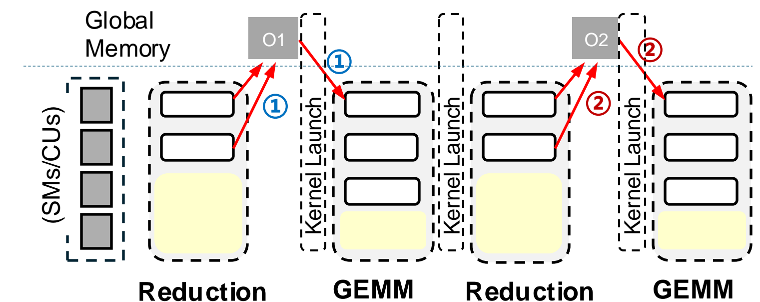


Inefficient data access due to weight access in **Partial Block Independence**

Block-Level Dependency

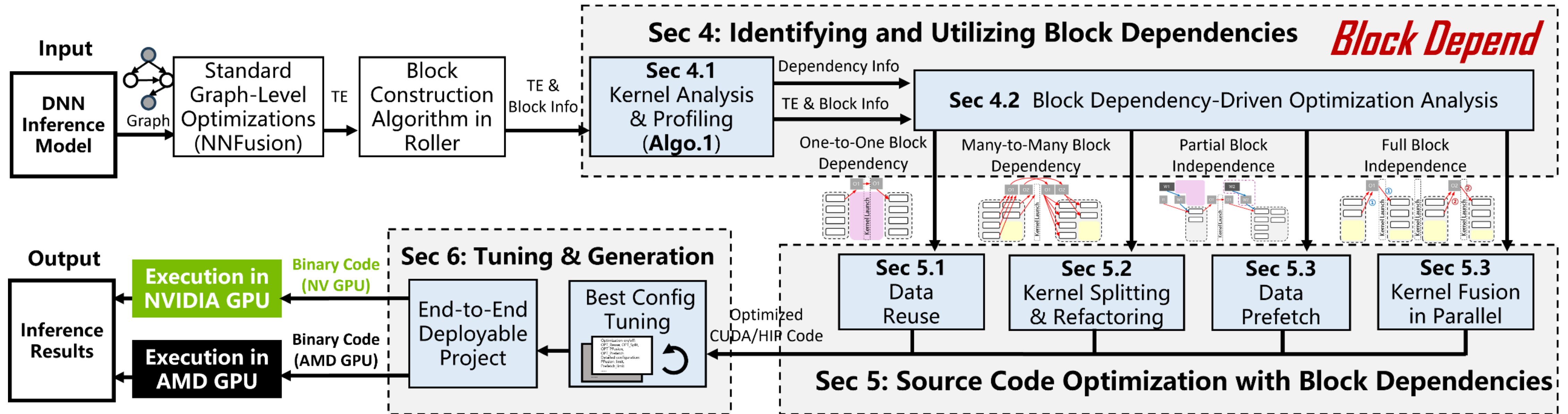


Idle computing resources in **Many-to-Many** Block Dependency



Idle computing resources in **Full Block Independence** (① & ②)

Overview

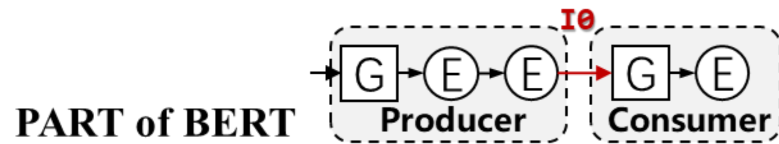


System overview of BlockDepend

- The process begins with **ONNX** Models, which is optimized at the graph level based on static data dependencies using **NNFusion**, and then converted to an intermediate representation (**TE**).
- **TE** is processed by the construction algorithms in **Roller.**

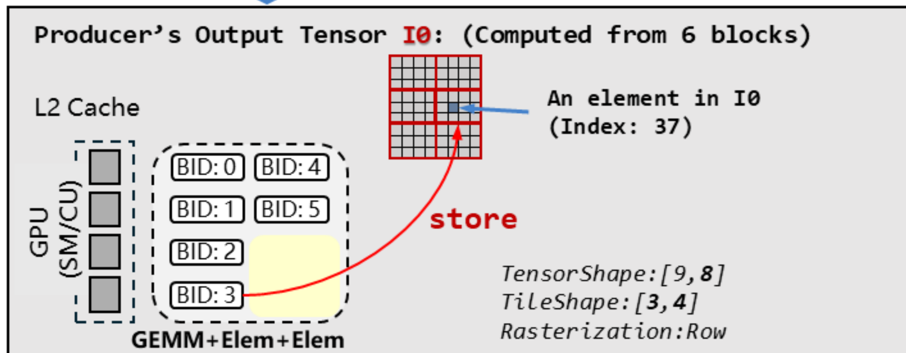
STEP 1 - Identifying Block Dependencies

Stage-1: Obtain mapping(CalculateBlockID) from an element’s index to its producer’s block ID.




```
temp0 = te.compute([9,8], lambda N, M:
te.sum((input0[N, K] * input1[K, M]), axis=[K]))
......
```

Block Construction

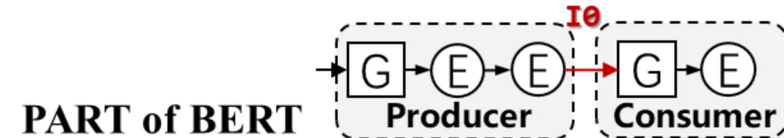


Analysis

```
CalculateBlockID(index) {
  BID = (index // 8) // 3 * (8 // 4) + ((index % 8) // 4)
  return BID
}
* mapping an element's index to its producer's block ID
```

CalculateBlockID(37) = 3, means the element (index=37) depends on the producer block(ID=3).

Stage-2: A consumer block uses CalculateBlockID to determine the dependent producer block ID based on the element’s index it accesses.



Block Construction & Code Generation

```
# Consumer Kernel (I0 is Producer’s output)
GEMMElemProgram(…):
 …… # gridDim-6
 for K_0 in range 2:
  syncthreads
# CUR_BID: Current Consumer Block ID
  # DEP_BID: Dependent Producer Block ID
  for ax0 in range 4:
   SI0[SI0_INDEX] = I0[I0_INDEX]
   DEP_BID = CalculateBlockID(I0_INDEX)
   RecordDEP(DEP_BID, CUR_BID, dep_buffer)
  for ax0 in range 4:
   SW0[SW0_INDEX] = W0[W0_INDEX]
  ……
```

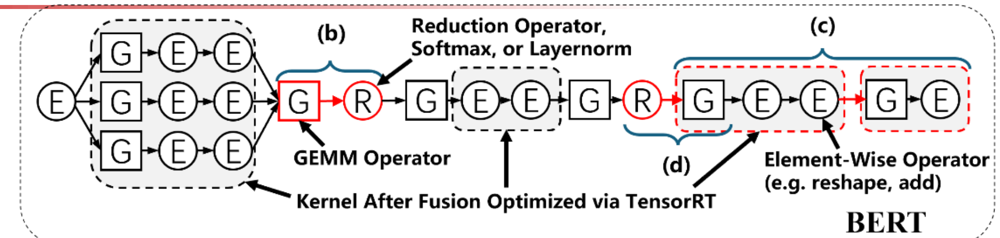
Format dep_buffer

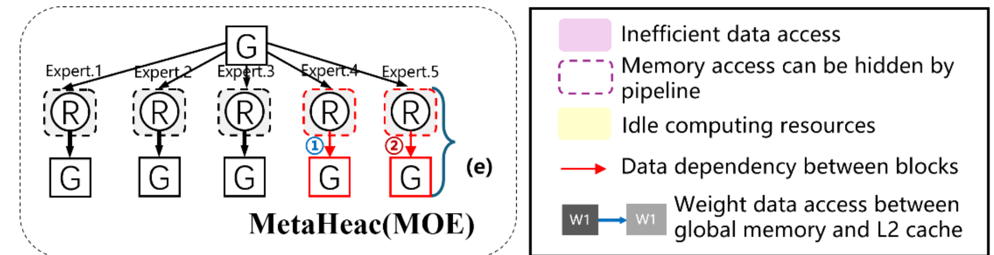
[Producer Blockid List] → Consumer Blockid
[0,1]→ 0, [0,1]→ 1
[2,3]→ 2, [2,3]→ 3
[4,5]→ 4, [4,5]→ 5

STEP 2 - Dependency-Driven Optimization Analysis

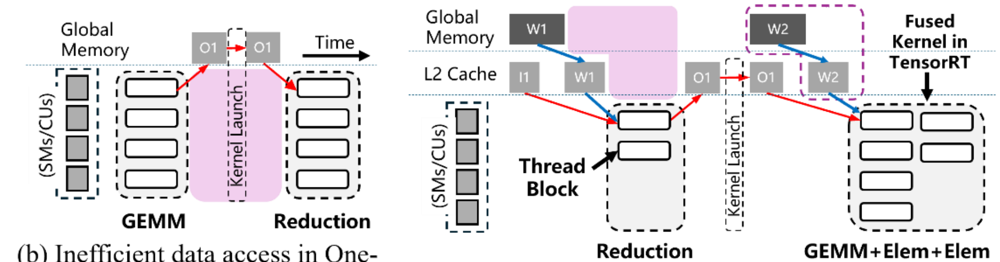
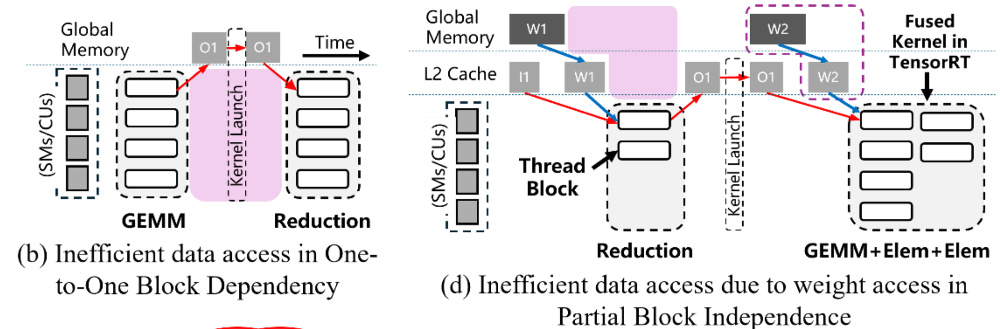
Four Block Dependency Types:

- One-to-One Block Dependency (b)
- Many-to-Many Block Dependency (c)
- Partial Block Independence (d)
- Full Block Independence (e)



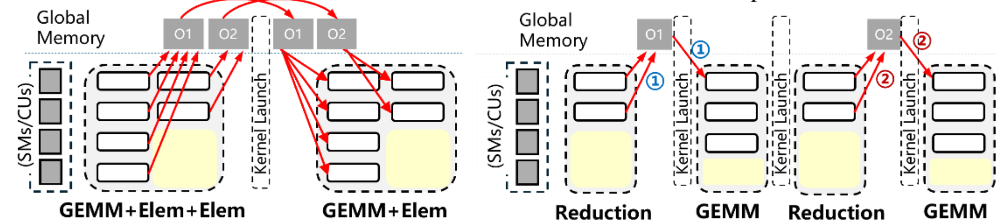
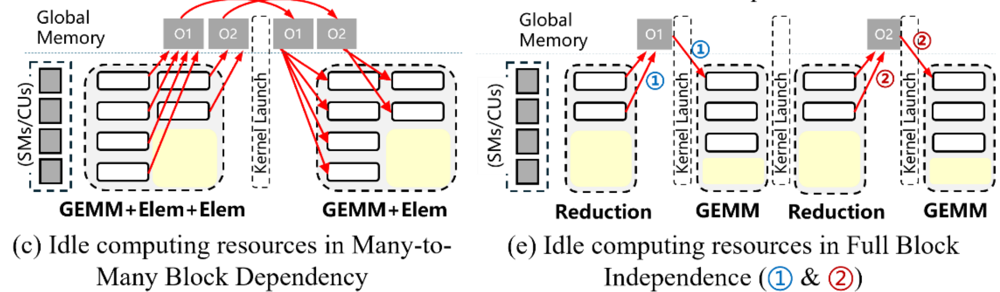


(a) Two subgraphs in BERT and MOE models optimized via TensorRT

(b) Inefficient data access in One-to-One Block Dependency

(d) Inefficient data access due to weight access in Partial Block Independence

(c) Idle computing resources in Many-to-Many Block Dependency

(e) Idle computing resources in Full Block Independence (① & ②)

STEP 3 - Code Optimization

One-to-One Block Dependency: Data Reuse Optimization

```
FusedProgram(I0, W0, W1, O1): # O0 & I1 are removed
 shared SI0[16][16], SW0[16][16], SW1[16][16]
 shared SO0[16][16], SO1[16][16]
 if threadIdx.x < … and threadIdx.y < … # STAGE0: GEMM0
  for k in range 2:
   # Global I0,W0 to Shared SI0,SW0
   ldg2s(SI0, I0[…][…])
   ldg2s(SW0, W0[…][…])
   wmma_16x16(SO0, SI0, SW0)
 block.sync() # Maintain dependency between GEMM0&GEMM1
 if threadIdx.x < … and threadIdx.y < … # STAGE1: GEMM1
  # Global W1 to Shared SW1
  ldg2s(SW1, W1[…][…])
  wmma_16x16(SO1, SO0 + …, SW1) # SO0 reused
  sts2g(O1, SO1)
```

Kernel after fusion

In the new kernel, intermediate results are reused in shared memory, **reducing global memory access** and the number of kernel launches.

Full Block Independence: Parallel Kernel Fusion

```
SubProgram_0(P0_ARGS, PB, PTID, PBID): # Number of blocks - P0_BLOCKS
 LO0[P0_LOCAL_SIZE]
 shared *P0_SI0 = PB + P0_OFFSET0
 shared *P0_SI1 = PB + P0_OFFSET1
 dim3 threadIdx = ThreadMap1Dto3D(PTID) # Thread Index Remapping
 dim3 blockIdx = BlockMap1Dto3D(PBID) # Block Index Remapping
 …… # Tile implementation for SubProgram 0
…… # Other SubPrograms

FusedProgram(GLOBAL_ARGS):
 shared PB[PUBLIC_SHARED_SIZE] # Public buffer shared by all SubPrograms
 if blockIdx.x < P0_BLOCKS and threadIdx.x < P0_THREADS:
  # Execute SubProgram_0 on blocks [0, P0_BLOCKS - 1]
  SubProgram_0(P0_ARGS, PB, threadIdx.x, blockIdx.x)
 else if blockIdx.x < P0_BLOCKS + P1_BLOCKS and threadIdx.x < P1_THREADS:
  SubProgram_1(P1_ARGS, PB, threadIdx.x, blockIdx.x - P0_BLOCKS)
 …… # Execute other SubPrograms
```

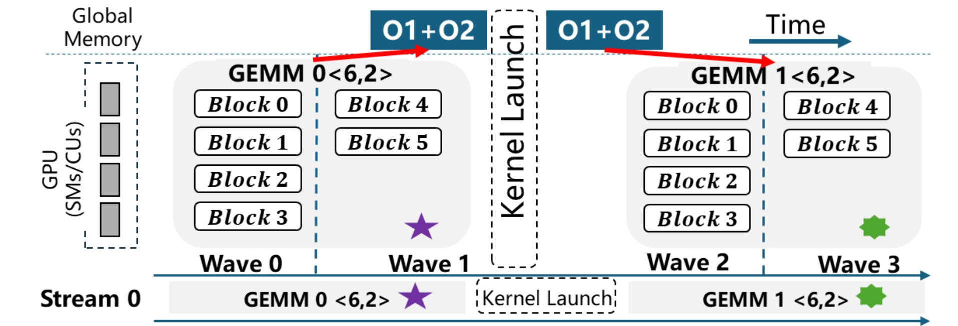
A case study of the kernel template used to generate a fused kernel for parallel optimization

Various implementations are executed based on the block’s serial number, enabling block-level fusion for **improved GPU utilization**.

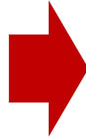
STEP 3 - Code Optimization

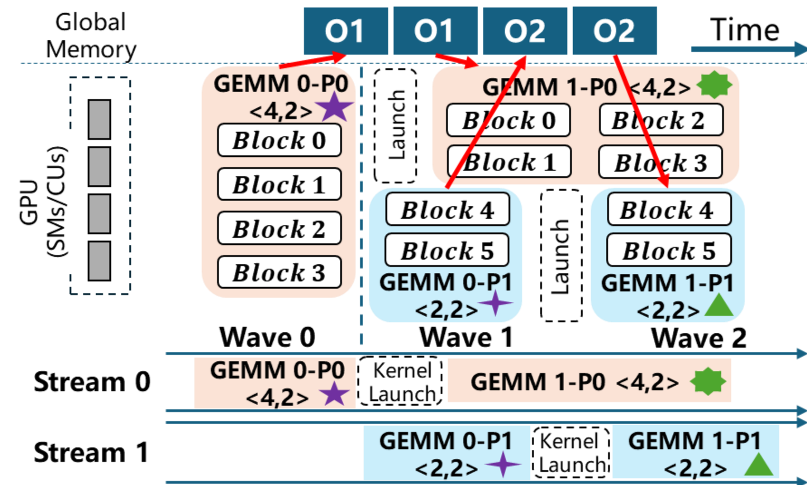
Many-to-Many Block Dependency:

Kernel Splitting and Refactoring Optimization

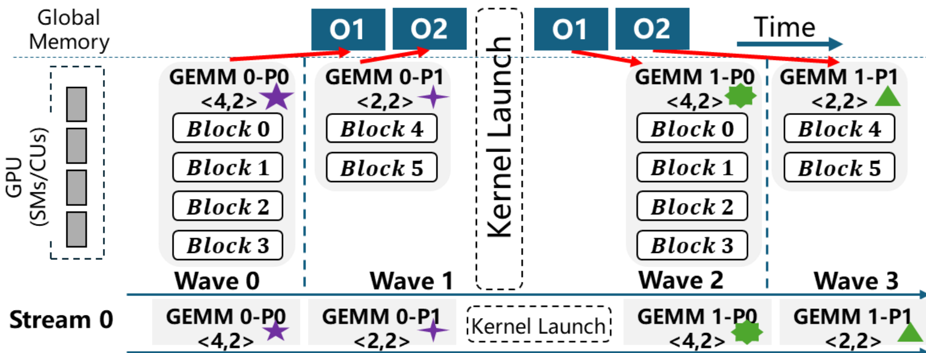


Execute 2 GEMM ops (4 waves) in one stream





Execute 4 reorganized ops (3 waves) in two parallel streams



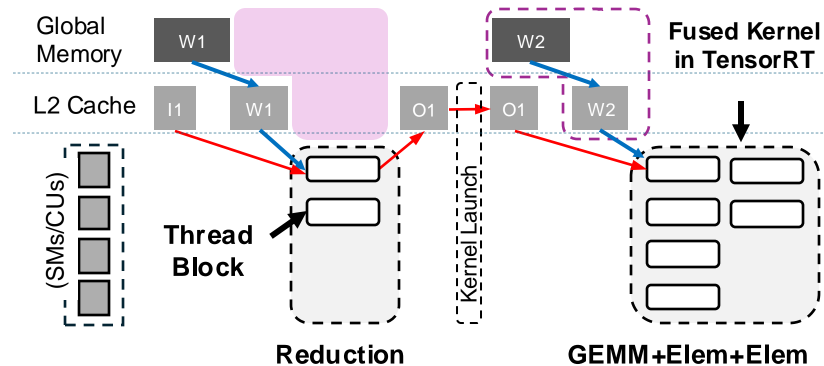
Execute 4 reorganized ops (4 waves) in one stream

Our strategies enhance the utilization of parallel computing resources and **reduce the number of waves**.

STEP 3 - Code Optimization

Partial Block Independence:

Data Prefetching



Our approach introduces L2 cache management and partition-aware prefetching optimizations not found incurrent MLCs, enabling more **efficient memory access**.

Experiment

Baselines:

- ONNX Runtime v1.14.0, PyTorch v1.12, PyTorch XLA v2.2, TensorRT v8.5.3, TVM v0.12, Welder, BladeDISC v0.4.0 (AStitch), MIGraphX v2.4 (AMD)
- Libraries: CUTLASS 3.1, xFormers v0.0.29

Configurations:

- PyTorch: JIT optimization enabled
- TVM: Ansor for kernel tuning

Evaluation Setup:

- 1,000+ iterations per workload, results averaged
- Warm-ups included for accuracy

Experiment

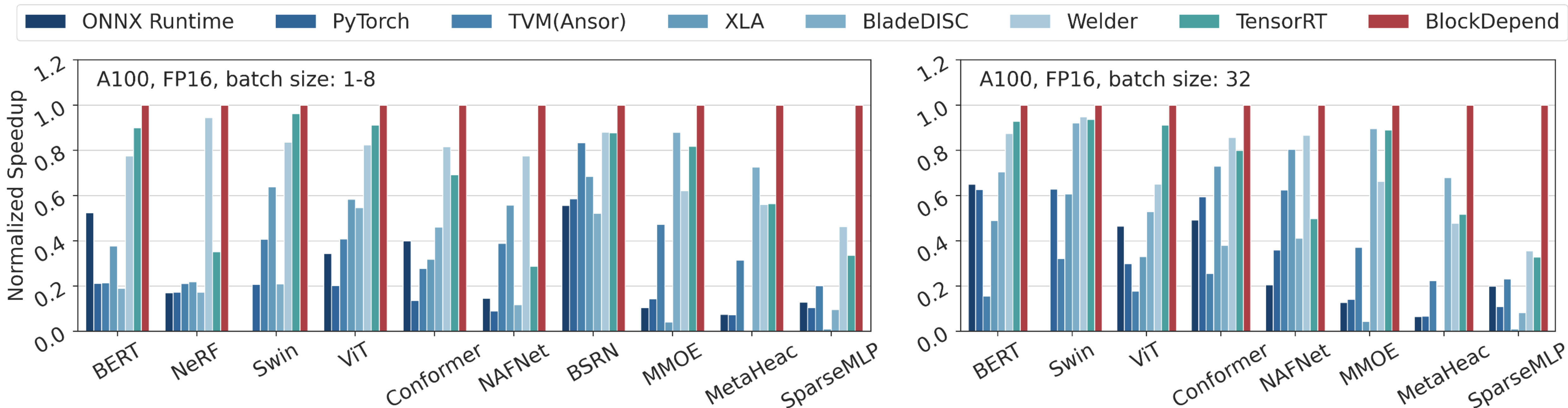
Platforms:

- NVIDIA 40GB A100 GPU, CUDA 12.0, cuDNN v8.7.0
- AMD Radeon MI100 GPU, ROCm 5.4.3

DNN Workloads:

- 12 DNN models tested:
  - BERT, NeRF, Swin-Transformer, ViT, Conformer
  - NAFNet, BSRN, MMoE, MetaHeac, SparseMLP
  - GPT-3, LLaMA
- *SparseMLP derived from Switch-Transformer*
- *All workloads in FP16 precision*

Experiment



End-to-end model inference performance on NVIDIA A100 GPU

Baselines expressed as the normalized speedup relative to the best result (BlockDepend)

- **BlockDepend's significant performance advantage over other systems.**
- **Compared to TensorRT, BlockDepend achieves speedups from 1.04 to 3.47×, averaging at 1.71×.**

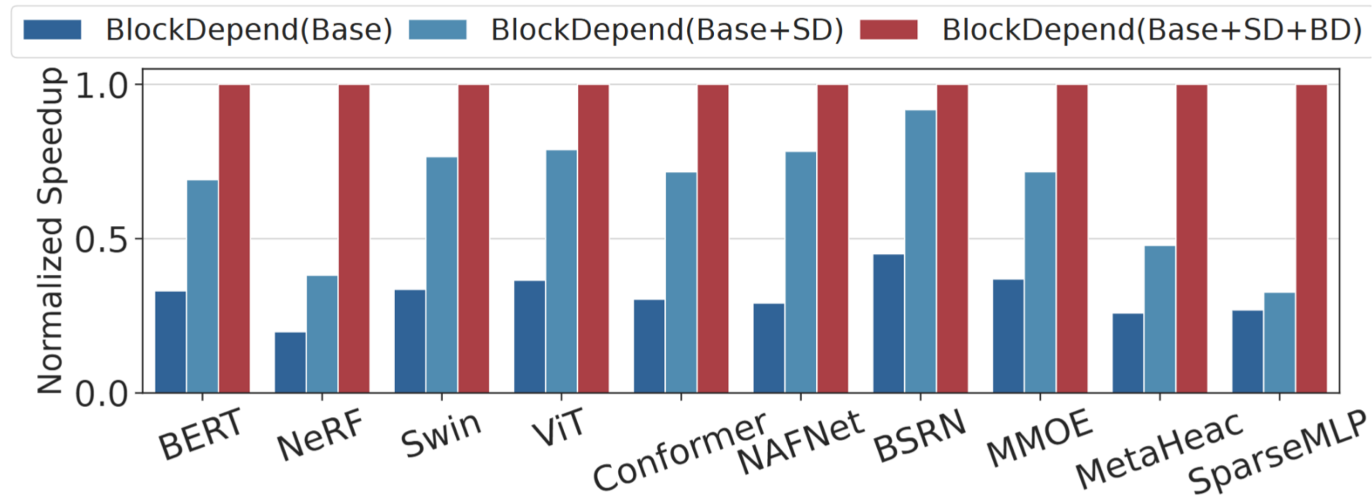
Experiment

Model	Batch	PyTorch (ms)	xFormers (ms)	CUTLASS (ms)	Ours (ms)	Speedup vs CUTLASS
GPT-3-M	768	1.59	1.40	1.38	1.20	1.15×
GPT-3-M	2048	4.25	3.50	2.77	2.56	1.08×
LLaMA-M	768	0.95	0.65	0.71	0.62	1.14×
LLaMA-M	2048	2.03	1.70	1.50	1.34	1.12×
GPT-3-A	768	2.20	1.20	0.79	0.79	1.01×
GPT-3-A	2048	5.55	3.37	2.11	1.86	1.13×
LLaMA-A	768	1.83	0.72	0.54	0.49	1.10×
LLaMA-A	2048	4.80	1.78	1.20	1.11	1.08×

Performance improvement in GPT-3 and LLaMA core structures (M: MLP; A: Attention) on an NVIDIA A100.

- **BlockDepend effectively leverages block-level optimization to reduce idle resources during kernel execution and enhance execution efficiency.**

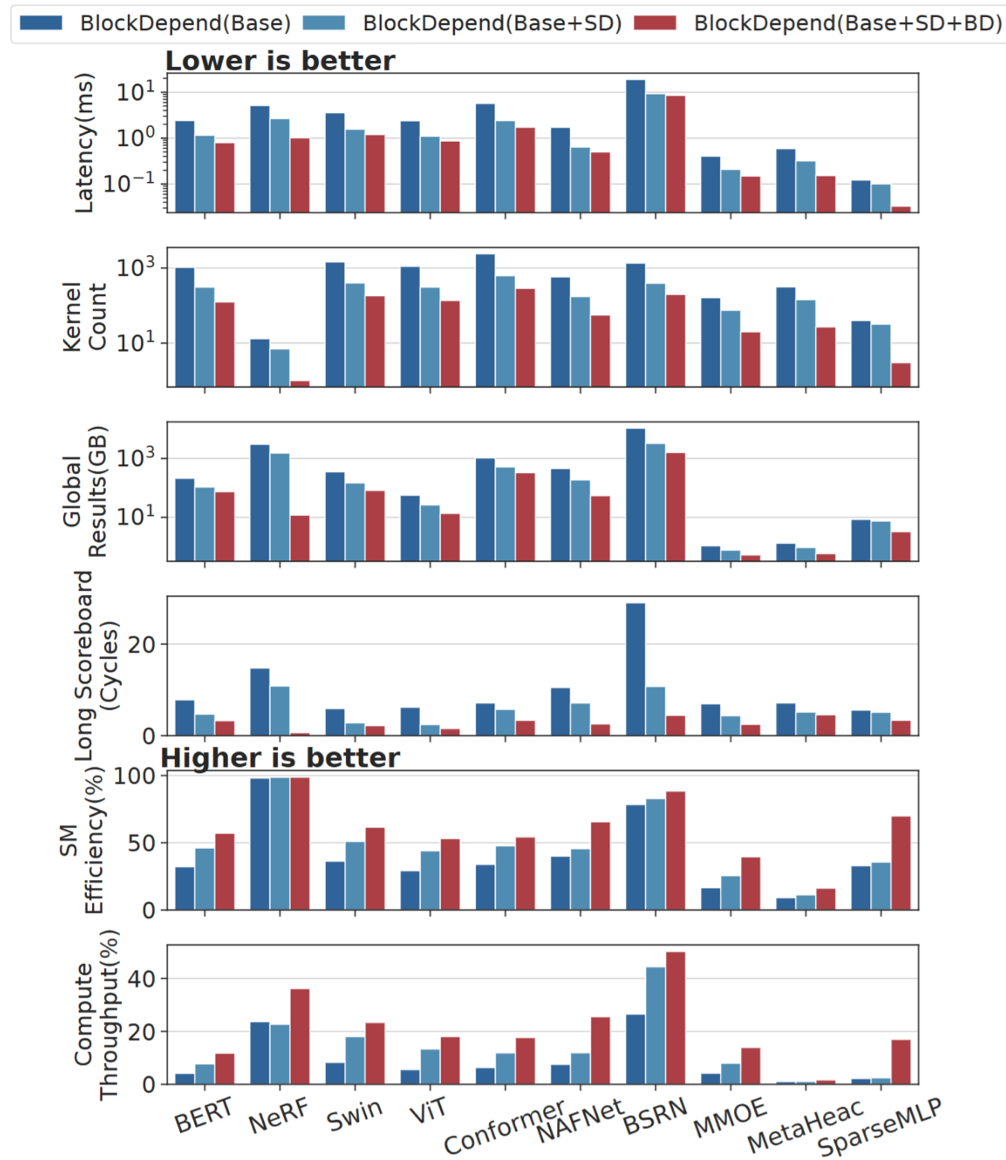
Experiment



Performance improvement breakdown of BlockDepend

- **BlockDepend(Base):** BlockDepend without any inter-operator optimization
- **BlockDepend(Base+SD):** BlockDepend with optimizations based on static data dependency
- **BlockDepend(Base+SD+BD):** the fully optimized BlockDepend, with both previous and block-level optimizations

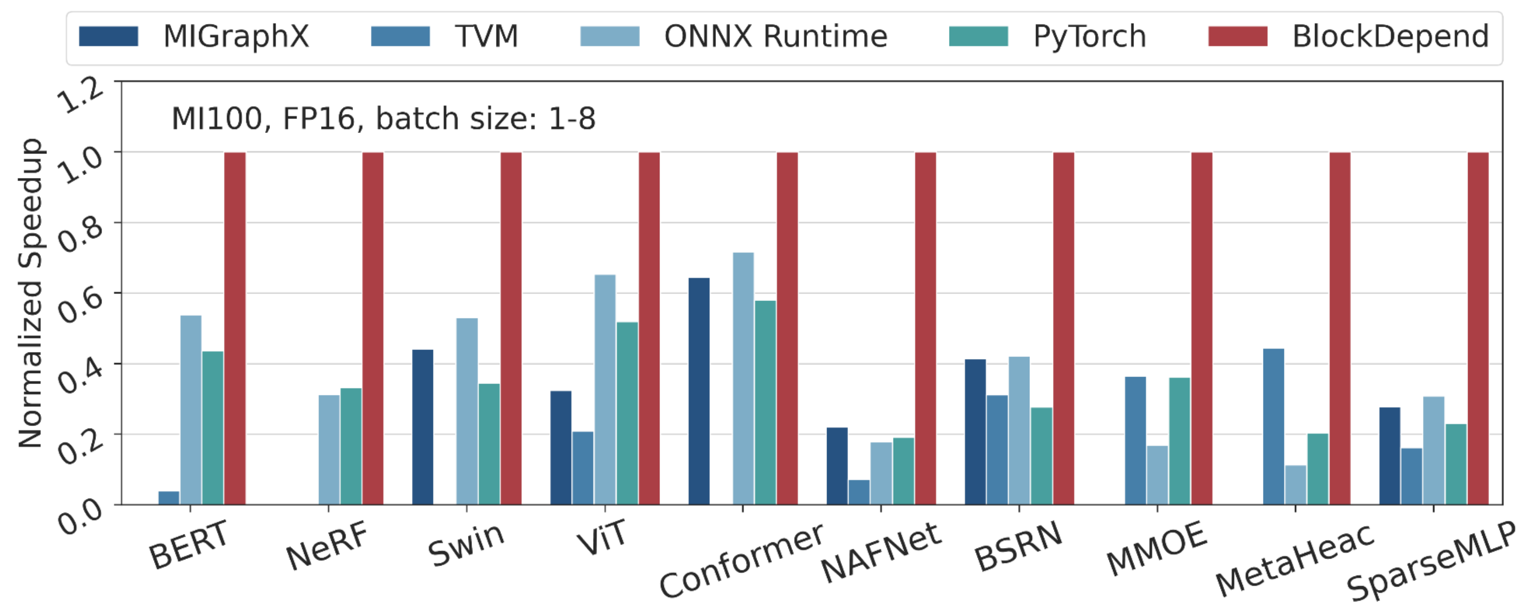
Experiment



Latency, kernel count, global results, long scoreboard, SM efficiency, and compute throughput for workloads with BlockDepend optimizations

- **BlockDepend(Base):** BlockDepend without any inter-operator optimization
- **BlockDepend(Base+SD):** BlockDepend with optimizations based on static data dependency
- **BlockDepend(Base+SD+BD):** the fully optimized BlockDepend, with both previous and block-level optimizations

Experiment



End-to-end model inference performance on AMD MI100 GPU

Experiment

	BERT (s)	BSRN (s)	MetaHeac (s)	SparseMLP (s)
PyTorch	7	12	8	7
TensorRT	59	396	39	22
BladeDISC	55	173	38	23
TVM	9432	30938	10519	3125
BlockOpt	**36**	**133**	**47**	**19**
Roller	368	1125	750	141
Total (Our)	416	1280	804	166

Compilation time on NVIDIA A100 GPU

BlockOpt time: The duration BlockDepend requires to optimize workloads

- **BlockOpt achieves similar compilation times compared to non-auto-tuned solutions such as TensorRT and BladeDISC.**

Conclusion

1. **An in-depth analysis** of existing problems in current compilers and the introduction of **a novel (Block Dependency) abstraction** to represent the potential dependency rselationships between parallel task units.

2. **A compilation defect detection tool** that identifies inefficiency issues based on the unique types of dependencies among different blocks.

3. **A compilation optimization and code generation tool** comprising four optimization methods targeting various inefficient scenarios, simultaneously addressing data locality and task parallelism limitations.

4. **An implementation** on **both NVIDIA and AMD** GPUs.

JCST Journal of Computer Science & Technology

- Indexed in SCIE, EI, INSPEC, Scopus, DBLP, etc.
- China's first English journal on computer
- Sponsored by ICT, CAS & China Computer Federation (CCF)

Thanks

E-mail: dizhanyuan20s@ict.ac.cn

JCST Official WeChat Account

URL: https://jcst.ict.ac.cn

E-mail: jcst@ict.ac.cn

Tel.: +86-10-62600340

Twitter: JCST_Journal