GPU MODE Community

BitBLAS: Enabling Efficient Low-Precision Deep Learning Computing

Lei Wang (**/leɪ wɑːŋ/**)

leiwang1999@outlook.com

Oct 26, 2024

Outline

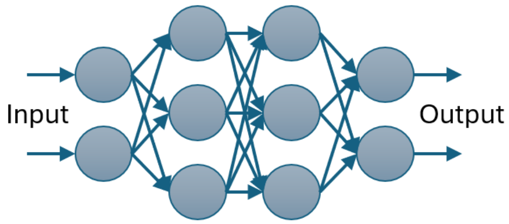
Background: Mixed-Precision Computing

Introduction: Design of BitBLAS/Ladder

Experiments (End2End/OP): NVIDIA/AMD

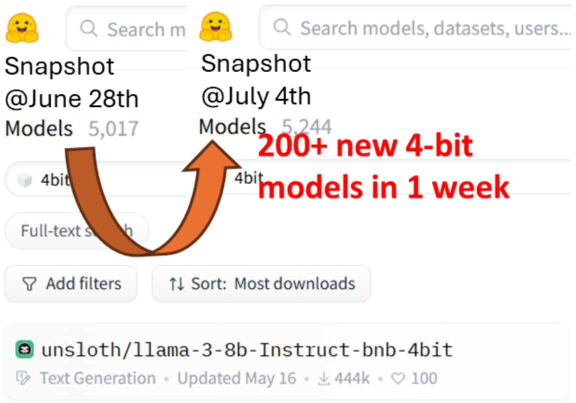
Tutorials in Jupyter: BitBLAS\Ladder\Tile Language

Larger Scale, Fewer Bits



Stable Diffusion

LLAMA-65B
LLAMA-2-70B
LLAMA-3-400B



FP32 → FP16 | FP8 | MXFP | INT4 | FP4 | INT1 | ......

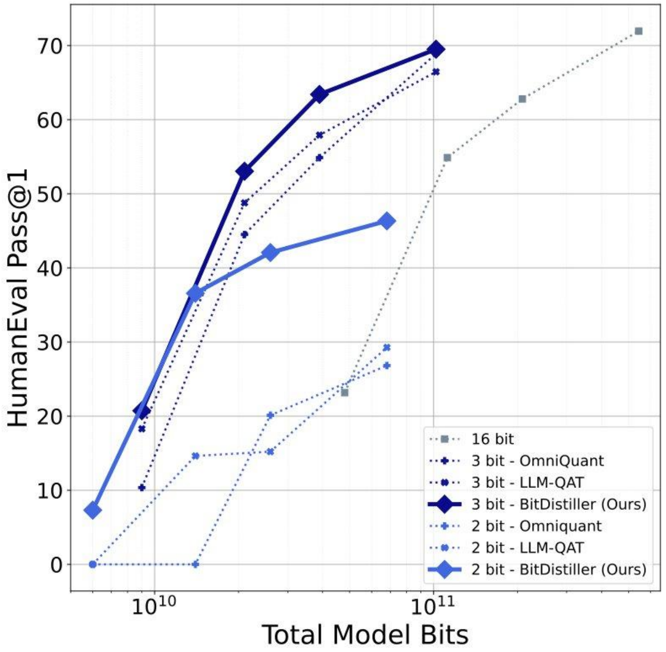
Conventional Quantization:

DNN → quant → Efficient Hardware Inst. : INT8xINT8 Tensor Core...

LLAMA-2-7B with FP16 precision requires at least **14GB** of memory to host the model

Model	**Checkpoint**
LLAMA-7B	13 GB
LLAMA-13B	37 GB
LLAMA-30B	76 GB
LLAMA-65B	122 GB

LLM → quant → Efficient Memory Utilization: FP16xINT4/INT2 ...



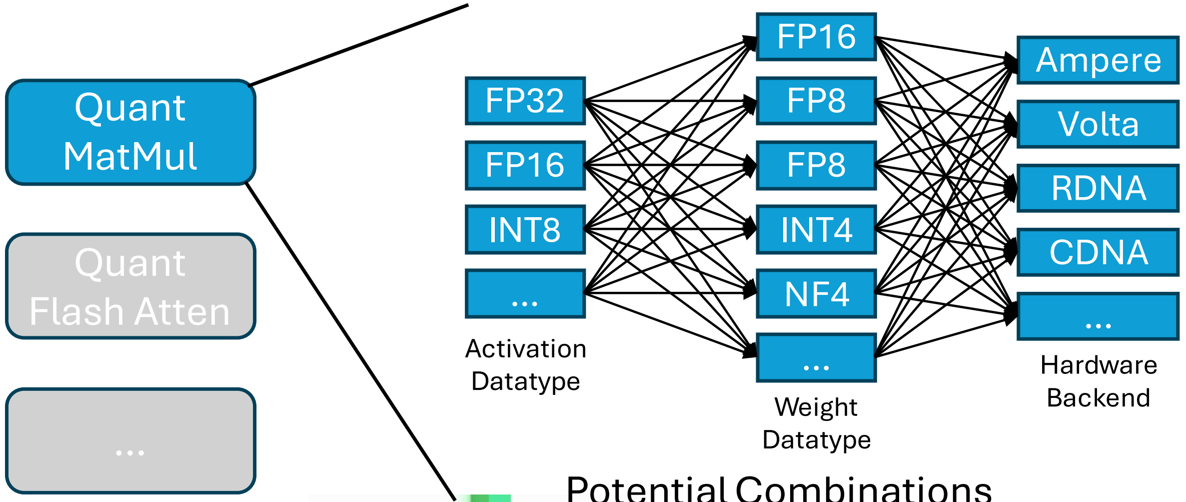
Recent research has pushed the boundaries of low-bit !

bits

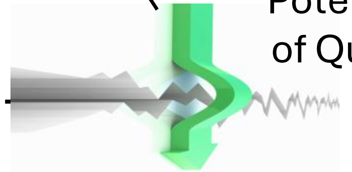
8 SmoothQuant
AutoGPTQ
4 BitDistiller*
2 BitNet-1.58bits*
1 BitNet* OneBit

*represents research from MSRA

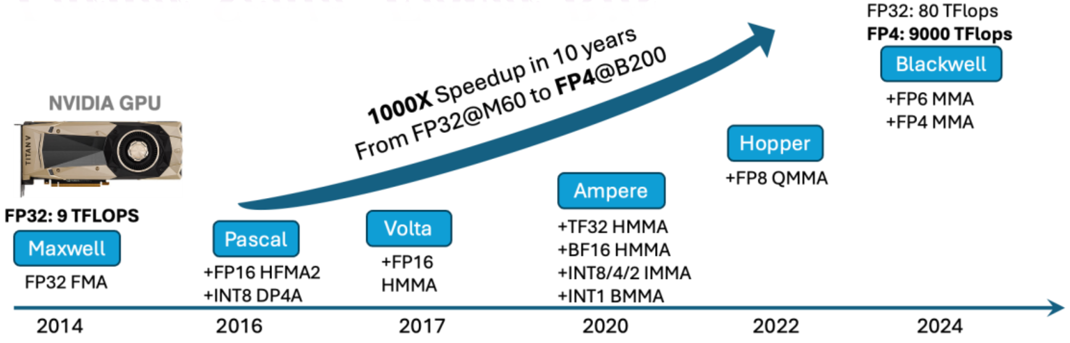
Challenges



Potential Combinations of Quant MatMul in LLM



Larger Scale, Fewer Bits



Hardware evolutions of Lower Precision Computing

Three Major Challenges

Unsupported numerical precision in software

New data types such as NF4/AF4/MXFP have emerged.

Unsupported compute inst. in hardware

Most Hardware doesn't have FP16xINT4 unit.

Combination explosion and hard to optimize

Though vendors and developers has given attention.

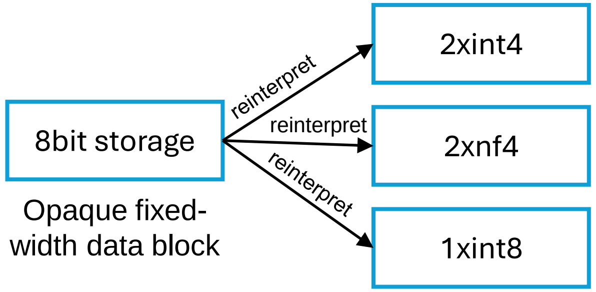
Supports of Vendor Library and MLC

Data Type	$W_{FP16}A_{FP16}$			$W_{INT8}A_{INT8}$			$W_{FP8}A_{FP8}$	$W_{NF4}A_{FP16}$
GPU	V100	A100	MI250	V100	A100	MI250	V100/A100/MI250	
cuBLAS	78%	87%	X	X	68%	X	X	X
rocBLAS	X	X	46%	X	X	75%	X	X
AMOS	64%	38%	X	X	45%	X	X	X
TensorIR	67%	56%	22%	X	X	X	X	X
Roller	50%	70%	29%	X	X	X	X	X

Insights

Key Observation 1
The memory system has compatibility.

How?

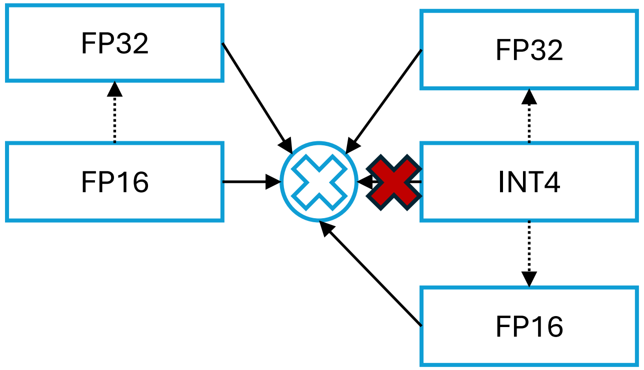


Can be reinterpreted into arbitrary datatypes

The memory system can store any data type by converting these custom data types into fixed-width opaque data blocks.

Key Observation 2
The compute inst. has compatibility.

Which?

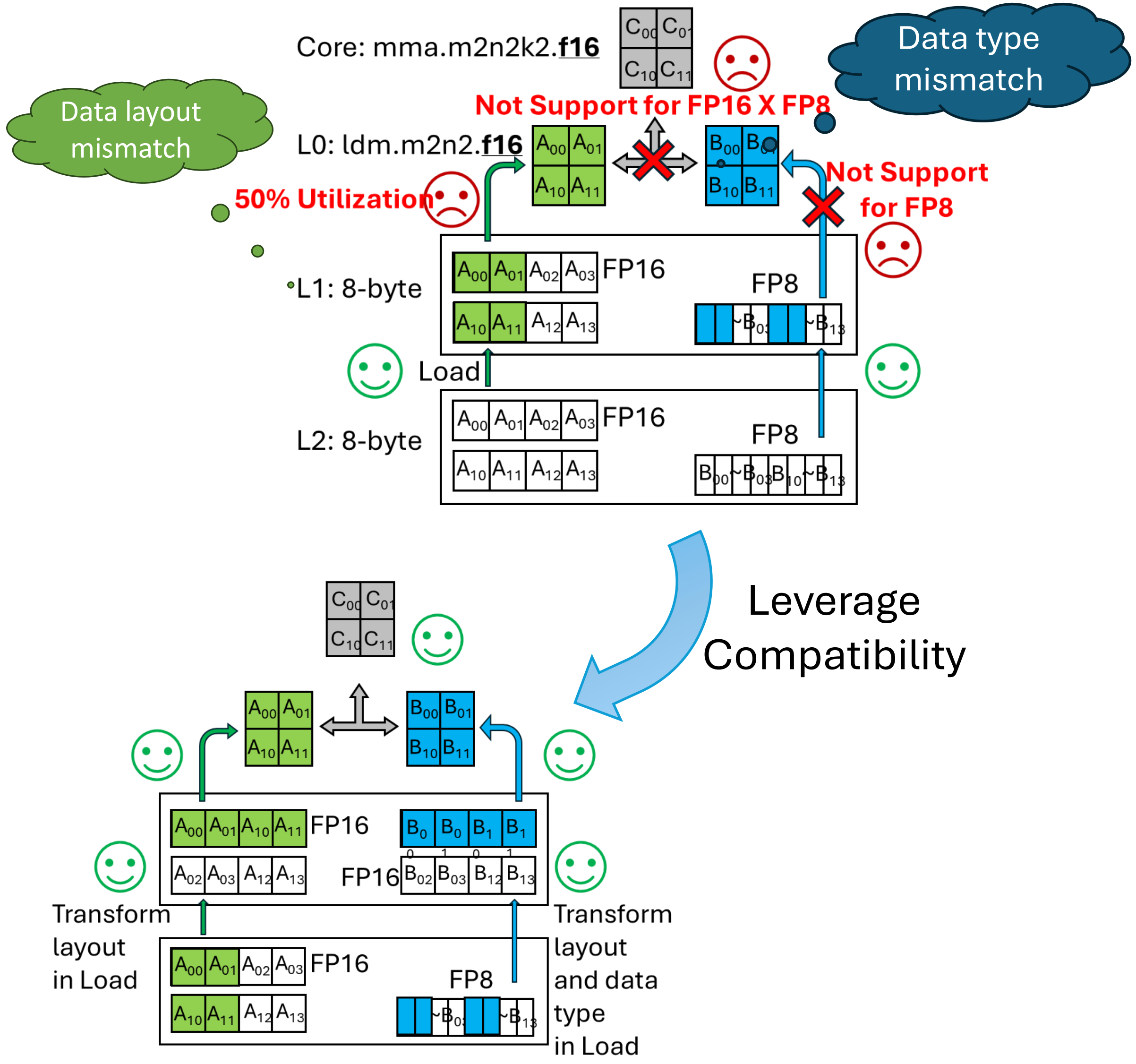


Losslessly compatible with FP16/FP32.

Most custom data types can be losslessly converted into wider standard data types supported by existing hardware computing units for processing.

Mixed-Precision GEMM Execution Flow

C[M,N]@FP16=A[M,K]@FP16 X B[N,K]@FP8, M=2, N=2, K=4



Separate Datatype and Computing with Machine Learning Compilation

Conventional MLC

Separate Compute from Schedule

A Python Like DSL

↓

Intermediate representation
TensorIR, MLIR ...

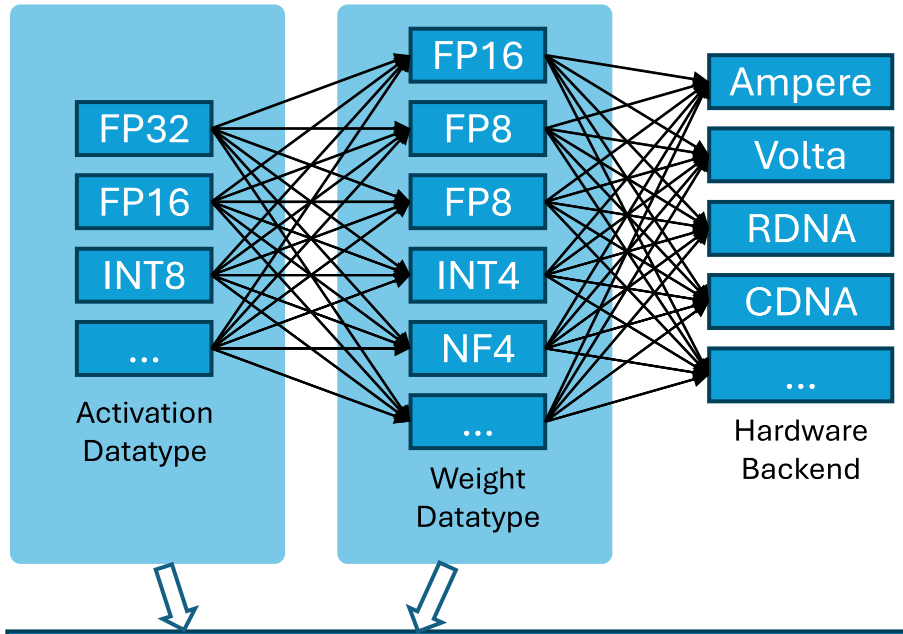
↓

Transformation
Loop unroll, vectorize...

↓

Backend
Generate code or executables for different hardware

Like ML Compilation, Can we ..

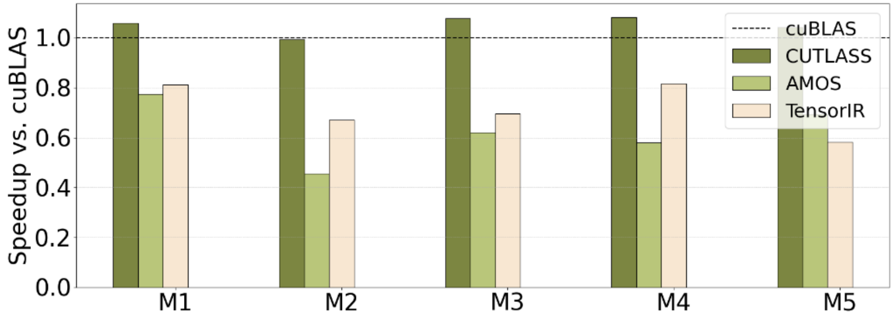


We need a universe Type Representation to hide the conversion and do efficient codegen.

However, the performance of current machine learning compilation tasks is still unsatisfactory, even under hardware-supported instructions.

Existing compilation systems fail to fully utilize the performance of computing units

MatMul Performance of MLC under RTX3090(Tensor Core)



AMOS, Tensor IR can only reach 60-80% performance of cuBLAS.

Major Factors for Performance

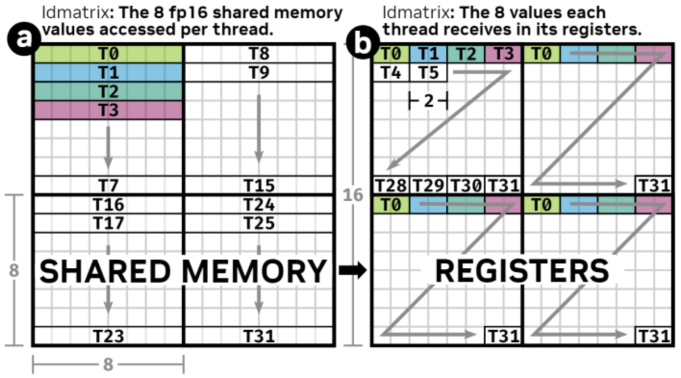
1) Efficient Tiling Existing MLC primitives can handle

Control the compute-to-memory ratio, cache usage size, and register size

2) Utilize Bandwidth can not handle

Better Memory Access pattern

Simple memory accesses struggle to meet the demands of various storage levels simultaneously.



GMEM: expect coalesced access

SMEM: expect free bank conflict

REG: align with instruction

A Swizzling Rule for 8-Bit Tensor Cores (NVIDIA GTC 2020)

It's hard to get the rule

Swizzle Inventor (ASPLOS 2021)

Graphene (ASPLOS 2023)

Insight: The Abstract needs to be aware of and manipulate the data layout of tensors!

Tensor-Centric System Abstractions

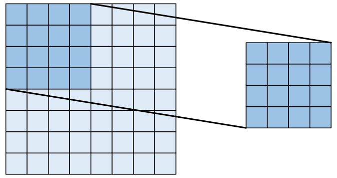
```
class tType {
    TileShape shape;
    size_t nElemBits;
    struct metaData;
    map<tType, prim_func>  ctypes;
}
```

```
class tTile {
    TileShape shape;
    tType type;

    struct metadata;
}
```

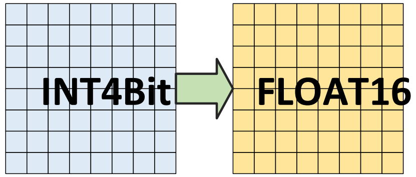
```
struct IndexMap {
    Array initial_indices;

    Array final_indices;

}
```

Four tTile Schedule Primitives

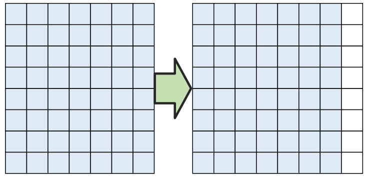
`tTile slice(tTile, index, shape, output_shape);`



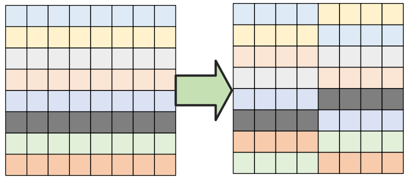
`tTile Convert(tTile, scope, c_func);`



`tTile Pad(tTile, pad_shape, pad_value);`



`tTile TransformLayout (tTile, scope, index_map);`



An example of using tTile to build a mixed Precision Computing expression:

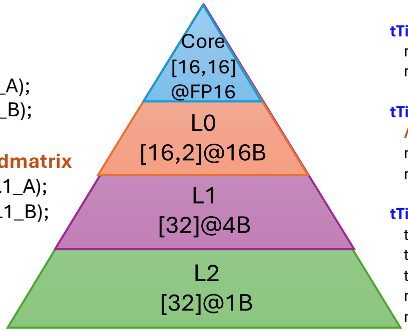
```
C = compute((M, N),
    lambda i, j: (sum A[i, k]@FP16 * B[j, k]@NF4)@FP32
      )@FP32@FP16)
where M=32,N=32,K=63
```

A General Type System

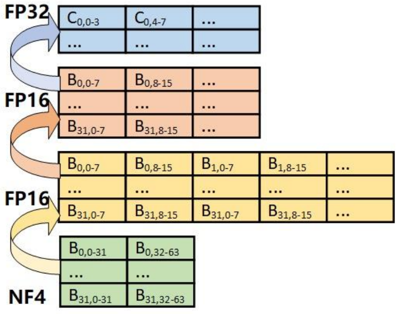
Schedule Primitives For tTile

Enable ml compiler to schedule Tensor across different Operators and Memory layers

```
for L1_iter in L2_tTile.split(L1_tTile):
  // Load A and B from L2 to L1
  L1_A = TransformLoad_L1A(L1_iter.L2_A);
  L1_B = TransformLoad_L1B(L1_iter.L2_B);
  for L0_iter in L1_tTile.split(L0_tTile):
    // Load A and B from L1 to L0 with ldmatrix
    L0_A = TransformLoad_L0A(L0_iter.L1_A);
    L0_B = TransformLoad_L0B(L0_iter.L1_B);
    // Compute with mma instruction
    L0_C = Compute(L0_A, L0_B);
    // Store C to L2
    TransformStore_L0C(L0_C, L2_C);
```



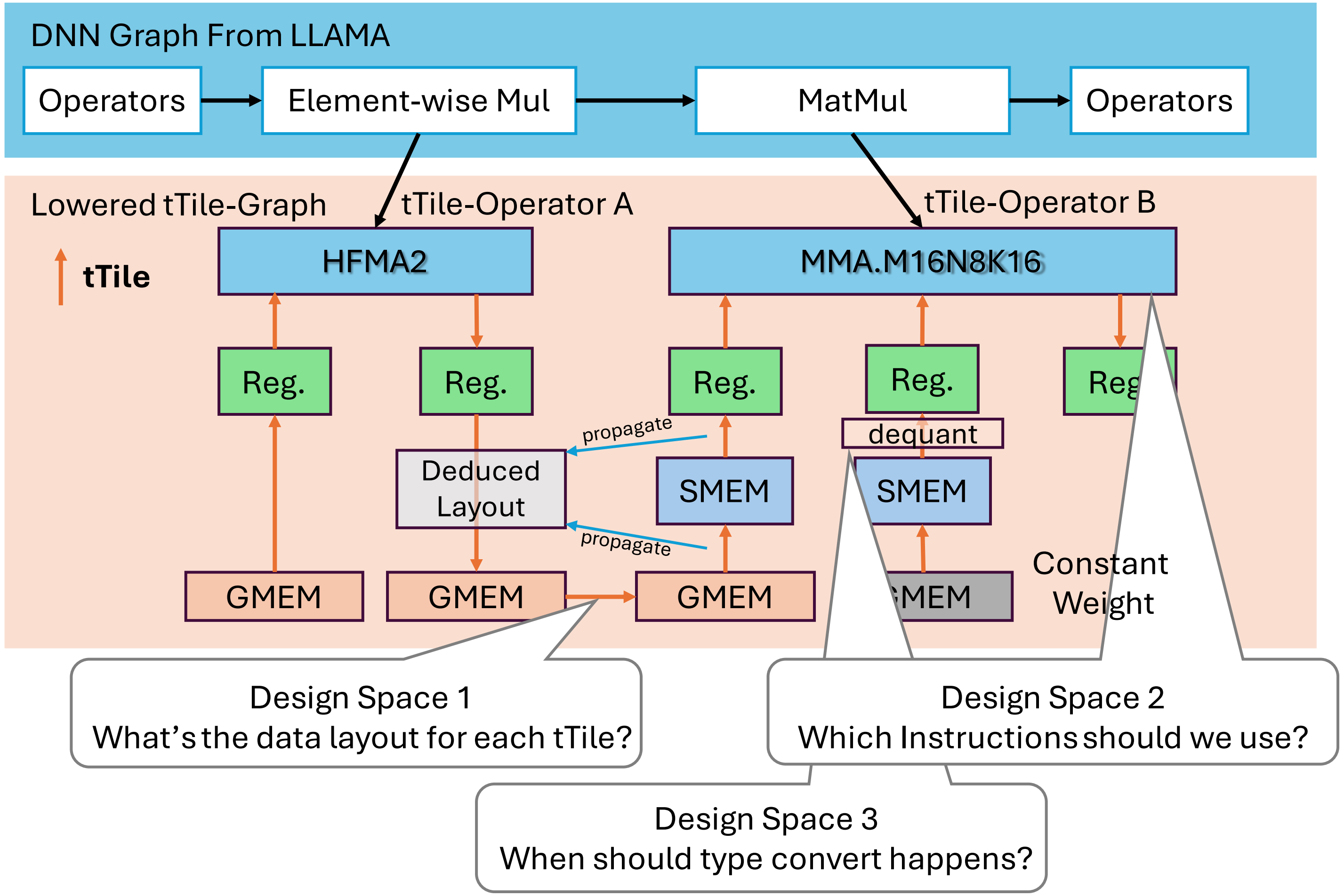

```
tTile Compute(tTile_A, tTile_B):
  ret = mma.f16.f32(tTile_A, tTile_B);
  return ret;

tTile TransformLoad_L0B(tTile):
  // slice with ldmatrix, m8n8x4
  ret = slice(tTile, 0, [4, 64], [16, 16]);
  return ret;

tTile TransformLoad_L1B(tTile):
  t0 = slice(tTile, 0, [16, 63], [16, 63]);
  t1 = pad(t0, [0, 0, 0, 1]);
  t2 = convert(t1, FP16);
  ret = transform_layout(t2, map_func);
  return ret;
```



An example scheduled executed plan with tTile schedule primitives on nvidia gpus.

New Design Space

Example of our tTile-Graph abstraction for end2end optimization from LLAMA, enabling more fine-grained control across operators and even different memory layers.



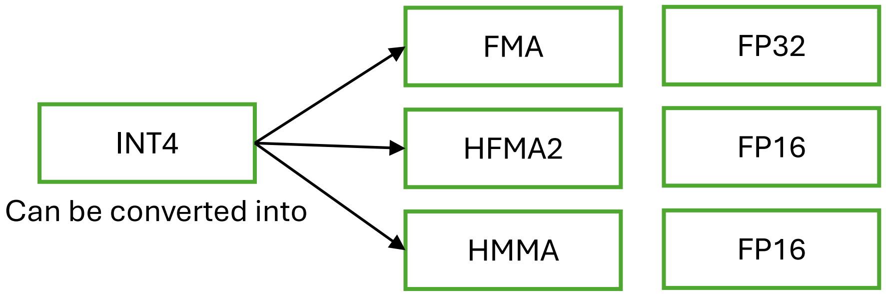
These abstractions enlarge the scheduling space for DNN computation!

More detail, download:

OSDI 2024' Ladder

Auto Normalize Computation into Hardware Instructions

Bit-nearest instruction matching



Matches the instruction type to be converted based on the instruction computation pattern and throughput.

Device	Inst	Data Type	TFLOPS/OPS	Expression
RTX 3090	DFMA	FLOAT64	8.9 TFLOPS	D[0] = A[0] * B[0] + C[0]
RTX 3090	FMA	FLOAT32	35.6 TFLOPS	D[0] = A[0] * B[0] + C[0]
RTX 3090	IMAD	INT32	17.8 TOPS	D[0] = A[0] * B[0] + C[0]
RTX 3090	HFMA2	FLOAT16	35.6 TFLOPS	D[0:2] = A[0:2] * B[0:2] + C[0:2]
RTX 3090	DP4A	INT8	71.2 TOPS	D[0] = dot(A[0:4], B[0:4]) + C[0]
RTX 3090	HMMA.m16n8k16.f16	FLOAT16	142 TFLOPS	D[0:16, 0:16] = dot(A[0:4], B[0:4]) + C[0]
RTX 3090	IMMA.m16n8k32.s8	INT8	284 TOPS	D[0:16, 0:16] = dot(A[0:4], B[0:4]) + C[0]

Iterator-based auto expr normalization

Example of normalizing conv2d into tensorcore inst.

Tutorial: Auto Tensorize

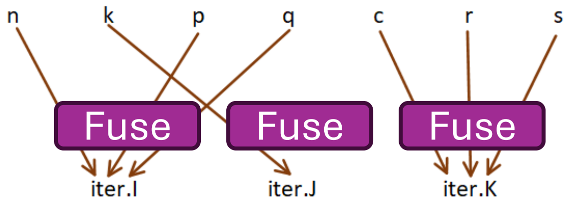
Which enables us to explore if a given customized op(conv, stencil) can be tensorized by target instruction.

(a, conv2d Expression

```
for {n, k, p, q} in domain{128, 64, 112, 112}:
    for {c, r, s} in domain{3, 7, 7}:
        out[n, k, p, q] += input[n, c,
    p+r, q+s]* weight[k, c, r, s]
```

(b, tensorcore expression

```
for {i, j, k} in domain{16, 16, 8}:
    out[i, j] += input[i, k]* weight[k,j]
```



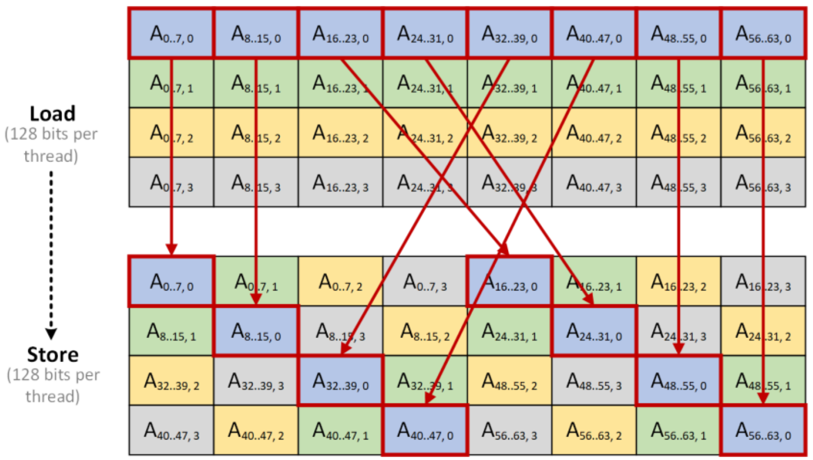
(c, Classified by iterators

(d, Auto-normalized conv2d program

```
for {n, p, q, r, s, c} in domain{128, 112, 112, 7, 7, 3}:
  input1[n * 12544 + p * 112 + q, r * 21 + s * 3 + c]
        = input[v0, v1 * 2 + v3, v2 * 2 + v4, v5]

for k, r, s, c in domain{64, 7, 7, 3}:
  weight1[k, r * 21 + s * 3 + c] = weight[k, r, s, c]

for {i, j, k} in domain{1605632, 64, 147}:
  out[i, j] += input1[i, k]* weight1[k, j]
```

Layer	n	k	p	q	c	r	s	stride	Input Layout	Weight Layout	Target Instructions	Auto Tensorize Mapping
C0	128	64	224	224	3	7	7	2	NHWC	HWIO	mfma.m16n8k16	[n * 12544 + h * 112 + w, f, r * 21 + s * 3 + c] → [I, J, K]
C1	128	64	56	56	64	3	3	1	NHWC	OHWI	mfma.m16n8k16.trans	[n * 3136 + h * 56 + w, f, r * 192 + s * 64 + c] → [I, J, K]
C2	128	64	56	56	64	1	1	1	NHWC	HWIO	mfma.m16n8k16	[n * 3364 + h * 58 + w, f, c] → [I, J, K]
C3	128	64	56	56	64	1	1	1	NHWC	OHWI	mfma.m16n8k16.trans	[n * 3364 + h * 58 + w, f, c] → [I, J, K]
C4	128	128	28	28	128	3	3	1	NHWC	OHWI	mfma.m16n8k16.trans	[n * 784 + h * 28 + w, f, r * 384 + s * 128 + c] → [I, J, K]
C5	128	256	14	14	128	3	3	2	NHWC	HWIO	mfma.m16n8k16	[n * 49 + h * 7 + w, f, r * 384 + s * 128 + c] → [I, J, K]
C6	128	256	14	14	128	1	1	2	NHWC	OHWI	mfma.m16n8k16.trans	[n * 64 + h * 8 + w, f, c] → [I, J, K]

Hardware Aligned Layout Propagation



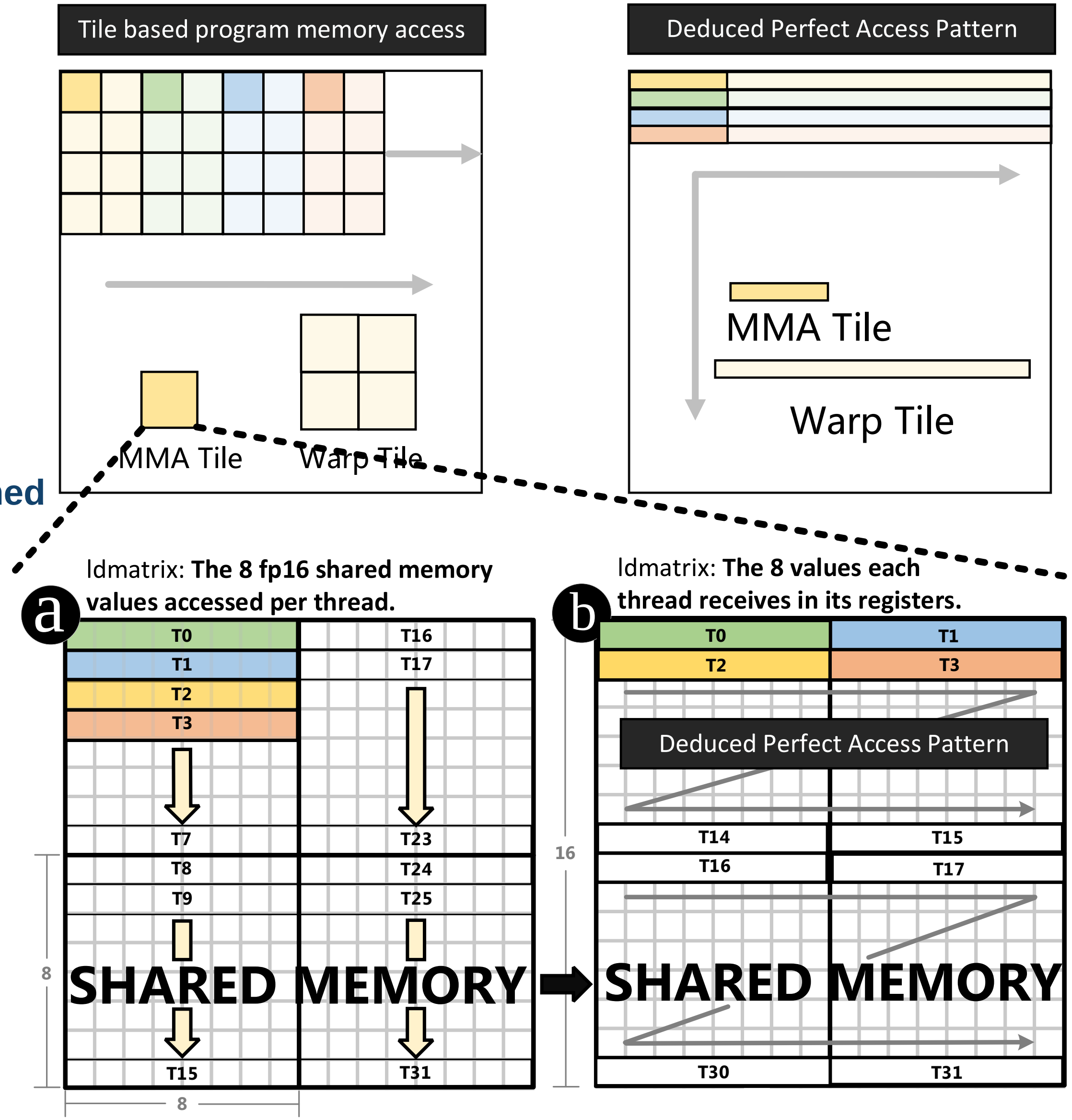
The search space is vast, with possible combinations in the order of O(N!) !
It's impossible to traverse all of them.

tDevice: Hardware abstraction

- Explicitly Define the preferred access pattern for different memory layers.
- Explicitly Define the access pattern for instructions in warp level.

Hardware-Aligned

Optimal Layout Deduction



Hardware Aligned Layout Propagation

Hardware Aligned Layout Deduction

Define Computation with DSL (TIR)

```
@tvm.script.ir_module
class MyModule:
    @T.prim_func
    def main(a: T.handle, b: T.handle, c: T.handle):
        T.func_attr({"global_symbol": "main", "tir.noalias": True})
        A = T.match_buffer(a, [M, K], dtype="float16")
        B = T.match_buffer(b, [N, K], dtype="float16")
        C = T.match_buffer(c, [M, N], dtype="float16")

        for i, j, k in T.grid(M, N, K):
            with T.block("B"):
                vi, vj, vk = T.axis.remap("SSR", [i, j, k])
                with T.init():
                    C[vi, vj] = T.float16(0)
                C[vi, vj] = C[vi, vj] + \
                    A[vi, vk].astype("float16") * B[vj,
vk].astype("float16")
```

Specify a Hardware ("rtx-3090")

Bottom-up hardware instruction selection		
Depth	Type	Instructions
0	Compute	2xmma.sync.aligned.m16n8k16.row.col.f16.f16.f16.f16
1	Shared Load	ldmatrix.sync.aligned.m8n8.x4.trans.shared.b16
2	Shared Store	st.shared.v4.u32
3	Global Load	ld.global.v4.u32

Deduce

The memory-intensive operator for re-layout the input.

```
B[vi // 16, vj // 16, vi % 16, vj % 16] =
  A[vi // 8 * 8 + vi % 4 * 2 + vj % 16 // 8, vj // 16 * 16 + vi % 8 // 4 * 8 + vj % 8]

B[vi // 16, vj // 16, vi % 16, vj % 16] =
  A[vi // 8 * 8 + vi % 4 * 2 + vj % 16 // 8, vj // 16 * 16 + vi % 8 // 4 * 8 + vj % 8]
```

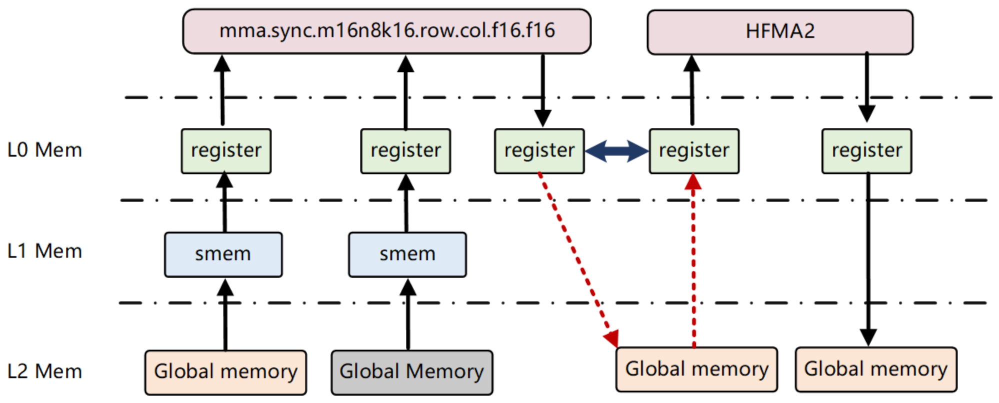
Compute-Intensive Op with Perfect Layout Access

```
@I.ir_module
class Module:
    @T.prim_func
    def main(A: T.Buffer(), B: T.Buffer(), C: T.Buffer():
        __fetch2shared()
        for ax0, ax1, ax2, ax3 in T.grid(1024, 1024, 16, 16):
            with T.block("A_shared_warp"):
                v0, v1, v2, v3 = T.axis.remap("SSSS", [ax0, ax1, ax2, ax3])
                A_shared_warp[v0, v1, v2 * 2 + v3 // 8, v3 % 8] = A_shared[v0, v1, v2, v3]
        for ax0, ax1, ax2, ax3 in T.grid(1024, 1024, 16, 16):
            with T.block("B_shared_warp"):
                v0, v1, v2, v3 = T.axis.remap("SSSS", [ax0, ax1, ax2, ax3])
                B_shared_warp[v0, v1, v2 * 2 + v3 // 8, v3 % 8] = B_shared[v0, v1, v2, v3]
        for ii, jj, kk, i, j, k in T.grid(1024, 1024, 1024, 16, 16, 16):
            with T.block("B"):
                vii, vjj, vkk, vi, vj, vk = T.axis.remap("SSRSSR", [ii, jj, kk, i, j, k])
                with T.init():
                    C_warp[vii, vjj, vi % 8 * 4 + vj % 8 // 2, vj // 8 * 4 + vi // 8 * 2 + vj % 2]
                        = T.float16(0)
                C_warp[vii, vjj, vi % 8 * 4 + vj % 8 // 2, vj // 8 * 4 + vi // 8 * 2 + vj % 2]
                    +=  A_shared_warp[vii, vkk, vi * 2 + vk // 8, vk % 8]
                      * B_shared_warp[vjj, vkk, vj * 2 + vk // 8, vk % 8]
        for ax0, ax1 in T.grid(16384, 16384):
            with T.block("C_warp"):
                v0, v1 = T.axis.remap("SS", [ax0, ax1])
                C[v0, v1] = C_warp[v0 // 16, v1 // 16,
```

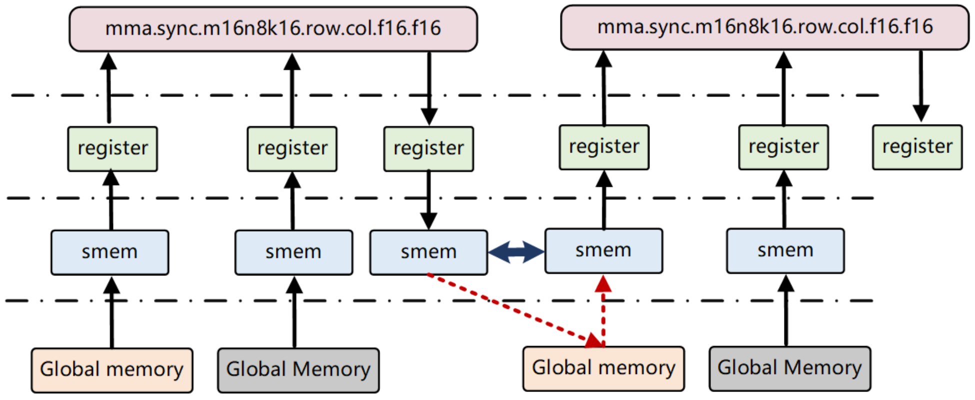
Advantages and Limitations

- **Advantages**: Eliminates the search space for data layout in tensor scheduling, requiring only derivation.
- **Limitations**: Requires pre-conversion of data layout, which introduces conversion overhead.

Resolve the Limitation with Tile-Graph



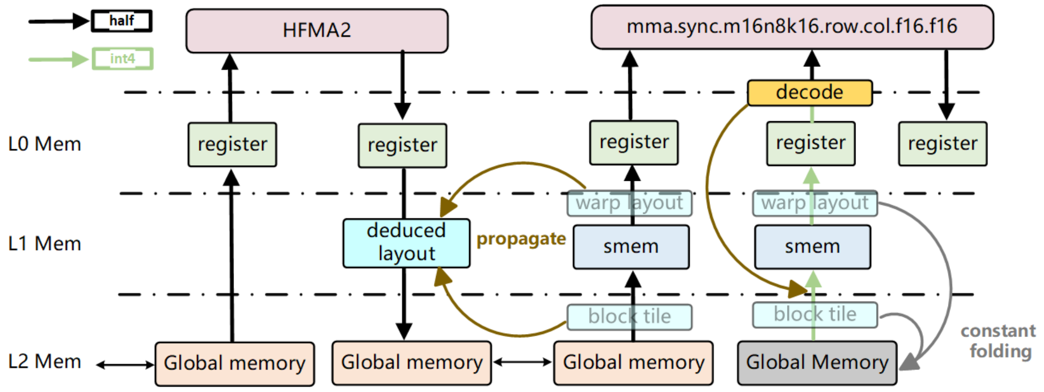
Compute-intensive operators and memory-intensive operators are connected through registers



Compute-intensive operators are connected through shared memory.

OSDI'23: Welder: High Performance Operator Fusion with Tile-Graph

Latency Hiding Method Based on Tile-Graph



Constant Folding for Static Weights: Arrange weights during the compilation phase to hide latency.

Forward Propagation of Data Layout Between Operators: The preceding operator can process and write back data directly in the layout expected by the subsequent operator during execution, thereby avoiding additional data layout conversion operations between the two operators.

Discussion: The performance Impact of introducing Layout Transformation Fusion.

Why we need to introduce Layout Propagation?

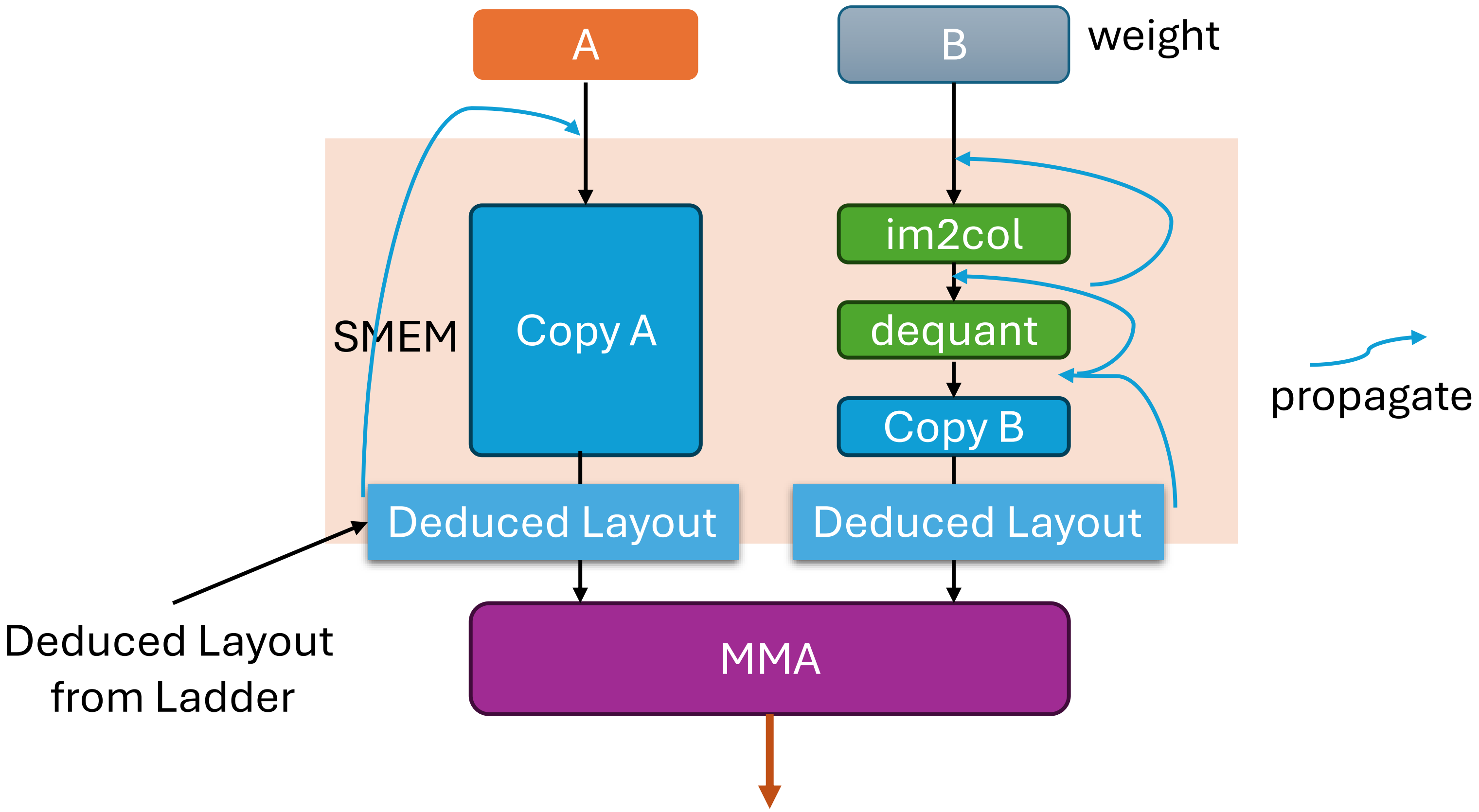
Challenges

1. The dimensions of the instructions and computations do not align.

2. There are several peripheral computations outside the core **MMA** instructions.

3. Complex mapping relationships introduced by nonlinear transformations (dequant, group-scale).

Example of a Compute Flow
With Layout Propagation



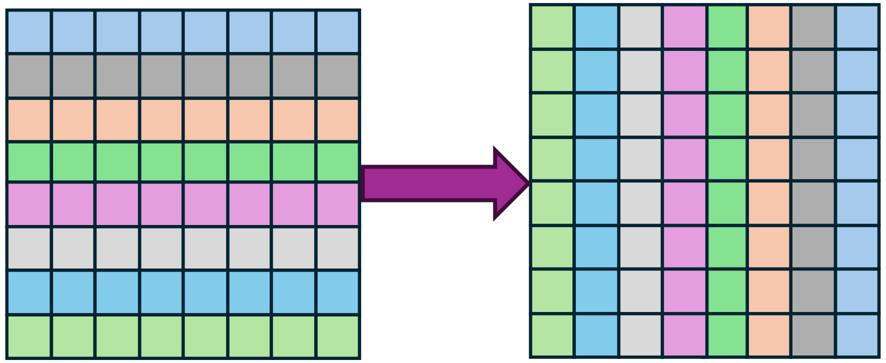
Im2col and dequant will transform the layout as well

The deduced layout should be able to propagate across different compute blocks !

Methodology: Three different layout propagate modes

Case 1: Linear Transformation

Transpose as an example



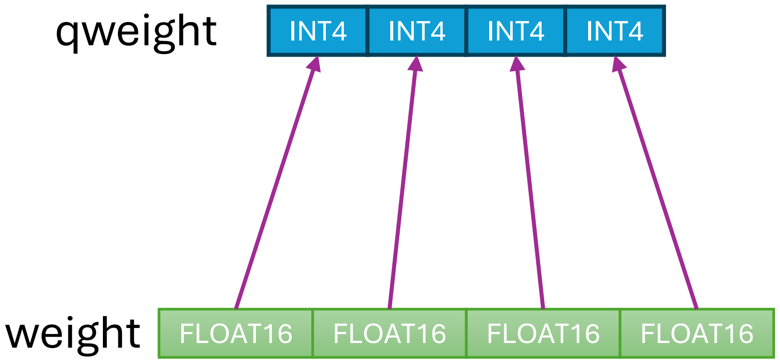
```
lambda i, j: (i // 8 * 8 + j // 8 * 4 + i % 8 // 2, i % 2 * 8 + j % 8)
```

Propagate

```
lambda i, j: (j // 8 * 8 + i // 8 * 4 + j % 8 // 2, j % 2 * 8 + i % 8)
```

Case 2: Compressed Transformation

Dequantize as an example




```
lambda i, j: (i // 8 * 8 + j // 8 * 4 + i % 8 // 2, i % 2 * 8 + j % 8)
```

Propagate

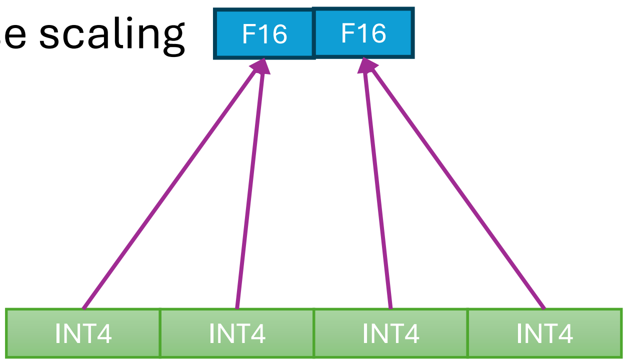
```
lambda i, j: (i // 8 * 8 + j * 4 + i % 8 // 2, i % 2)
```

Methodology: Three different layout propagate modes

Case 3: non-injective Transformation

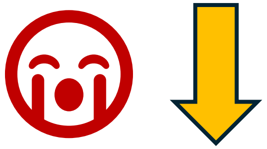
Dequantize as an example

Group wise scaling



Quant weight

```
lambda i, j: (i // 8 * 8 + j // 8 * 4 + i % 8
// 2, i % 2 * 8 + j % 8)
```

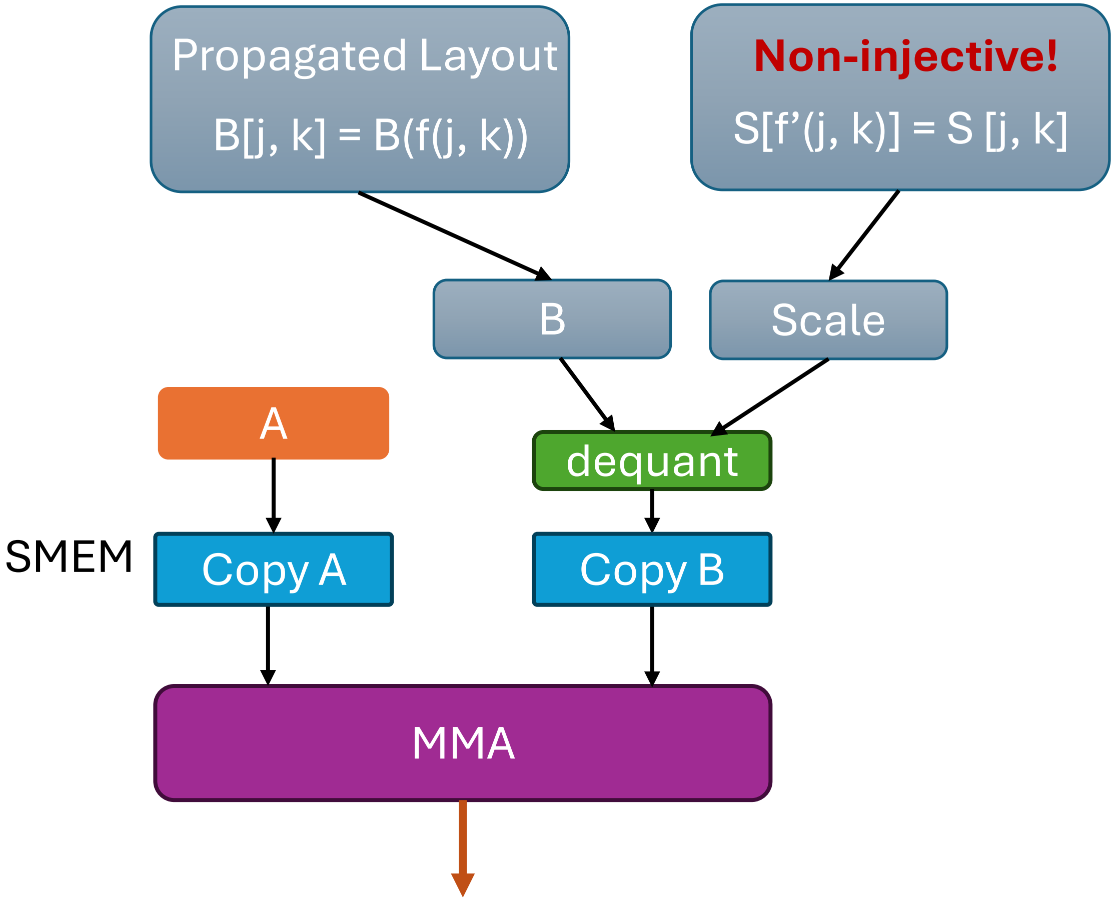


Cannot Propagate

BitBLAS implements auto-layout propagation rules based on three patterns.

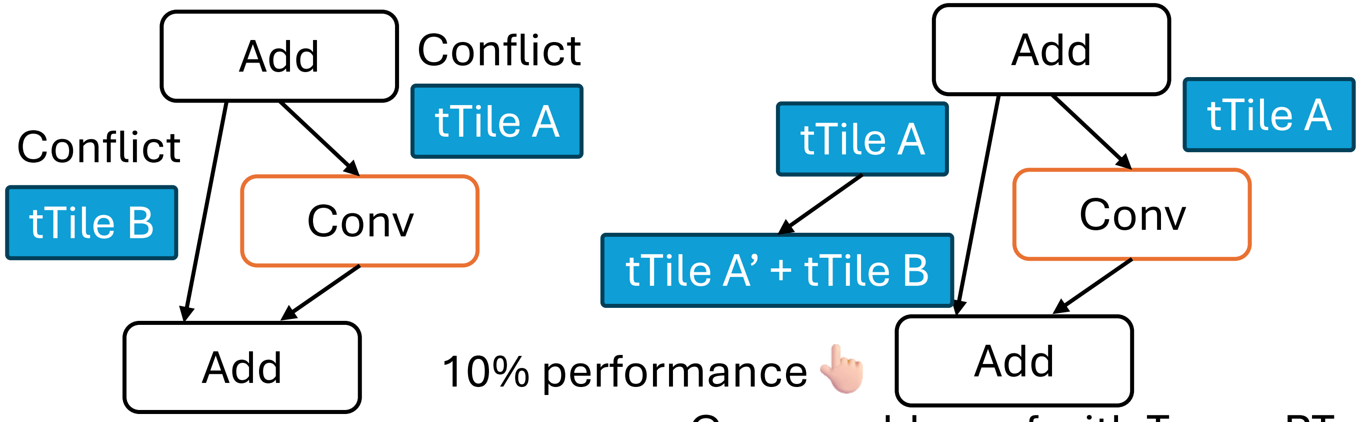
Resolve Conflict: Layout Auto Differentiation

Layout Conflict with correlative Buffers



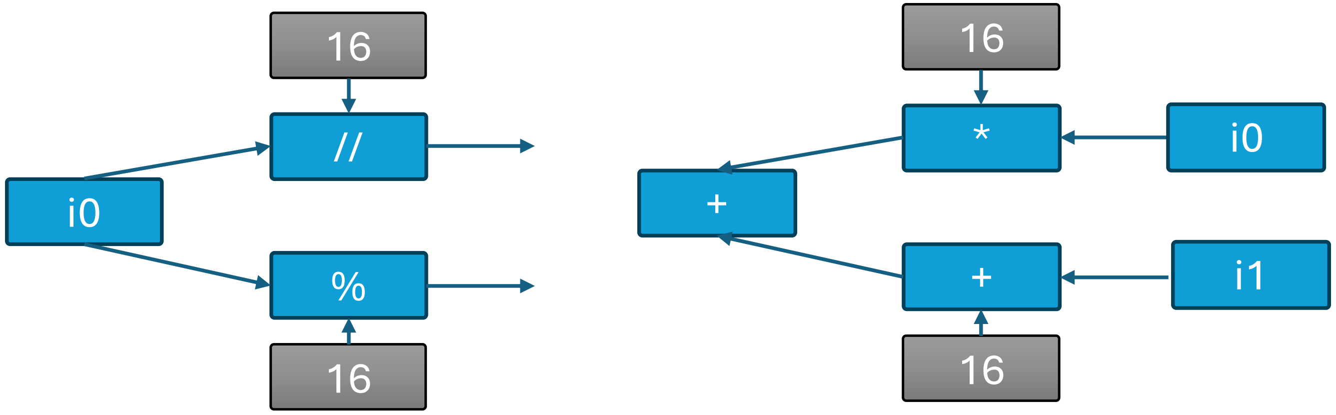
F' represents inverse transformation of Layout.

Resolve Conflict With CSE



Comparable perf with TensorRT

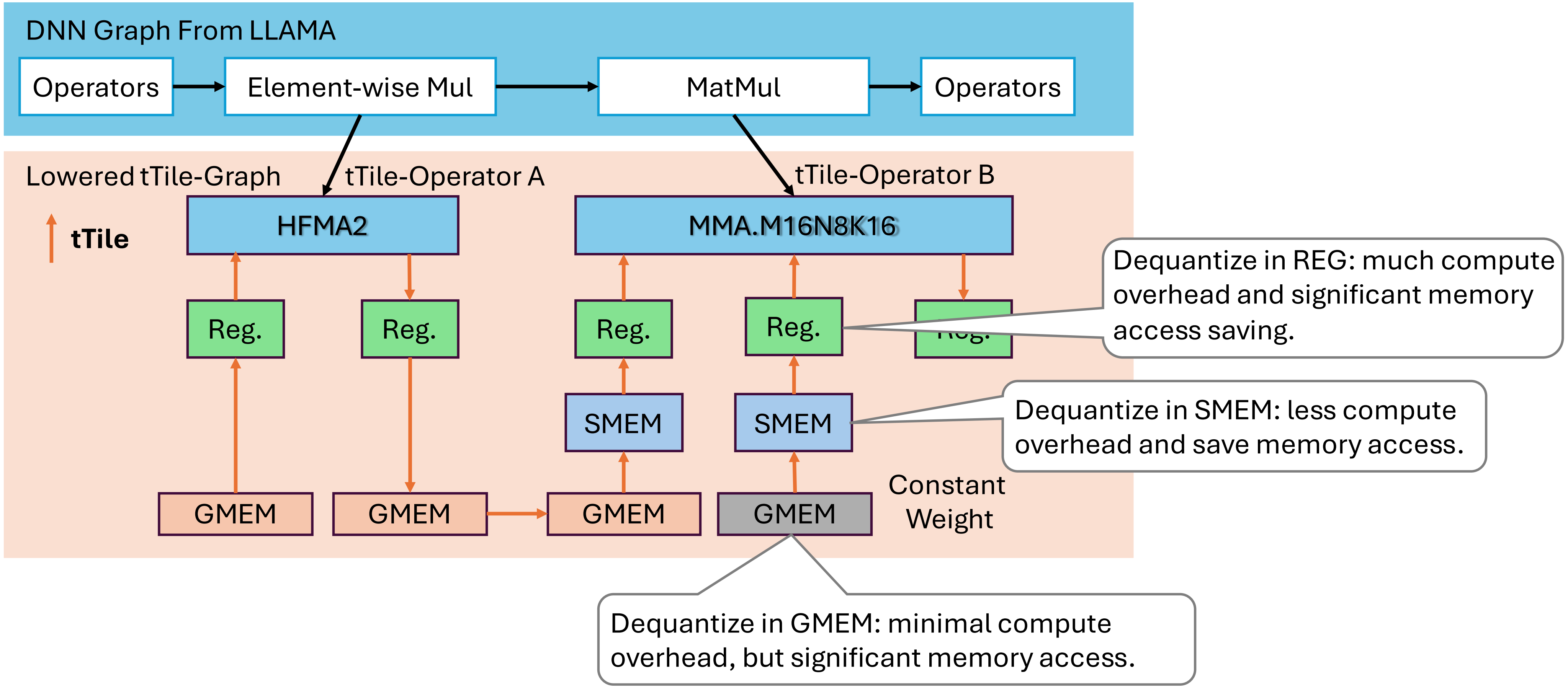
Layout Auto Differentiation




```
lambda i: (i // 16, i % 16)    lambda i, j: i * 16 + j
```

Latency-Oriented Optimization Search Policy

The abstraction enlarges the scheduling space for DNN computation and opens a new trade-off between memory footprint efficiency and latency efficiency.



When the storage of the system is sufficient, additional searches are made for the latency overhead of performing type conversions at each stage and the configuration with the shortest latency is selected

System Overview of Ladder/BitBLAS

Ladder

System for end2end optimizations

PyTorch ONNX

DFG of tTile-Operator — tType & tTile

Scheduling on top of Welder(OSDI'23) and Roller(OSDI'22) — Transformations

Code Generation on top of TVM — PTX/ISA Opt.

CUDA → NVIDIA GPU

HIP → AMD GPU

NVIDIA GPU

AMD GPU

Modern Accelerators ......

Has been used for 1.58bits Model

BitBLAS

Example Usage

Runtime Kernel Library

```
matmul_config = bitblas.MatmulConfig(
        A_dtype="float16",
        W_dtype="int4",
        accum_dtype="float16",
        out_dtype="float16", ...
)
Matmul = bitblas.matmul(matmul_config)
```

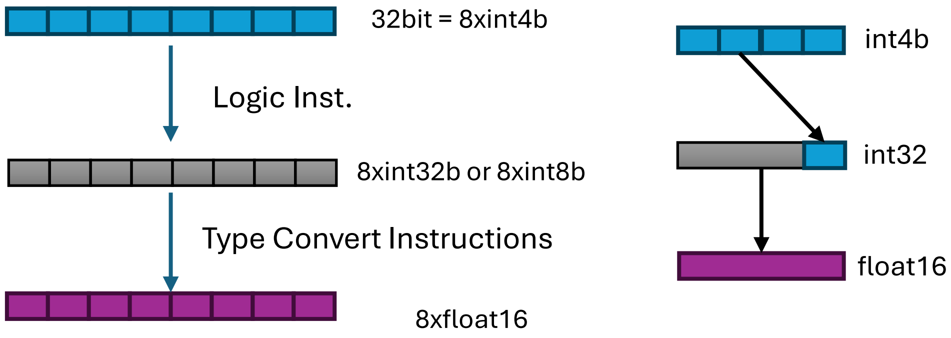
Operator ⇨ Auto Tensorization / Layout Propataion / Hardware aware Tuning ⇨ CUDA Source ⇨ Executable

Now integrated into vLLM, AutoGPTQ, EfficientQAT, HQQ!

Shipped to Ads team for BitNet deployment!

Vectorized Dequantization with Weight Interleave

Conventional Dequantization



Introducing a certain amount of computation can become a bottleneck in performance Especially on devices with fewer bits and weaker compute cores (for example, cuda core on a100).

Who Says Elephants Can't Run:
Bringing Large Scale MoE Models into Cloud Scale Production

$$\boxed{(-1)^{sign} * 2^{exponent\ -15}} * \left(1 + \frac{fraction}{1024}\right)$$

MAGIC Number

$$\boxed{1024} * \left(1 + \frac{fraction}{1024}\right) = 1024 + fraction$$

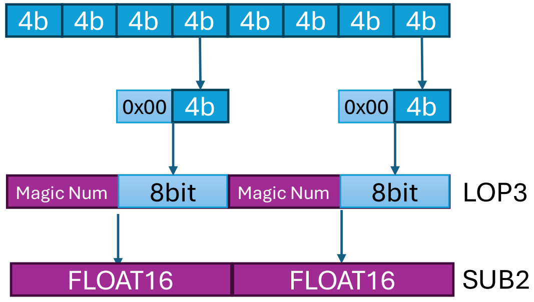
For example, for number 3, we can add 1024 → 0x6400 | 0x0003

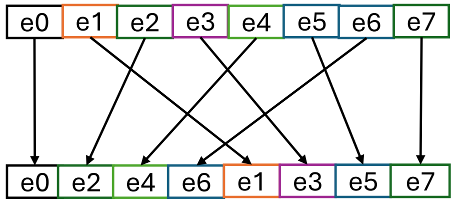
1024 * (1 + 3/ 1024) = 1024 + 3

And to get float 3.0 → (1024 + 3) - 1024

Vectorized Dequantization

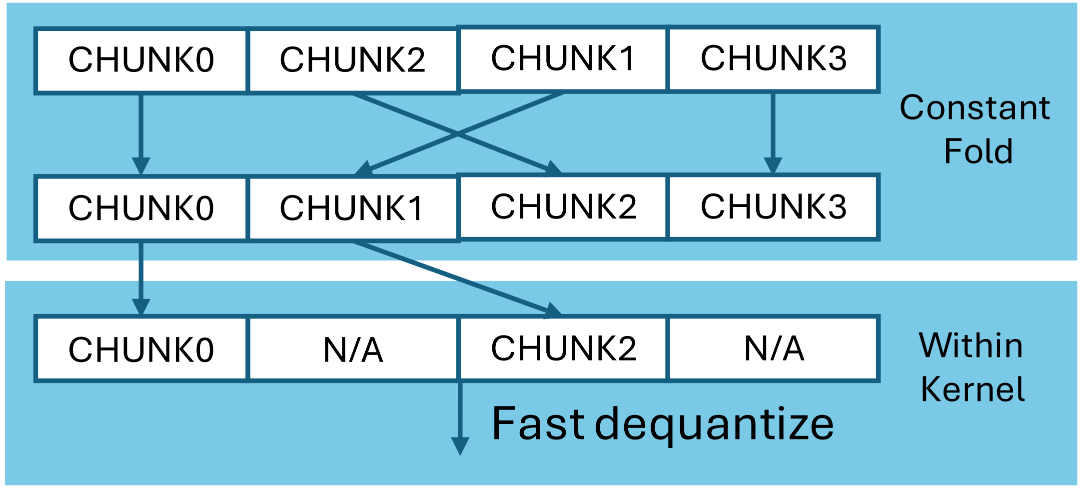
Tutorial: Fast Dequantize





While it's hard to be extended into fewer bits

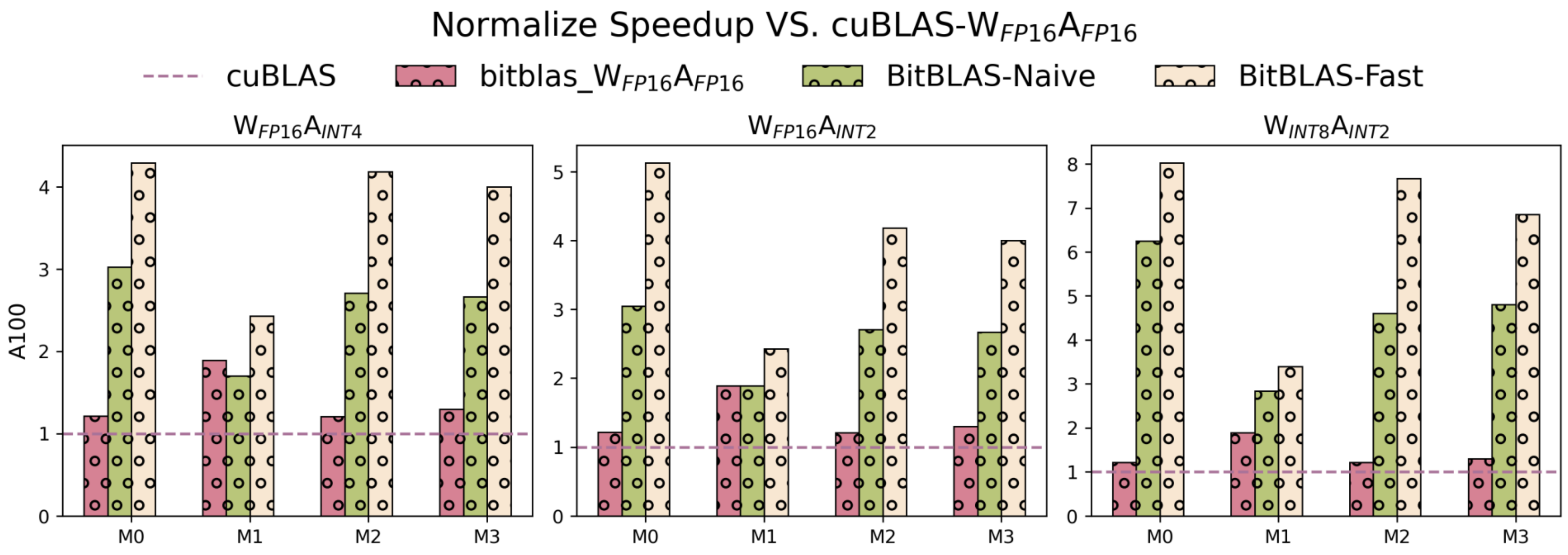
BitBLAS: Chunk Level Interleave



Extension for BitBLAS To Support More Fewer Bits (1/2b to 8/16b)

And we also provide Other Fast Dequantize: FP8->FP16

Fast Decoding Performance on A100 GPU

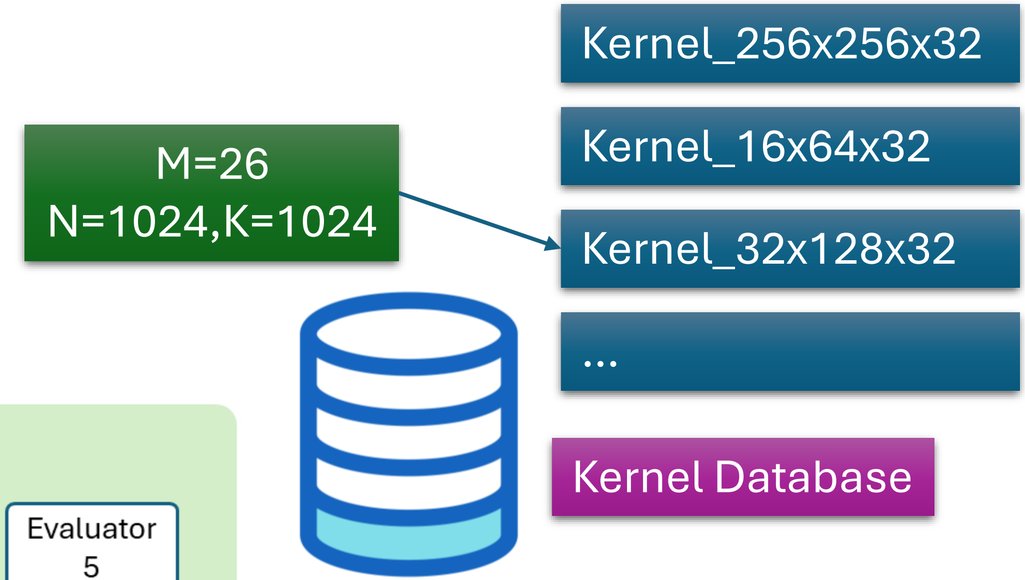


Fast and Efficient Dynamic Kernel Tuning

Building a universal library is challenging.

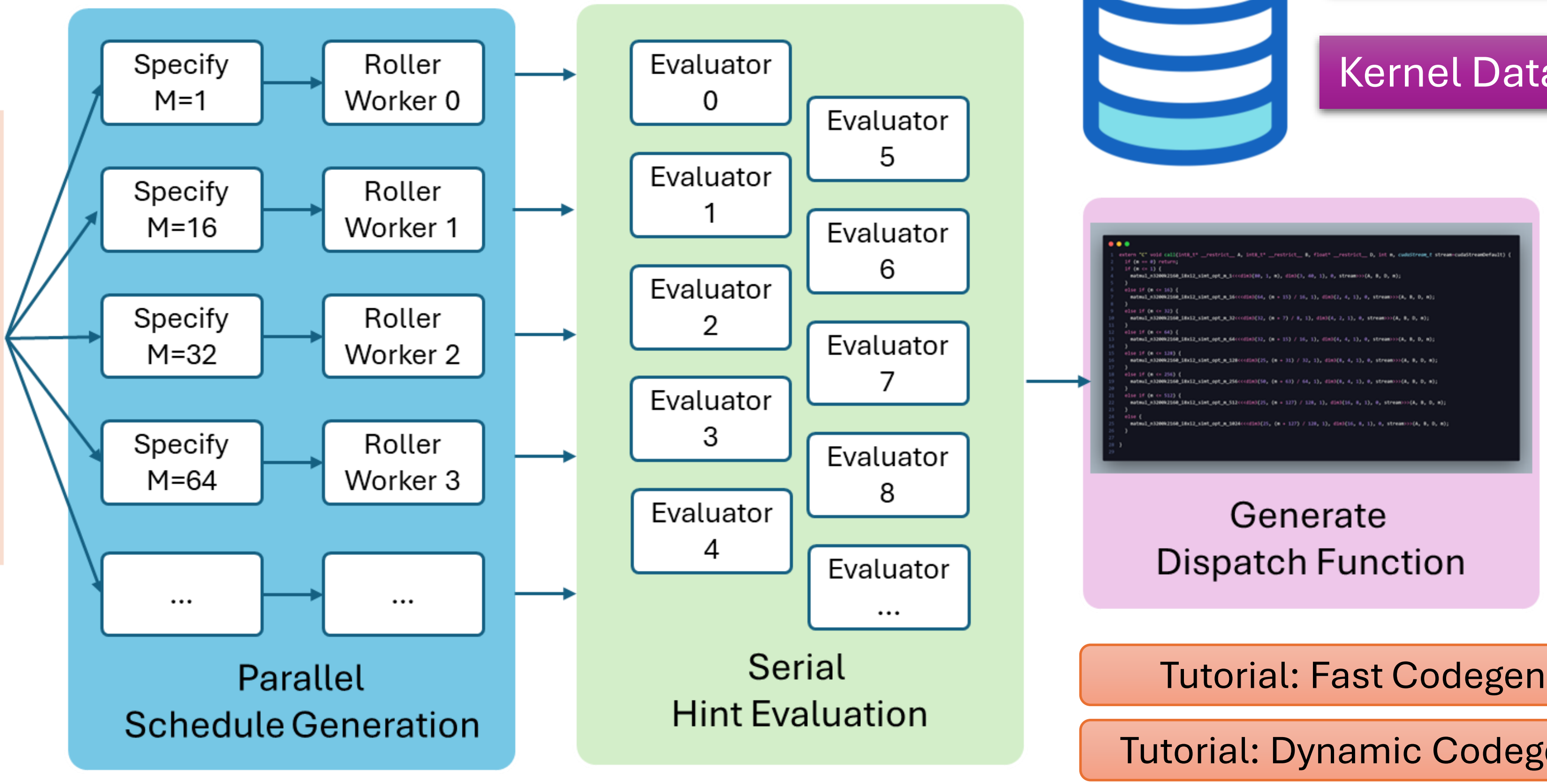
Different shapes (architectures) prefer different kernel config.

Only one dimension is dynamic within
LLM Compute Workload.



OP Description
With Dynamic Range

```
matmul_config = bitblas.MatmulConfig(
        M=[1, 16, 32, 64, 128, 256, 512],
        K=3200,
        N=8640,
        A_dtype="float16",
        W_dtype="int4",
        accum_dtype="float16",
        out_dtype="float16",
        ...
)
Matmul = bitblas.matmul(matmul_config)
```



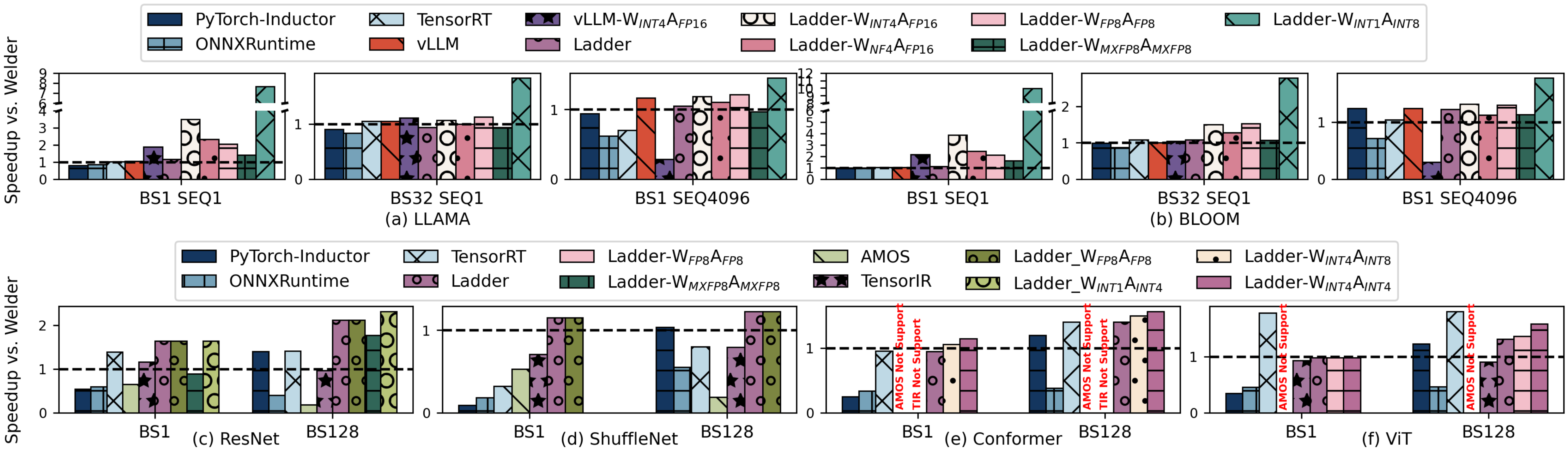
Tutorial: Fast Codegen

Tutorial: Dynamic Codegen

```
extern "C" void call(int8_t* __restrict__ A, int8_t* __restrict__ B, float* __restrict__ D, int m, cudaStream_t stream=cudaStreamDefault) {
  if (m == 0) return;
  if (m <= 1) {
    matmul_n3200k2160_i8xi2_simt_opt_m_1<<<dim3(80, 1, m), dim3(3, 40, 1), 0, stream>>>(A, B, D, m);
  }
  else if (m <= 16) {
    matmul_n3200k2160_i8xi2_simt_opt_m_16<<<dim3(64, (m + 15) / 16, 1), dim3(2, 4, 1), 0, stream>>>(A, B, D, m);
  }
  else if (m <= 32) {
    matmul_n3200k2160_i8xi2_simt_opt_m_32<<<dim3(32, (m + 7) / 8, 1), dim3(4, 2, 1), 0, stream>>>(A, B, D, m);
  }
  else if (m <= 64) {
    matmul_n3200k2160_i8xi2_simt_opt_m_64<<<dim3(32, (m + 15) / 16, 1), dim3(4, 4, 1), 0, stream>>>(A, B, D, m);
  }
  else if (m <= 128) {
    matmul_n3200k2160_i8xi2_simt_opt_m_128<<<dim3(25, (m + 31) / 32, 1), dim3(8, 4, 1), 0, stream>>>(A, B, D, m);
  }
  else if (m <= 256) {
    matmul_n3200k2160_i8xi2_simt_opt_m_256<<<dim3(50, (m + 63) / 64, 1), dim3(8, 4, 1), 0, stream>>>(A, B, D, m);
  }
  else if (m <= 512) {
    matmul_n3200k2160_i8xi2_simt_opt_m_512<<<dim3(25, (m + 127) / 128, 1), dim3(16, 8, 1), 0, stream>>>(A, B, D, m);
  }
  else {
    matmul_n3200k2160_i8xi2_simt_opt_m_1024<<<dim3(25, (m + 127) / 128, 1), dim3(16, 8, 1), 0, stream>>>(A, B, D, m);
  }

}

```

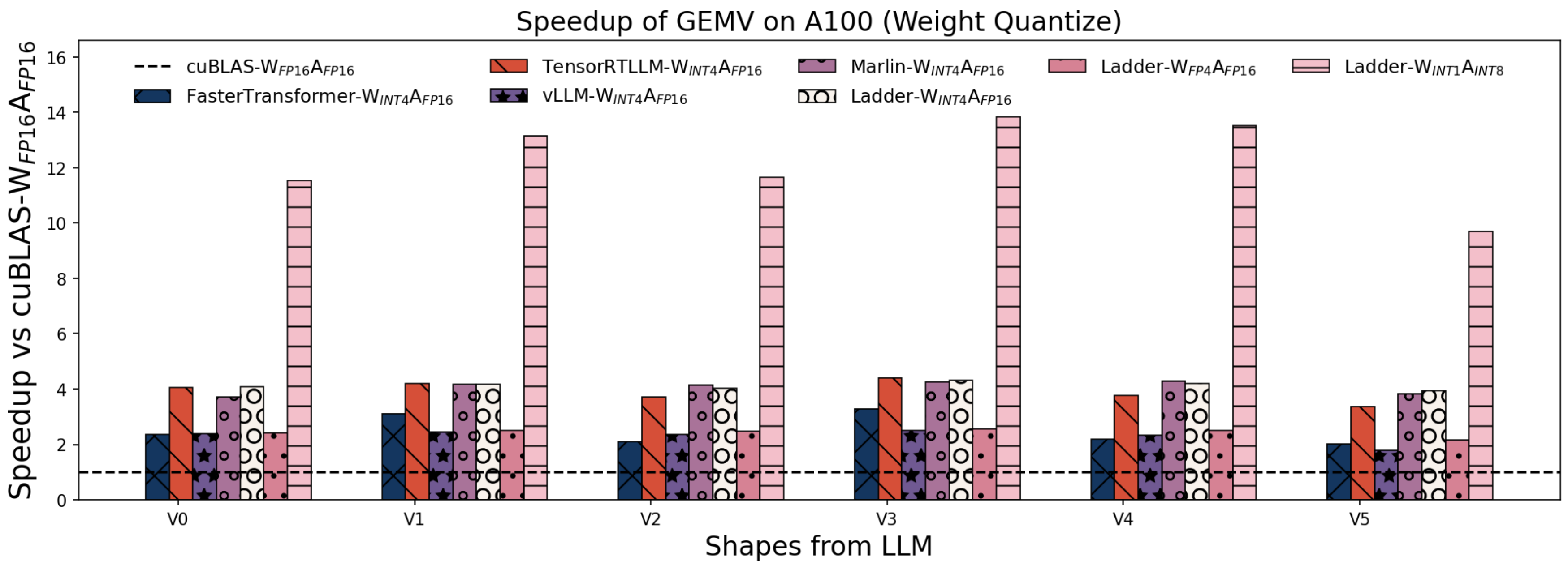
End2End Performance of Ladder

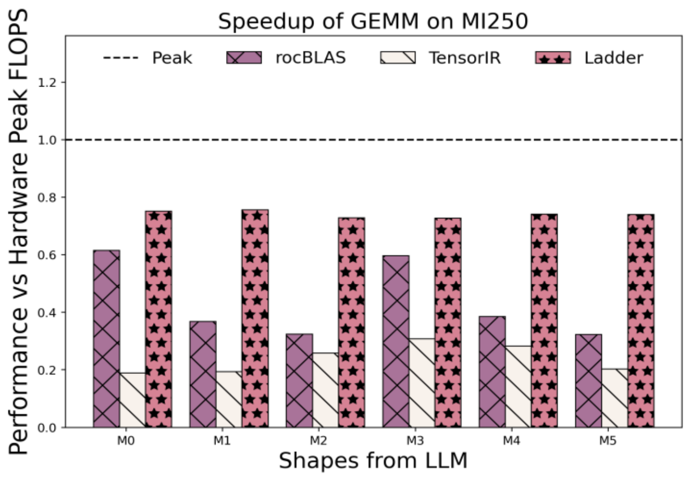
A100 80G

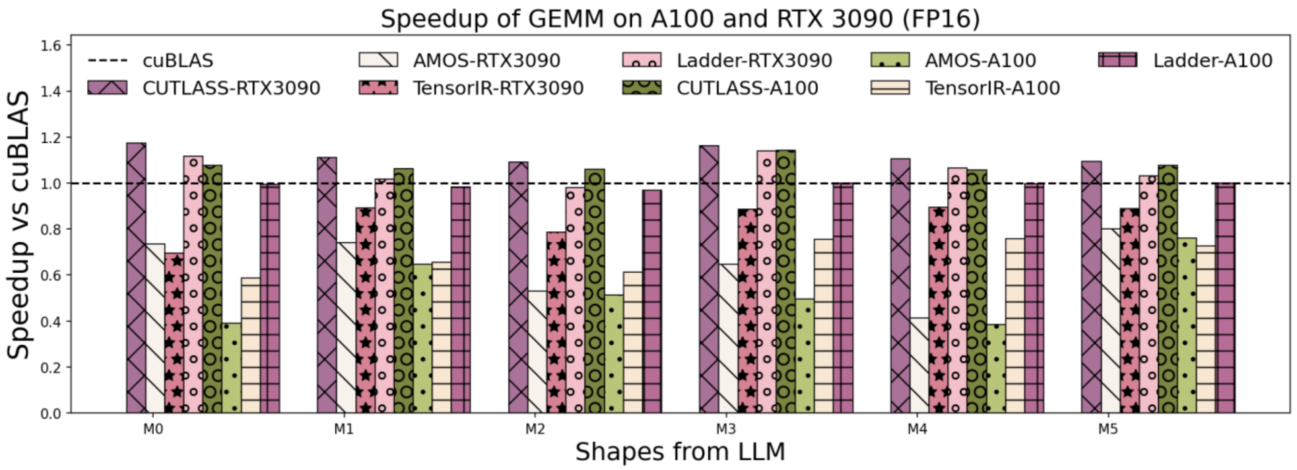


- $W_{FP16}A_{FP16}$: ~ **1.1x/1.1x** avg. speedup over Welder/TensorRT
- $W_{INT4}A_{FP16}$ (GPTQ) ~ **2.3x** avg. speedup over vLLM-$W_{INT4}A_{FP16}$
- $W_{INT1}A_{INT8}$ (BitNet): up to **8.8x** speedup over Ladder- $W_{FP16}A_{FP16}$ (on BLOOM-176B-BS1SEQ1)

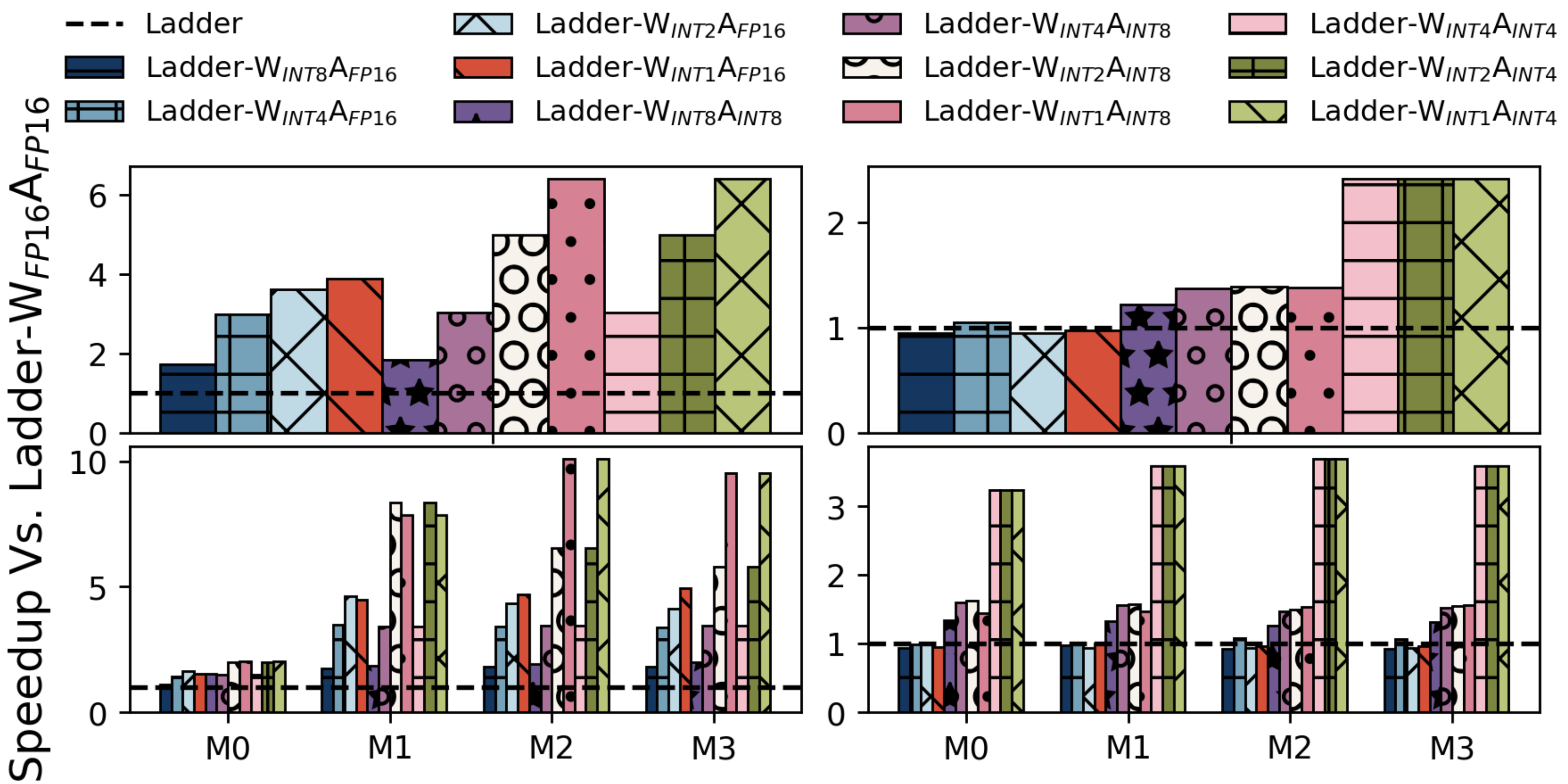
Operator Performance of BitBLAS







System Performance Scaling Up



(a) BS1 SEQ1 (b) BS1 SEQ4096

Decode: Memory Intensive

Quantized kernels can benefit from reduced memory bandwidth usage.

Prefill Compute Intensive

Quantized kernels can benefit from more efficient hardware instructions.

- BS1 SEQ1: bounded by memory bw., up to 6.4x speedup (10.1x speedup on kernel)
- BS1 SEQ4096: bounded by tensor core, up to 2.4x speedup (3.7x speedup on kernel)

Summary

We proposed universe Tensor Abstractions and Schedule Primitives to enable ml compiler explore tensor scheduling

We proposed a hardware-aligned Memory Layout Propagation Strategy to auto inference Memory Layout and eliminate the overhead.

We proposed a bit-nearest and instruction aligned tensorization strategy.

We introduce a Latency-oriented Search Policy

We designed Ladder and BitBLAS.

Challenges From The Community

Though bitblas has been integrated into vLLM, AutoGPTQ, HQQ

Kernel Compilation takes too much time even though with Kernel Database.

Runtime Kernel Library may lead to uncomfortable user experience.

Schedule Based Implementations make it hard for developers to extend BitBLAS.

Schedule Based Implementation is hard to describe complex ops(like stream-k, flash Atten)

We're leveraging TileLang to handle issue 2 and 3 as triton is hard to describe dequant related items.

Tutorials

Microsoft

Thanks for watching

Oct 26, 2024

More info, reproduce, reach:

https://github.com/microsoft/BitBLAS

More detail, download:

OSDI 2024’ Ladder