Ladder: Enabling Efficient Low-Precision Deep Learning Computing through Hardware-aware Tensor Transformation

Lei Wang †◊, Lingxiao Ma ◊, Shijie Cao ◊, Quanlu Zhang ◊, Jilong Xue ◊, Yining Shi ‡◊ Ningxin Zheng ◊, Ziming Miao ◊, Fan Yang ◊, Ting Cao ◊, Yuqing Yang ◊, Mao Yang◊

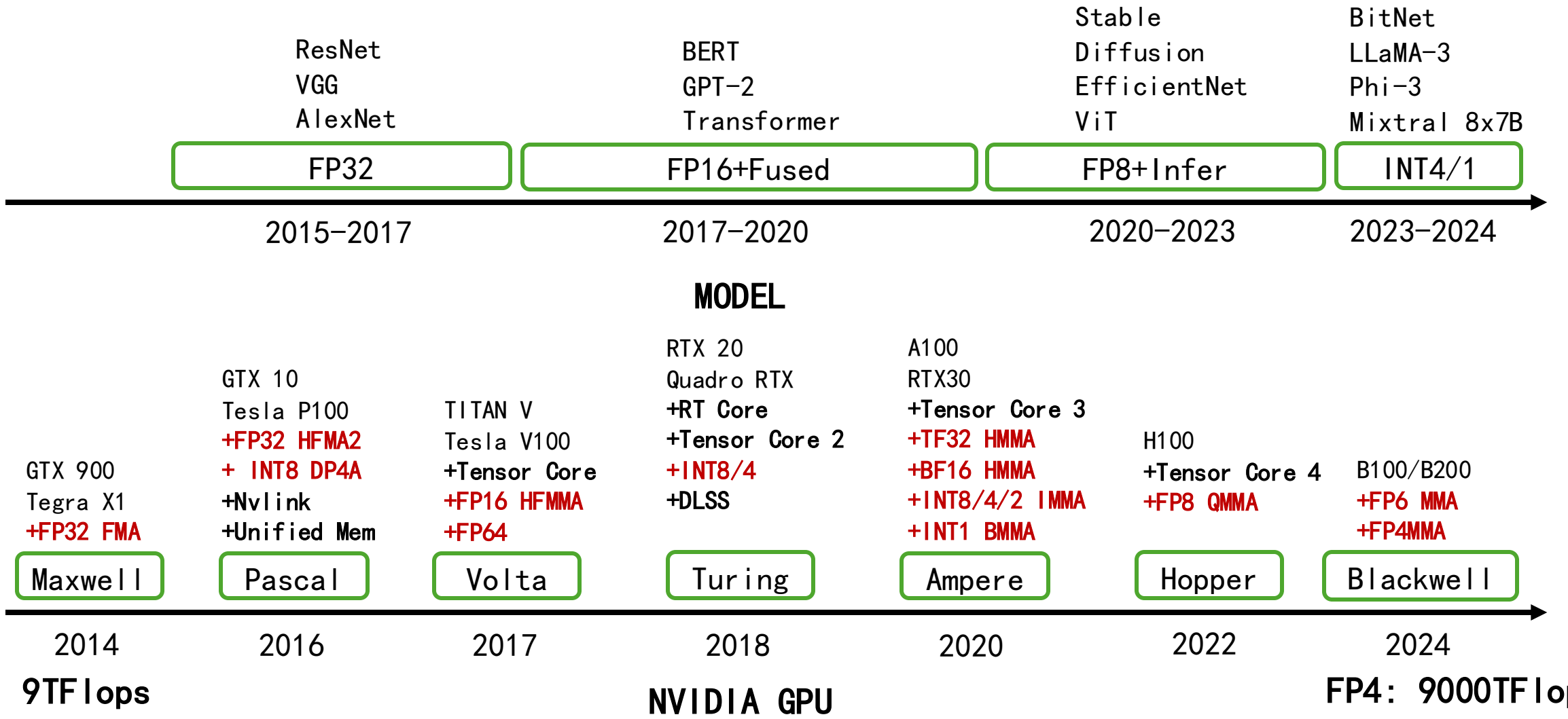
† University of Chinese Academy of Sciences

‡ Peking University

◊ Microsoft Research

Presenter: Chengru Yang

Larger Scale, Fewer Bits



Supporting Low-Precision DNN Computing

低精度训练趋势

- 使用FP16, INT8/4/2/1来训练
- 分组精度缩放
  - MXFP, 分组量化
- 混合精度运算
  - FP16 x INT4/NF4, INT8 x INT1

加速器面临困境

- 需使用代理类型：例如，用 FP16 模拟 FP4
- 低精度计算低效问题
  - FP16/INT8 矩阵乘法的平均利用率<60%

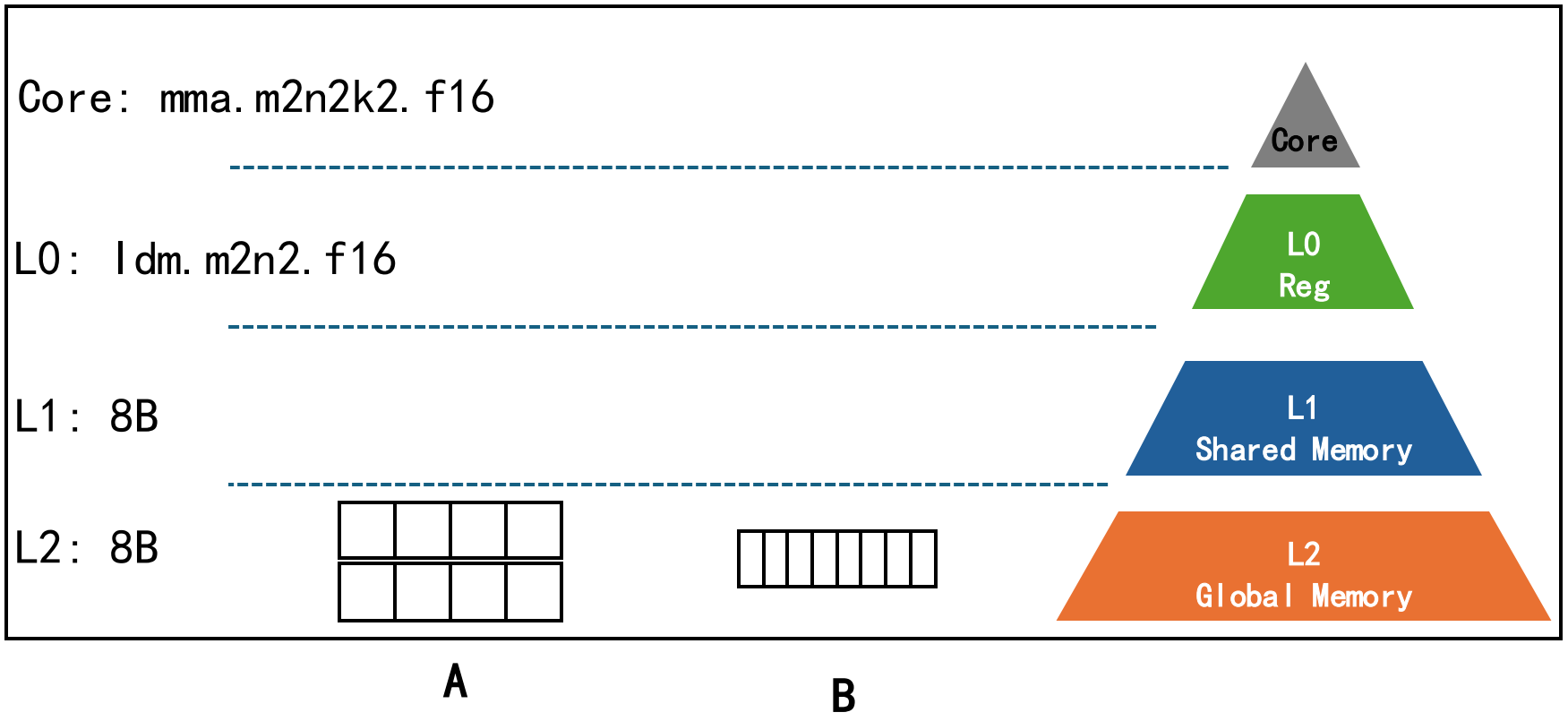
Data Type	W@FP16 / A@FP16			W@INT8 / A@INT8			W@FP8 / A@FP8	W@NF4 / A@FP16
GPU	V100	A100	MI250	V100	A100	MI250	V100/A100/MI250	
cuBLAS	78%	87%	X	X	68%	X	X	X
rocBLAS	X	X	46%	X	X	75%	X	X
AMOS	64%	38%	X	X	45%	X	X	X
TensorIR	67%	56%	22%	X	X	X	X	X
Roller	50%	70%	29%	X	X	X	X	X

MatMul在不同模型参数精度上和GPU上的利用率

低精度计算不断高速演化使得硬件进化难以跟上

Supporting Low-Precision DNN Computing

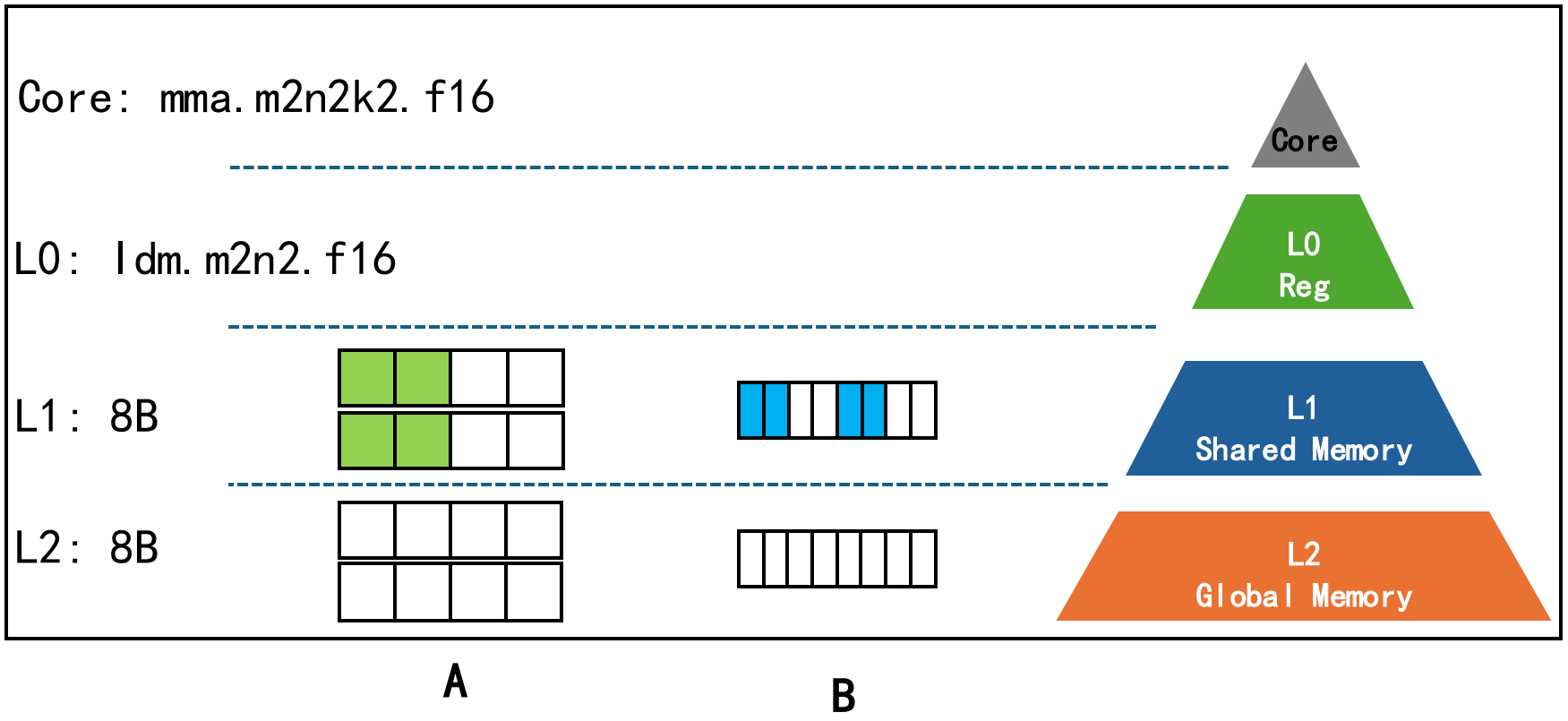
Matrix multiplication: C[M,N]@FP16=A[M,K]@FP16 X B[N,K]@FP8, M=2, N=2, K=4



[X,Y]@Z
表示:
shape是[X,Y]的
每一个元素大小是
Z的一组同质元素

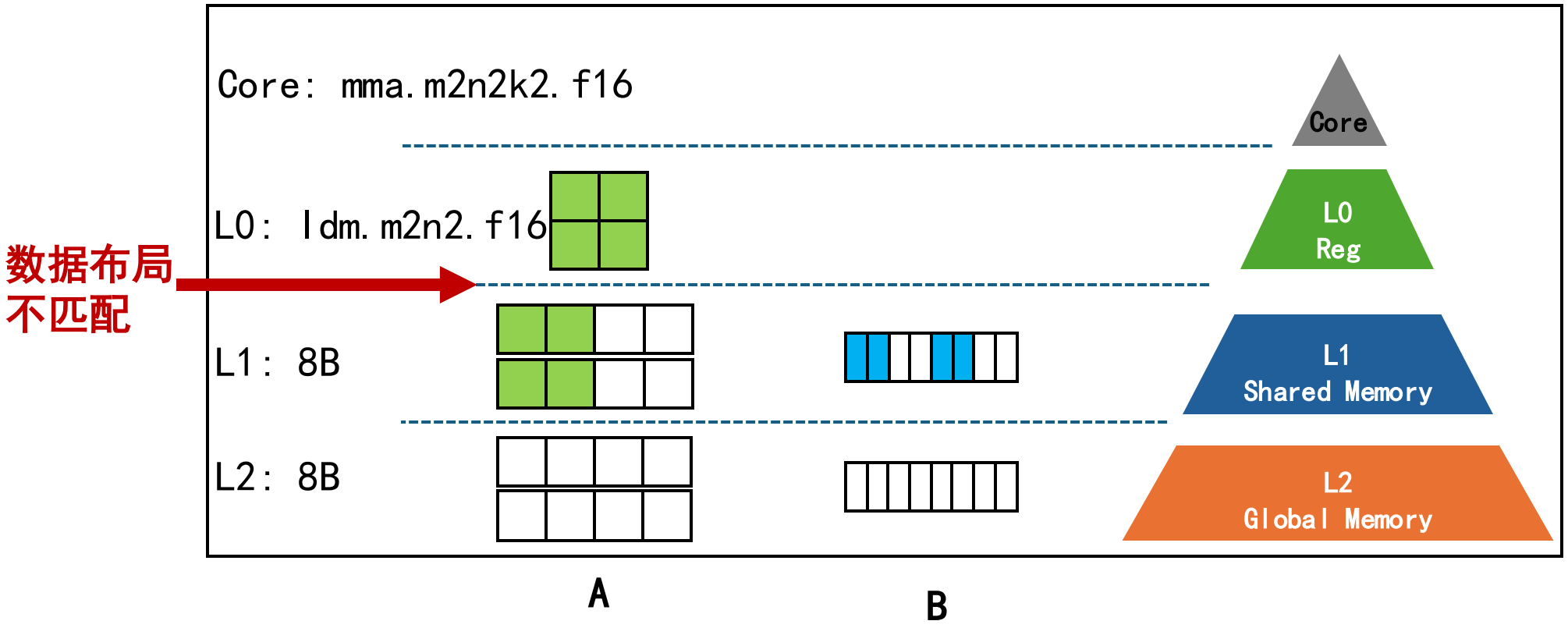
Supporting Low-Precision DNN Computing

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



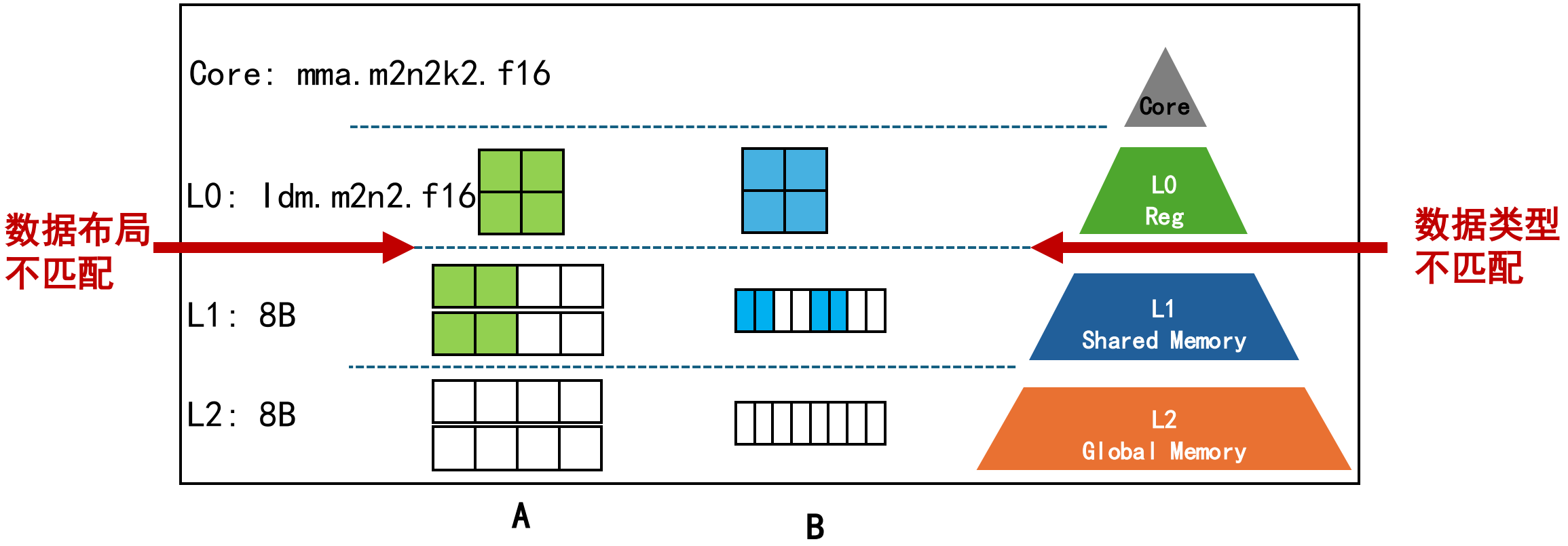
Supporting Low-Precision DNN Computing

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



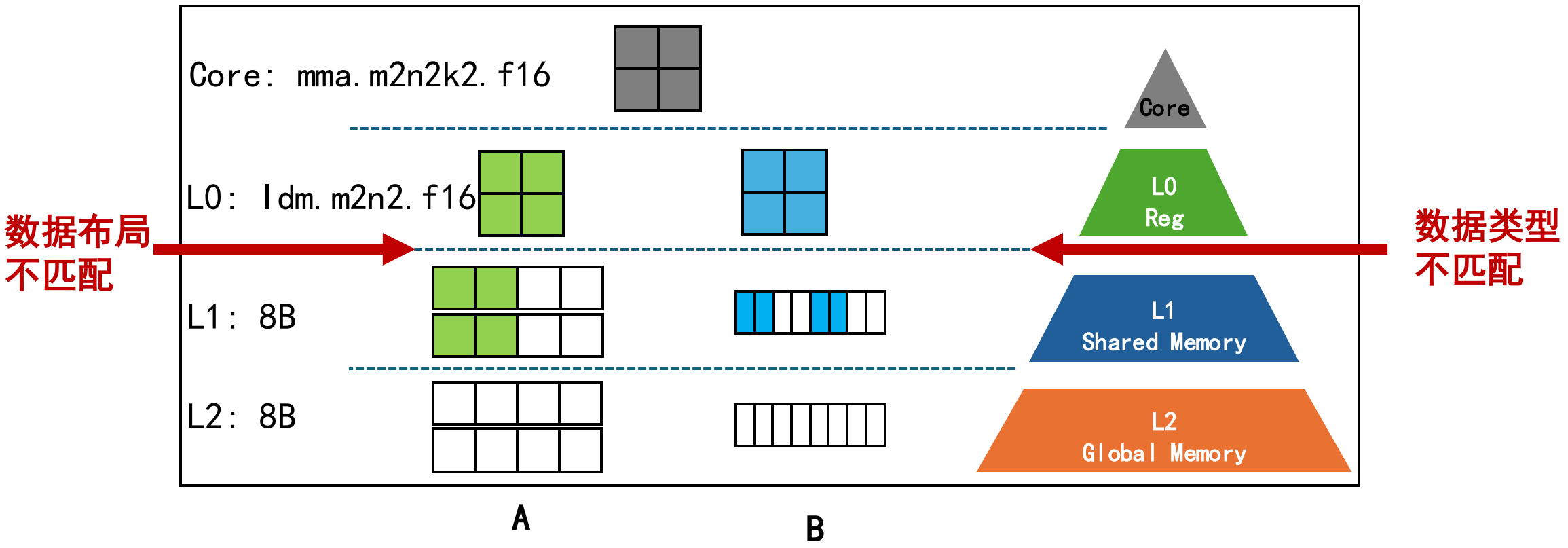
Supporting Low-Precision DNN Computing

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



Supporting Low-Precision DNN Computing

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



Insights

存储与计算分离：硬件加速器虽缺乏对自定义数据类型的计算指令，但其内存系统可通过将数据转换为固定位宽的不透明块来存储任意类型。

无损类型转换：多数自定义数据类型可无损转换为硬件支持的标准类型（如 NF4 张量可通过 FP16/FP32 运算计算）。

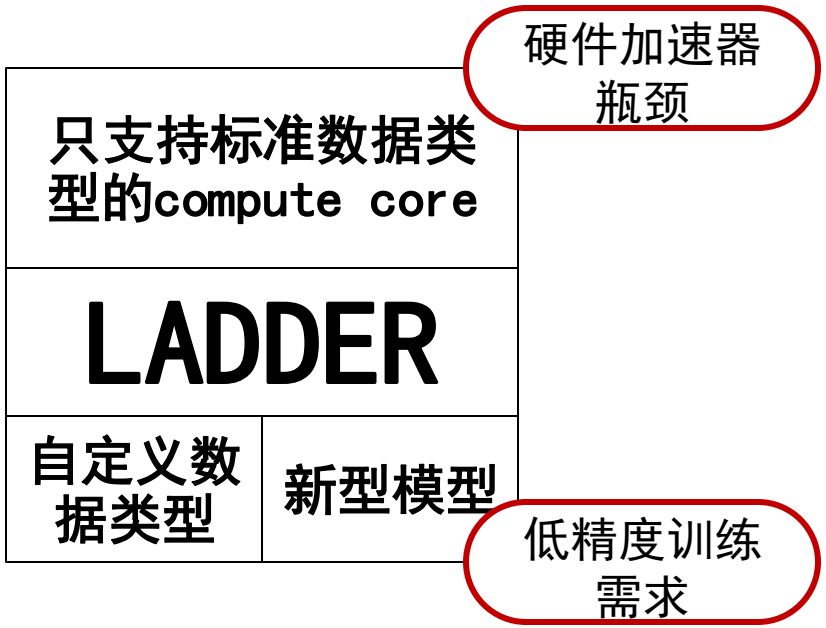
Insights

存储与计算分离：硬件加速器虽缺乏对自定义数据类型的计算指令，但其内存系统可通过将数据转换为固定位宽的不透明块来存储任意类型。

无损类型转换：多数自定义数据类型可无损转换为硬件支持的标准类型（如 NF4 张量可通过 FP16/FP32 运算计算）。

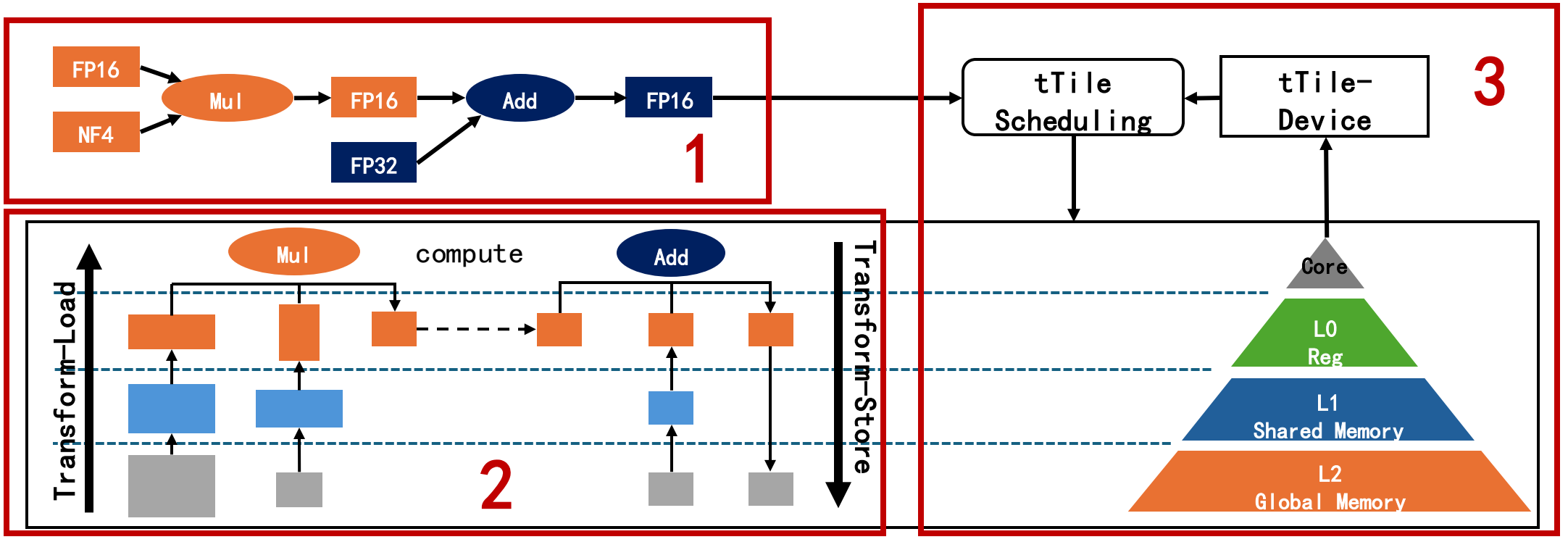
支持自定义数据类型进行tensor的存储与传输

通过transformation在标准数据类型下执行计算



LADDER Design

- 1 tTile Abstraction
- 2 tTile Transformation
- 3 Hardward-Aware tTile-Graph Scheduling



tTile Abstraction

tType

```
Class tType {
  TileShape shape;
  size_t nElembits;
  struct metadata;
  map<TileType, c_func> c_Types;
};
```

tType表示由一组同质元素组成的数据类型。
shape表示n维数组的形状，
nElemBits表示存储一个元素需要几位，
同时这些元素共享相同的metadata，
c_Types表示可以通过c_func函数无损的将其转换为另一个tType。

举例：
FP16: shape=[1] nElembits=16
NF4: shape=[1] nElembits=4 在metadata中共享value map
OCP-MXFP8: shape=[32] nElembits=8 在metadata中共享 scaling factor

tTile Abstraction

tType

```
Class tType {
  TileShape shape;
  size_t nElembits;
  struct metadata;
  map<TileType, c_func> c_Types;
};
```

tTile

```
Class tTile {
  TileShape shape;
  tType type;
  struct metadata;
};
```

tTile是一组具有相同数据类型type和n维数组形状布局的同质元素。
shape表示n维tensor的形状，
type表示对应的tType，
tTile中的元素共享metadata。

举例：
[16,16]@FP16：16x16形状，每一个元素为FP16
[16,2]@16B：16x2形状，每一个元素大小为16B
[32]@4B：32形状，每一个元素大小为4B

tTile Abstraction

tType

```
Class tType {
  TileShape shape;
  size_t nElembits;
  struct metadata;
  map<TileType, c_func> c_Types;
};
```

tTile

```
Class tTile {
  TileShape shape;
  tType type;
  struct metadata;
};
```

tTile-Operator

```
Class tTile-Operator {
  TensorExpr expr;
  TileShape shape;
  vector<tTile> get_input_tTiles();
  vector<tTile> get_output_tTiles();
  void compute();
};
```

tTile-Operator表示对形状为 shape 的元素的张量计算任务。

get_input_tTiles()和get_output_tTiles()分别返回此计算任务得输入和输出tTile,

compute()执行张量表达式expr中定义得输入和输出tTile得计算。

tTile Abstraction

tType

```
Class tType {
  TileShape shape;
  size_t nElembits;
  struct metadata;
  map<TileType, c_func> c_Types;
};
```

tTile

```
Class tTile {
  TileShape shape;
  tType type;
  struct metadata;
};
```

tTile-Operator

```
Class tTile-Operator {
  TensorExpr expr;
  TileShape shape;
  vector<tTile> get_input_tTiles();
  vector<tTile> get_output_tTiles();
  void compute();
};
```

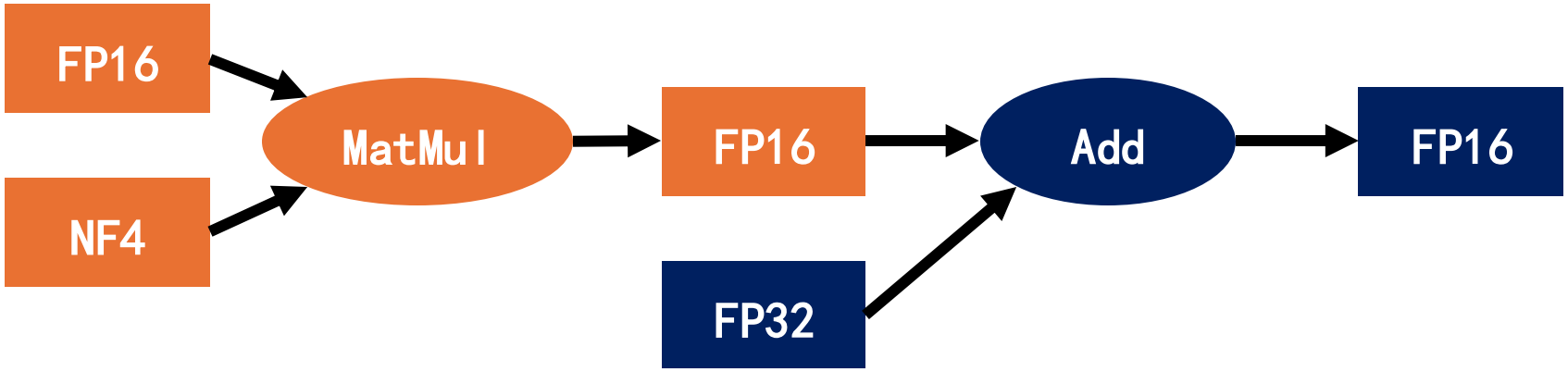
例子：

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

FP16 tensor A和NF4 tensor B的MatMul，并且以FP32作为累加类型，输出一个FP16类型的张量C[32,32]

tTile Graph

通过基于 tTile 的细粒度表示，DNN 模型可以被表示为一个细粒度的 tTile-graph



例子：

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

FP16 tensor A和NF4 tensor B的MatMul，并且以FP32作为累加类型，输出一个FP16类型的张量C[32,32]

tTile-based Hardware Abstraction

LADDER将硬件加速器抽象为一个由多层tTile描述的结构，tTile-device。
每一层可能是内存或者计算单元：

1. 其需求通过一个特定粒度的形状来描述，表示为tTile。
2. 对应的特定粒度使用tType来表示。

下面以NVIDIA A100为例：

Transaction: 32B
tTile: [32]@1B

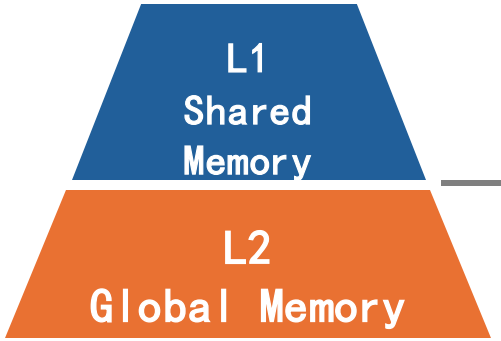
Global Memory以32B的粒度进行访问

tTile-based Hardware Abstraction

LADDER将硬件加速器抽象为一个由多层tTile描述的结构，tTile-device。
每一层可能是内存或者计算单元：

1. 其需求通过一个特定粒度的形状来描述，表示为tTile。
2. 对应的特定粒度使用tType来表示。

下面以NVIDIA A100为例：



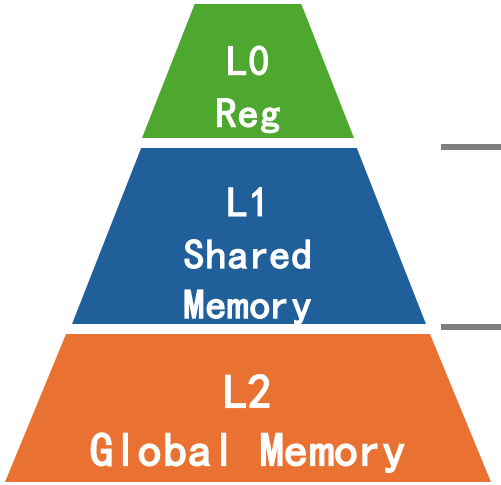
Transaction: 32*4B tTile: [32]@4B	Shared Memory以32个4B的bank为单位进行访问
Transaction: 32B tTile: [32]@1B	Global Memory以32B的粒度进行访问

tTile-based Hardware Abstraction

LADDER将硬件加速器抽象为一个由多层tTile描述的结构，tTile-device。
每一层可能是内存或者计算单元：

1. 其需求通过一个特定粒度的形状来描述，表示为tTile。
2. 对应的特定粒度使用tType来表示。

下面以NVIDIA A100为例：



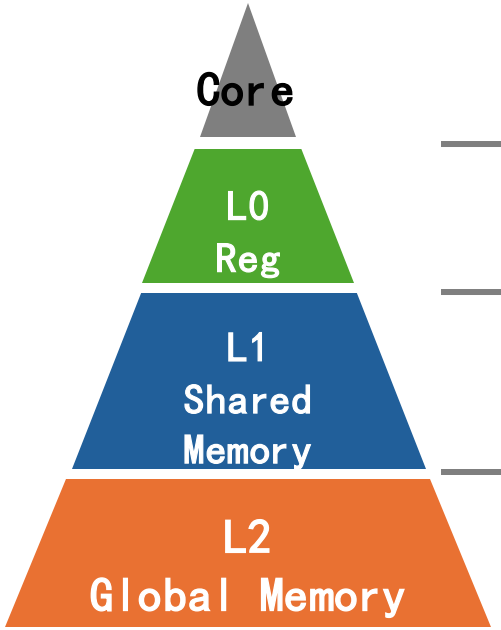
L0 Reg	Transaction: 16B tTile: [16,2]@16B	FP16 Core数据加载指令： ldmatrix.sync.aligned.m8n8.x4.shared.b16 以16B的粒度进行加载
L1 Shared Memory	Transaction: 32*4B tTile: [32]@4B	Shared Memory以32个4B的bank为单位进行访问
L2 Global Memory	Transaction: 32B tTile: [32]@1B	Global Memory以32B的粒度进行访问

tTile-based Hardware Abstraction

LADDER将硬件加速器抽象为一个由多层tTile描述的结构，tTile-device。
每一层可能是内存或者计算单元：

1. 其需求通过一个特定粒度的形状来描述，表示为tTile。
2. 对应的特定粒度使用tType来表示。

下面以NVIDIA A100为例：



tTile: [16,16]@F16	FP16 Core MMA指令： [16,16]或者[8,16]的两个粒度
Transaction: 16B tTile: [16,2]@16B	FP16 Core数据加载指令： ldmatrix.sync.aligned.m8n8.x4.shared.b16 以16B的粒度进行加载
Transaction: 32*4B tTile: [32]@4B	Shared Memory以32个4B的bank为单位进行访问
Transaction: 32B tTile: [32]@1B	Global Memory以32B的粒度进行访问

tTile Transformation

```
tTileslice(tTile_input,index,shape,out_shape);
tTilemap(tTile_input,map_func);
tTilepad(tTile_input,pad_shape,pad_value);
tTileconvert(tTile_input,new_tType);
```

- **Slice**

  Slice 原语 从 tTile_input 的地址索引中切出一组形状为 shape 的元素，并将其作为新的 tTile 返回，新 tTile 的形状为 out_shape。Slice 原语通常用于表示数据分块（tiling）。

- **Map**

  Map 原语 修改 tTile 中元素的布局。

- **Pad**

  Pad 原语 在 tTile_input 的每个边界上填充 pad_value，填充的形状由 pad_shape 指定。

- **Convert**

  Convert 原语 将 tTile_input 的 tType 转换为给定的 new_tType。

Hardware-AwaretTile-GraphScheduling

Algorithm 1: Hint-based layer-wise scheduling

Data: g: tTile-graph; D: tTile-device
Result: g_{ret}: scheduled tTile-graph

Function *GetDeviceHint(g, D)*:
    D = SelectDeviceConfig(g, D);
    HintShape = None, HintGranularity = None;
    for *layer* $\in$ *D.layers* **do**
        HintGranularity = LCM(HintGranularity, layer.tTile.type);
    for *layer* $\in$ *D.layers* **do**
        layer.tTile = `convert`(layer.tTile, HintGranularity);
        HintShape = LCM(HintShape, layer.tTile.shape);
    for *layer* $\in$ *D.layers* **do**
        layer.tTile.shape = HintShape;
    return D;
Function *ScheduleTransform(op,D,*l_{id}*)*:
    tTile$_h$ = op.tTile[l_{id}-1];
    tTile$_l$ = op.tTile[l_{id}];
    ScheduleSlice(tTile$_l$, tTile$_h$);
    if *LCM(tTile$_l$.shape, tTile$_h$.shape)* $\neq$ *tTile$_l$.shape* **then**
        SchedulePad(tTile$_l$, tTile$_h$, D);
    if *tTile$_l$.type* $\neq$ *tTile$_h$.type* **then**
        ScheduleConvert(tTile$_l$, tTile$_h$, D);
    if *nBits(tTile$_h$.shape[-1])* $\neq$ *nBits(D.layers[l_{id}].shape[-1])* **then**
        ScheduleMap(tTile$_l$, tTile$_h$, D);
    return op.transform[l_{id}-1];
Function *ScheduleConnectedGraph(g, D)*:
    D = GetDeviceHint(g, D);
    for l_{id} *in length(D.layers)* **do**
        for *op* $\in$ *g*[l_{id}] **do**
            op.tTile[l_{id}] = ScheduleTiling(op,D,l_{id});
            if $l_{id} > 0$ **then**
                op.transform[l_{id}] = ScheduleTransform(op,D,l_{id});
    g = ProfileAndSelect(g);
    return g;
Function *Schedule(g,D)*:
    g = ExtractConnectedGraph(g, D);
    for $g_{conn} \in g$ **do**
        g_{conn} = ScheduleConnectedGraph(g_{conn}, D);
    return g;

输入：表示为tTile-graph的DNN模型g和tTile device

Hardware-AwaretTile-GraphScheduling

Algorithm 1: Hint-based layer-wise scheduling

Data: g: tTile-graph; D: tTile-device
Result: g_{ret}: scheduled tTile-graph

Function *GetDeviceHint(g, D)*:
  D = SelectDeviceConfig(g, D);
  HintShape = None, HintGranularity = None;
  for *layer* $\in$ *D.layers* **do**
    HintGranularity = LCM(HintGranularity, layer.tTile.type);
  for *layer* $\in$ *D.layers* **do**
    layer.tTile = `convert`(layer.tTile, HintGranularity);
    HintShape = LCM(HintShape, layer.tTile.shape);
  for *layer* $\in$ *D.layers* **do**
    layer.tTile.shape = HintShape;
  return D;
Function *ScheduleTransform(op,D,*l_{id}*)*:
  tTile$_h$ = op.tTile[l_{id}-1];
  tTile$_l$ = op.tTile[l_{id}];
  ScheduleSlice(tTile$_l$, tTile$_h$);
  if *LCM(*$tTile_l$*.shape,* $tTile_h$*.shape)* $\neq$ $tTile_l$*.shape* **then**
    SchedulePad(tTile$_l$, tTile$_h$, D);
  if $tTile_l$*.type* $\neq$ $tTile_h$*.type* **then**
    ScheduleConvert(tTile$_l$, tTile$_h$, D);
  if *nBits(*$tTile_h$*.shape[-1])* $\neq$ *nBits(D.layers[*l_{id}*].shape[-1])* **then**
    ScheduleMap(tTile$_l$, tTile$_h$, D);
  return op.transform[l_{id}-1];
Function *ScheduleConnectedGraph(g, D)*:
  D = GetDeviceHint(g, D);
  for l_{id} *in length(D.layers)* **do**
    for *op* $\in$ *g[*l_{id}*]* **do**
      op.tTile[l_{id}] = ScheduleTiling(op,D,l_{id});
      if $l_{id} > 0$ **then**
        op.transform[l_{id}] = ScheduleTransform(op,D,l_{id});
  g = ProfileAndSelect(g);
  return g;
Function *Schedule(g,D)*:
  g = ExtractConnectedGraph(g, D);
  for $g_{conn} \in g$ **do**
    g_{conn} = ScheduleConnectedGraph(g_{conn}, D);
  return g;

输入：表示为tTile-graph的DNN模型g和tTile device

将g分为gconn子图，每个子图从最底层内存加载tTile到Core，再将结果存回最低层。可能是一个tTile-OP或一组可融合的tTile-OP

Hardware-AwaretTile-GraphScheduling

Algorithm 1: Hint-based layer-wise scheduling

Data: g: tTile-graph; D: tTile-device
Result: g_{ret}: scheduled tTile-graph

Function *GetDeviceHint(g, D)*:
  D = SelectDeviceConfig(g, D);
  HintShape = None, HintGranularity = None;
  for *layer* $\in$ *D.layers* **do**
    HintGranularity = LCM(HintGranularity, layer.tTile.type);
  for *layer* $\in$ *D.layers* **do**
    layer.tTile = `convert`(layer.tTile, HintGranularity);
    HintShape = LCM(HintShape, layer.tTile.shape);
  for *layer* $\in$ *D.layers* **do**
    layer.tTile.shape = HintShape;
  return D;
Function *ScheduleTransform(op,D,*l_{id}*)*:
  tTile$_h$ = op.tTile[l_{id}-1];
  tTile$_l$ = op.tTile[l_{id}];
  ScheduleSlice(tTile$_l$, tTile$_h$);
  if *LCM(*$tTile_l$*.shape,* $tTile_h$*.shape)* $\neq$ $tTile_l$*.shape* **then**
    SchedulePad(tTile$_l$, tTile$_h$, D);
  if $tTile_l$*.type* $\neq$ $tTile_h$*.type* **then**
    ScheduleConvert(tTile$_l$, tTile$_h$, D);
  if *nBits(*$tTile_h$*.shape[-1])* $\neq$ *nBits(D.layers[*l_{id}*].shape[-1])* **then**
    ScheduleMap(tTile$_l$, tTile$_h$, D);
  return op.transform[l_{id}-1];
Function *ScheduleConnectedGraph(g, D)*:
  D = GetDeviceHint(g, D);
  for l_{id} *in length(D.layers)* **do**
    for *op* $\in$ *g*[l_{id}] **do**
      op.tTile[l_{id}] = ScheduleTiling(op,D,l_{id});
      if $l_{id} > 0$ **then**
        op.transform[l_{id}] = ScheduleTransform(op,D,l_{id});
  g = ProfileAndSelect(g);
  return g;
Function *Schedule(g,D)*:
  g = ExtractConnectedGraph(g, D);
  for $g_{conn} \in g$ **do**
    g_{conn} = ScheduleConnectedGraph(g_{conn}, D);
  return g;

输入：表示为tTile-graph的DNN模型g和tTile device

计算Device Hint：
选择Core（比如优先FP16）
通过最小公倍数（LCM）来计算HintGranularity
确保跨层数据对齐，并且通过LCM来计算HintShape

将g分为gconn子图，每个子图从最底层内存加载tTile到Core，再将结果存回最低层。可能是一个tTile-OP或一组可融合的tTile-OP

Hardware-AwaretTile-GraphScheduling

Algorithm 1: Hint-based layer-wise scheduling

Data: g: tTile-graph; D: tTile-device
Result: g_{ret}: scheduled tTile-graph

Function *GetDeviceHint(g, D)*:
  D = SelectDeviceConfig(g, D);
  HintShape = None, HintGranularity = None;
  for *layer* $\in$ *D.layers* **do**
    HintGranularity = LCM(HintGranularity, layer.tTile.type);
  for *layer* $\in$ *D.layers* **do**
    layer.tTile = `convert`(layer.tTile, HintGranularity);
    HintShape = LCM(HintShape, layer.tTile.shape);
  for *layer* $\in$ *D.layers* **do**
    layer.tTile.shape = HintShape;
  return D;

Function *ScheduleTransform(op,D,*l_{id}*)*:
  $t\text{Tile}_h$ = op.tTile[l_{id}-1];
  $t\text{Tile}_l$ = op.tTile[l_{id}];
  ScheduleSlice($t\text{Tile}_l$, $t\text{Tile}_h$);
  if *LCM(*$tTile_l$*.shape,* $tTile_h$*.shape)* $\neq$ $tTile_l$*.shape* **then**
    SchedulePad($t\text{Tile}_l$, $t\text{Tile}_h$, D);
  if $tTile_l$*.type* $\neq$ $tTile_h$*.type* **then**
    ScheduleConvert($t\text{Tile}_l$, $t\text{Tile}_h$, D);
  if *nBits(*$tTile_h$*.shape[-1])* $\neq$ *nBits(D.layers[*l_{id}*].shape[-1])* **then**
    ScheduleMap($t\text{Tile}_l$, $t\text{Tile}_h$, D);
  return op.transform[l_{id}-1];

Function *ScheduleConnectedGraph(g, D)*:
  D = GetDeviceHint(g, D);
  for l_{id} *in length(D.layers)* **do**
    for *op* $\in$ *g*[l_{id}] **do**
      op.tTile[l_{id}] = ScheduleTiling(op,D,l_{id});
      if $l_{id} > 0$ **then**
        op.transform[l_{id}] = ScheduleTransform(op,D,l_{id});
  g = ProfileAndSelect(g);
  return g;

Function *Schedule(g,D)*:
  g = ExtractConnectedGraph(g, D);
  for $g_{conn} \in g$ **do**
    g_{conn} = ScheduleConnectedGraph(g_{conn}, D);
  return g;

输入：表示为tTile-graph的DNN模型g和tTile device

计算Device Hint：
选择Core（比如优先FP16）
通过最小公倍数（LCM）来计算HintGranularity
确保跨层数据对齐，并且通过LCM来计算HintShape

输入算子op，Device hint D，当前层lid
输出当前层调度的tTile Transformation

将g分为gconn子图，每个子图从最底层内存加载tTile到Core，再将结果存回最低层。可能是一个tTile-OP或一组可融合的tTile-OP

Hardware-AwaretTile-GraphScheduling

Algorithm 1: Hint-based layer-wise scheduling

Data: g: tTile-graph; D: tTile-device
Result: g_{ret}: scheduled tTile-graph

Function *GetDeviceHint(g, D)*:
  D = SelectDeviceConfig(g, D);
  HintShape = None, HintGranularity = None;
  for *layer* $\in$ *D.layers* **do**
    HintGranularity = LCM(HintGranularity, layer.tTile.type);
  for *layer* $\in$ *D.layers* **do**
    layer.tTile = `convert`(layer.tTile, HintGranularity);
    HintShape = LCM(HintShape, layer.tTile.shape);
  for *layer* $\in$ *D.layers* **do**
    layer.tTile.shape = HintShape;
  return D;

Function *ScheduleTransform(op,D,*l_{id}*)*:
  tTile$_h$ = op.tTile[l_{id}-1];
  tTile$_l$ = op.tTile[l_{id}];
  ScheduleSlice(tTile$_l$, tTile$_h$);
  if *LCM(*$tTile_l$*.shape,* $tTile_h$*.shape)* $\neq$ $tTile_l$*.shape* **then**
    SchedulePad(tTile$_l$, tTile$_h$, D);
  if $tTile_l$*.type* $\neq$ $tTile_h$*.type* **then**
    ScheduleConvert(tTile$_l$, tTile$_h$, D);
  if *nBits(*$tTile_h$*.shape[-1])* $\neq$ *nBits(D.layers[*l_{id}*].shape[-1])* **then**
    ScheduleMap(tTile$_l$, tTile$_h$, D);
  return op.transform[l_{id}-1];

Function *ScheduleConnectedGraph(g, D)*:
  D = GetDeviceHint(g, D);
  for l_{id} *in length(D.layers)* **do**
    for *op* $\in$ *g*[l_{id}] **do**
      op.tTile[l_{id}] = ScheduleTiling(op,D,l_{id});
      if $l_{id} > 0$ **then**
        op.transform[l_{id}] = ScheduleTransform(op,D,l_{id});
  g = ProfileAndSelect(g);
  return g;

Function *Schedule(g,D)*:
  g = ExtractConnectedGraph(g, D);
  for $g_{conn} \in g$ **do**
    g_{conn} = ScheduleConnectedGraph(g_{conn}, D);
  return g;

输入：表示为tTile-graph的DNN模型g和tTile device

计算Device Hint：
选择Core（比如优先FP16）
通过最小公倍数（LCM）来计算HintGranularity
确保跨层数据对齐，并且通过LCM来计算HintShape

输入算子op，Device hint D，当前层lid
输出当前层调度的tTile Transformation

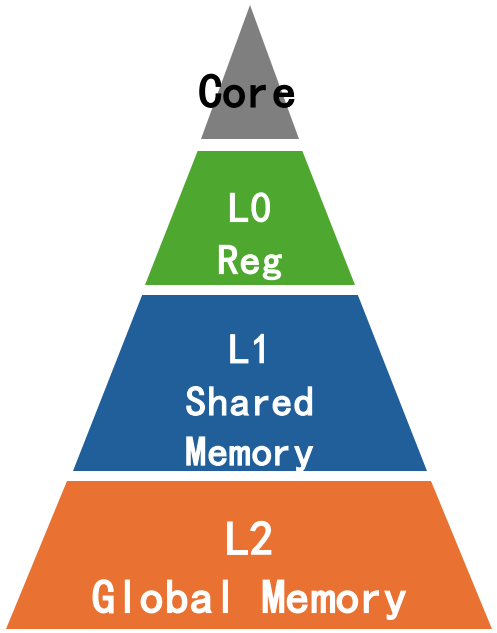
拼接子图
完善层间调度
性能分析
LADDER 提供了一种以 **延迟为导向的策略**，目标是最小化端到端延迟

将g分为gconn子图，每个子图从最底层内存加载tTile到Core，再将结果存回最低层。可能是一个tTile-OP或一组可融合的tTile-OP

Example

目标：A[32,63]@FP16 和 B[32,63]@NF4的MMA

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



tTile: [16,16]@F16

Transaction: 16B
tTile: [16,2]@16B

Transaction: 32*4B
tTile: [32]@4B

Transaction: 32B
tTile: [32]@1B

HintGranularity: ldmatrix使用的16B

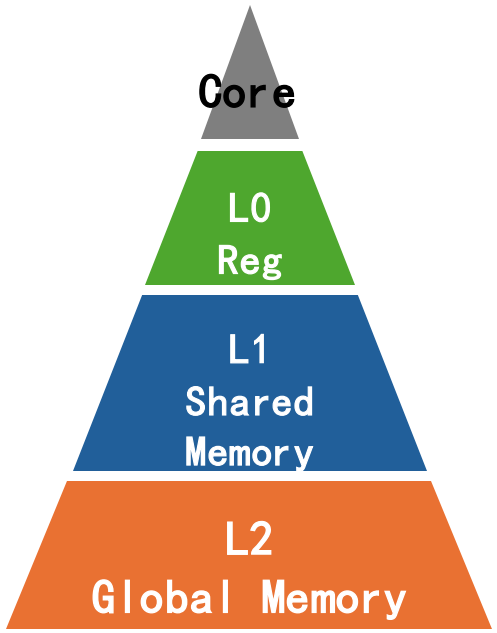
tTile-device for NVIDIA A100

计算Device Hint

Example

目标：A[32,63]@FP16 和 B[32,63]@NF4的MMA

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



tTile: [16,16]@F16

Transaction: 16B
tTile: [16,2]@16B

Transaction: 32*4B
tTile: [32]@4B

Transaction: 32B
tTile: [32]@1B

tTile-device for NVIDIA A100

HintGranularity：ldmatrix使用的16B

HintShape：
计算目标是使得L1和L2的Transaction都能对齐。
LCM(128, 32)=128B
innerdim = 128B/16B = 8
outerdim = 16*16*2B(FP16) / 128B = 4

最终算子分块为[4,8]的倍数且粒度为16B

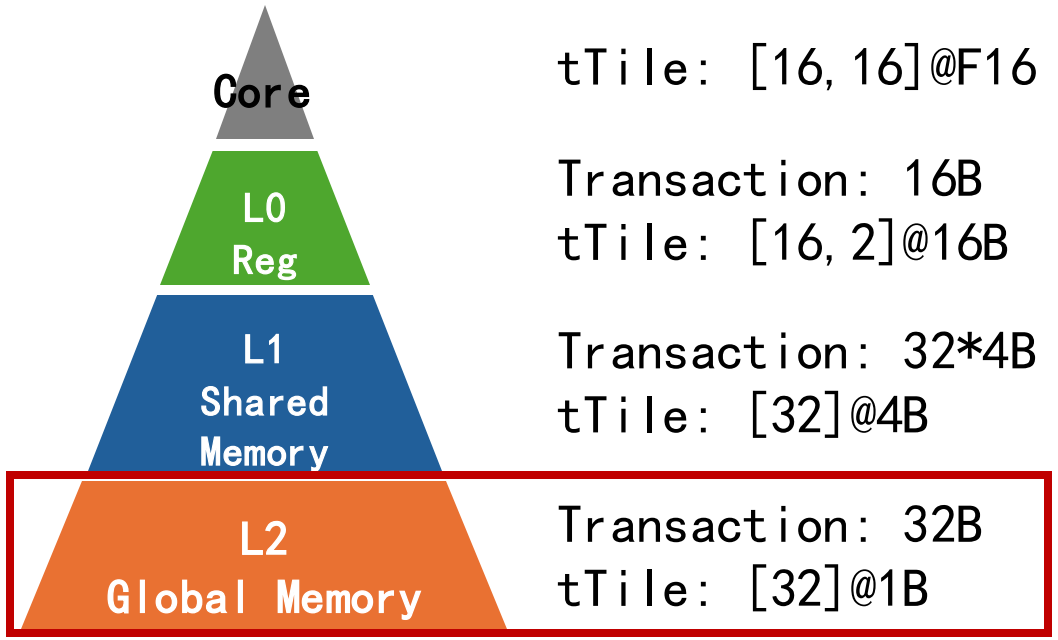
计算Device Hint

Example

目标：A[32,63]@FP16 和 B[32,63]@NF4的MMA

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

Device Hint: [4,8]@16B (size: 512B)



tTile-device for NVIDIA A100

Tensor大小：63*4bit*32 = 1008B
Hint size = 512B
sliceB: [16,63]@NF4
padB: [16,64]@NF4

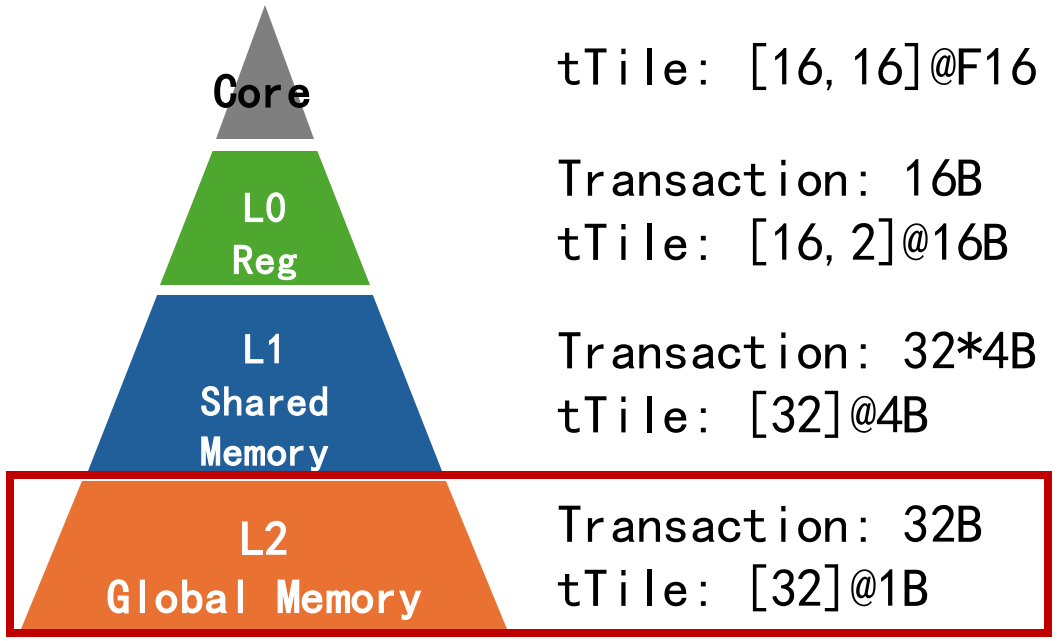
B[32,63]@NF4

Example

目标：A[32,63]@FP16 和 B[32,63]@NF4的MMA

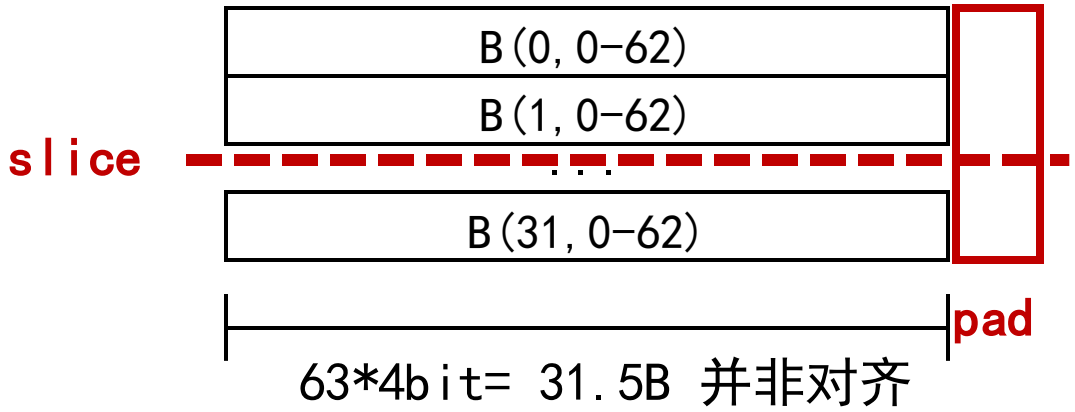
C=compute((M,N), lambda i,j:(sum((A[i,k]@FP16*B[j,k]@NF4)@FP32)@FP32)@FP16),
M=32, N=32, K=63

Device Hint: [4,8]@16B (size: 512B)



tTile-device for NVIDIA A100

Tensor大小：63*4bit*32 = 1008B
Hint size = 512B
sliceB: [16,63]@NF4
padB: [16,64]@NF4



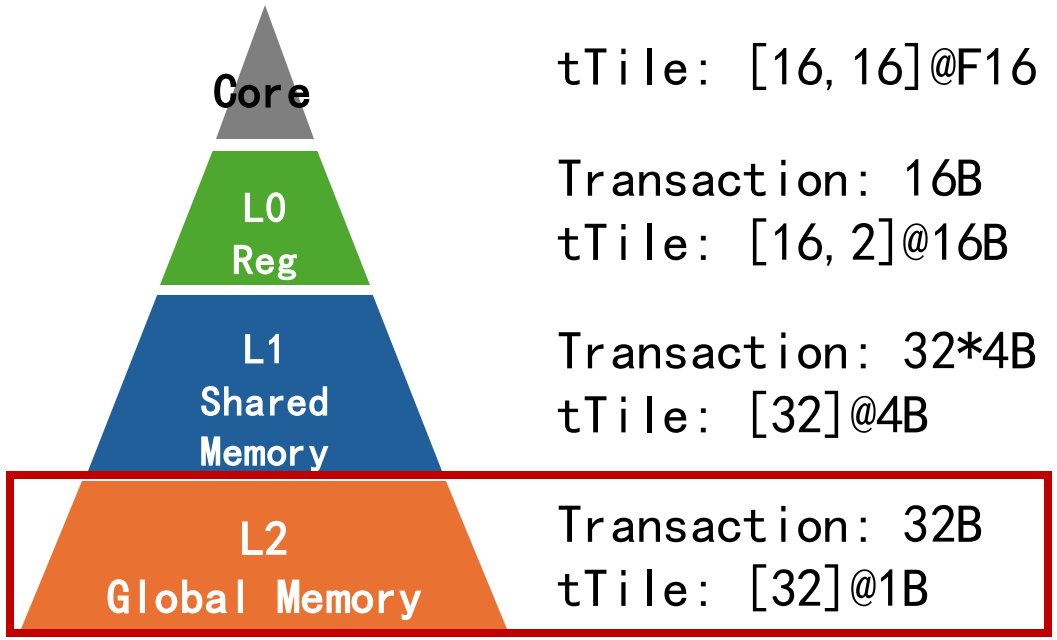
B[32,63]@NF4

Example

目标：A[32,63]@FP16 和 B[32,63]@NF4的MMA

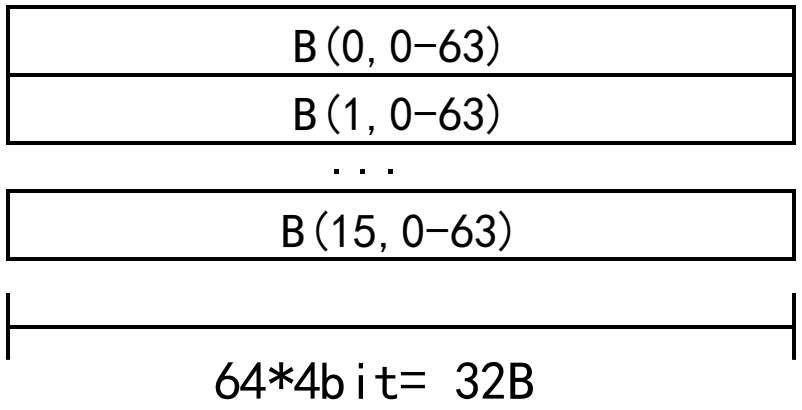
C=compute((M,N), lambda i,j:(sum((A[i,k]@FP16*B[j,k]@NF4)@FP32)@FP32)@FP16),
M=32, N=32, K=63

Device Hint: [4,8]@16B (size: 512B)



tTile-device for NVIDIA A100

Tensor大小：64*4bit*16 = 512B
Hint size = 512B
NF4≠FP16
convertB: [16,64]@FP16



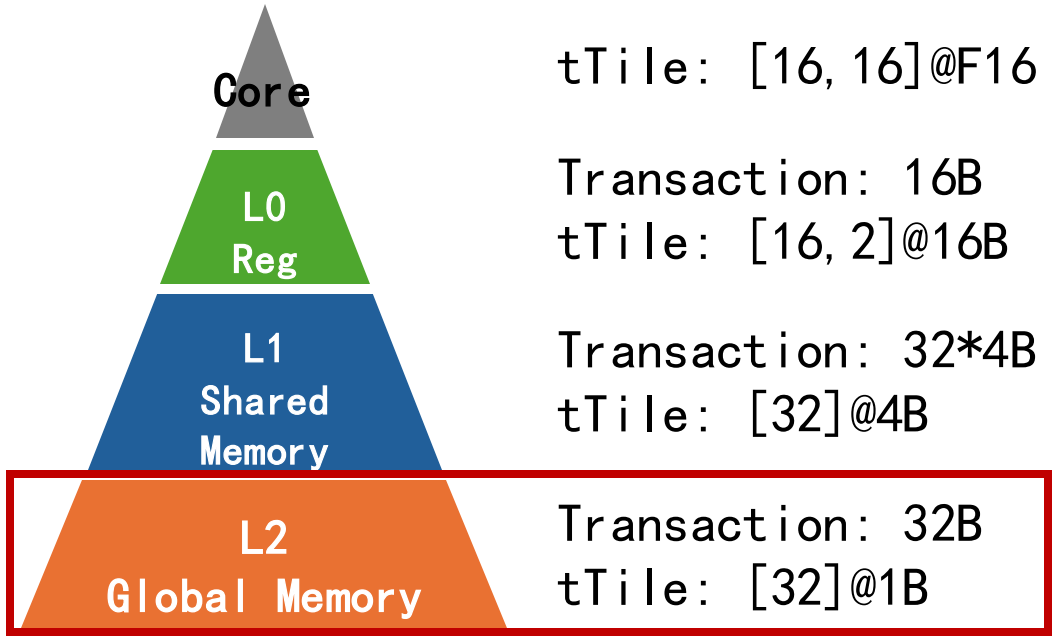
B[16,64]@NF4

Example

目标：A[32,63]@FP16 和 B[32,63]@NF4的MMA

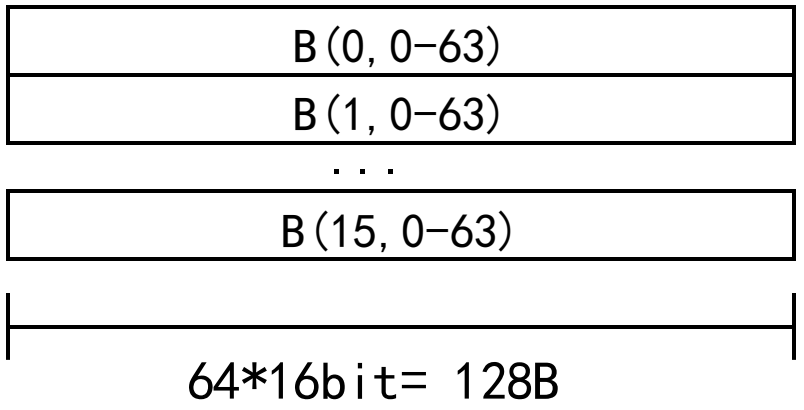
C=compute((M,N), lambda i,j:(sum((A[i,k]@FP16*B[j,k]@NF4)@FP32)@FP32)@FP16),
M=32, N=32, K=63

Device Hint: [4,8]@16B (size: 512B)



tTile-device for NVIDIA A100

Tensor大小：64*16bit*16 = 2048B
Hint size = 512B



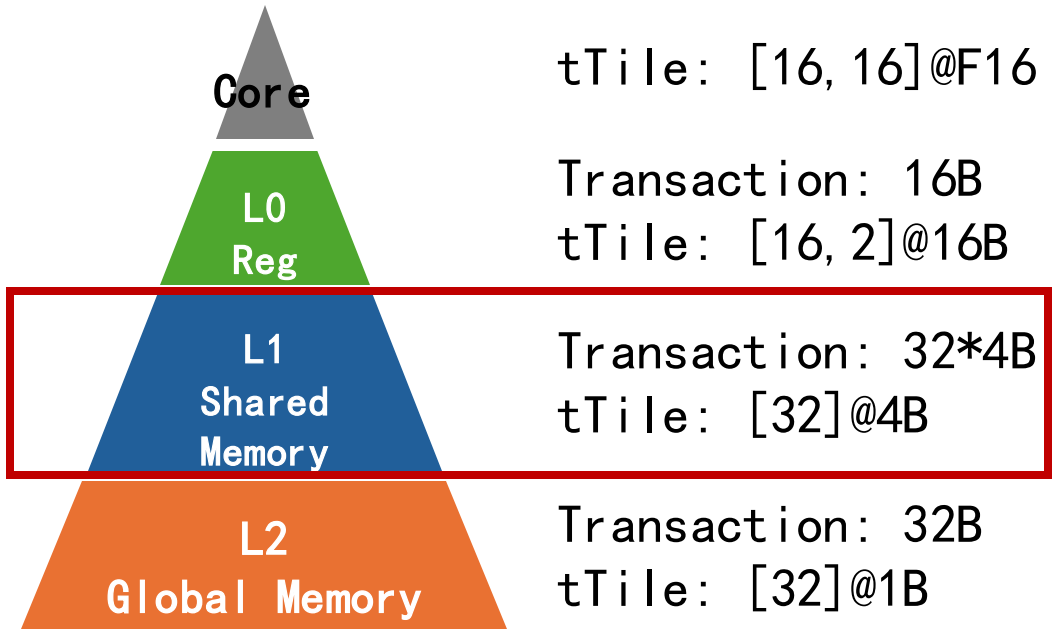
B[16,64]@FP16

Example

目标：A[32,63]@FP16 和 B[32,63]@NF4的MMA

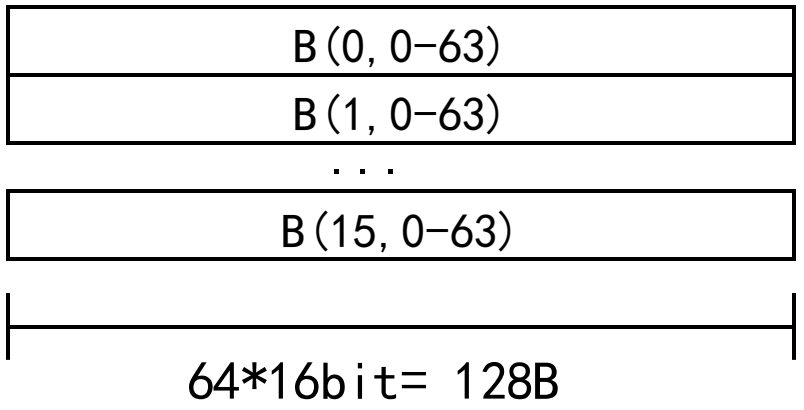
C=compute((M,N), lambda i,j:(sum((A[i,k]@FP16*B[j,k]@NF4)@FP32)@FP32)@FP16),
M=32, N=32, K=63

Device Hint: [4,8]@16B (size: 512B)



tTile-device for NVIDIA A100

Tensor大小：64*16bit*16 = 2048B
Hint size = 512B



B[16,64]@FP16

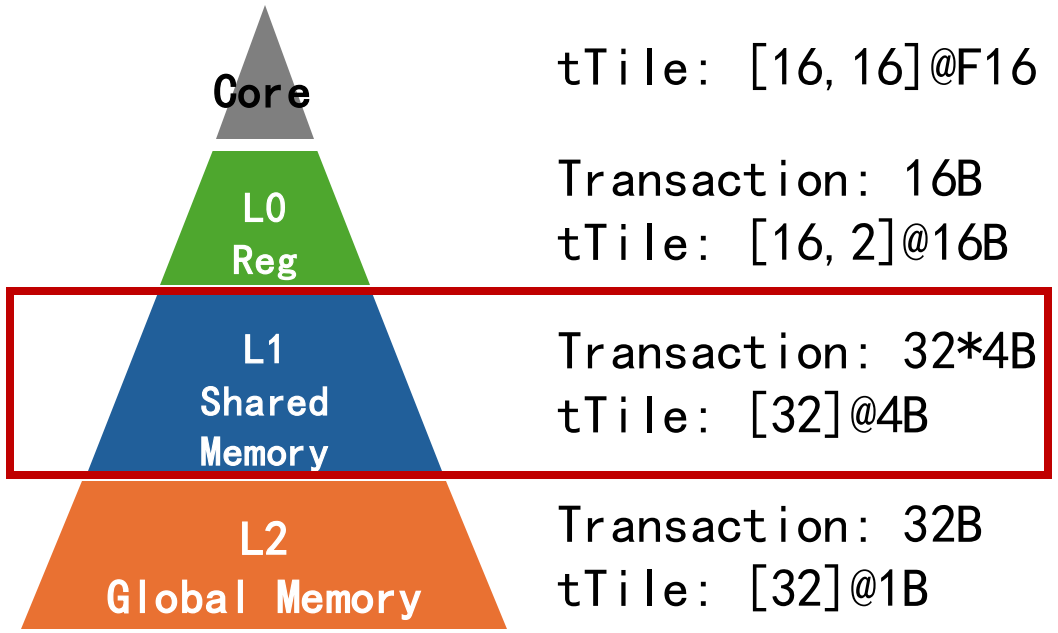
Example

目标：A[32,63]@FP16 和 B[32,63]@NF4的MMA

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

M=32, N=32, K=63

Device Hint: [4,8]@16B (size: 512B)

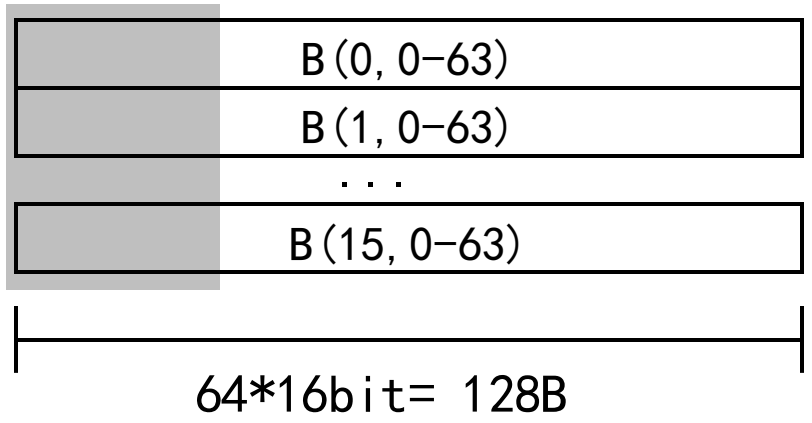


tTile-device for NVIDIA A100

Tensor大小：64*16bit*16 = 2048B

Hint size = 512B

mapB: [16,64]@FP16（使得切分后能对齐）



B[16,64]@FP16

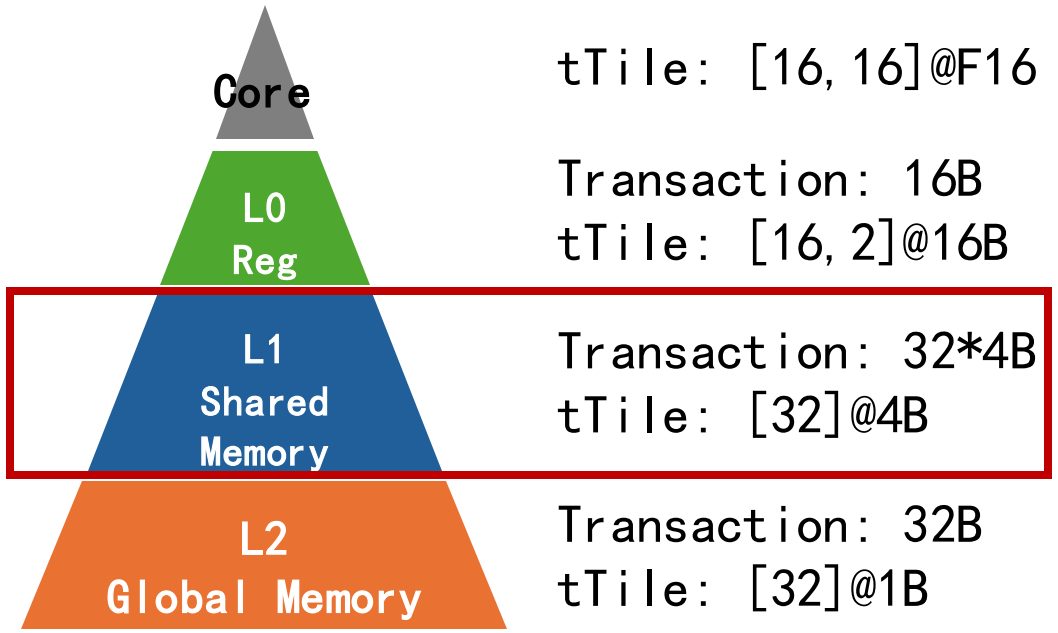
Example

目标：A[32,63]@FP16 和 B[32,63]@NF4的MMA

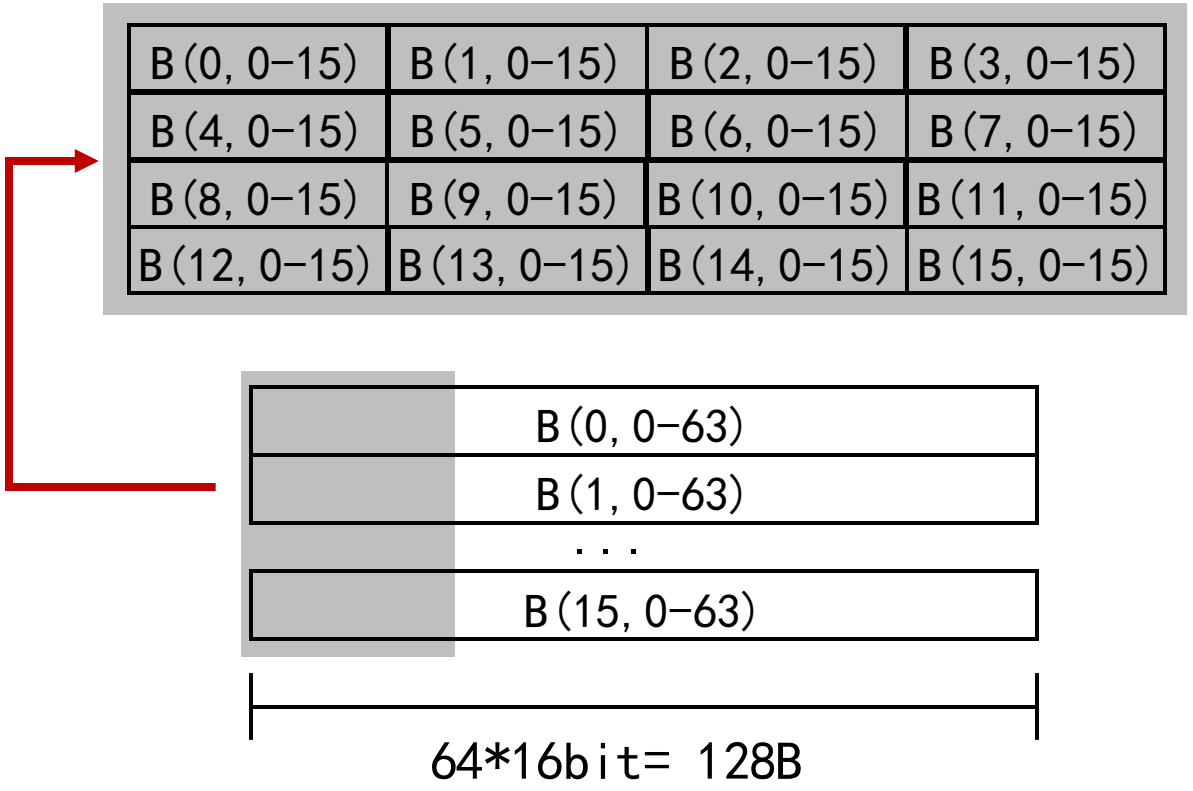
C=compute((M,N), lambda i,j:(sum((A[i,k]@FP16*B[j,k]@NF4)@FP32)@FP32)@FP16),

M=32, N=32, K=63

Device Hint: [4,8]@16B (size: 512B)



tTile-device for NVIDIA A100



B[16,64]@FP16

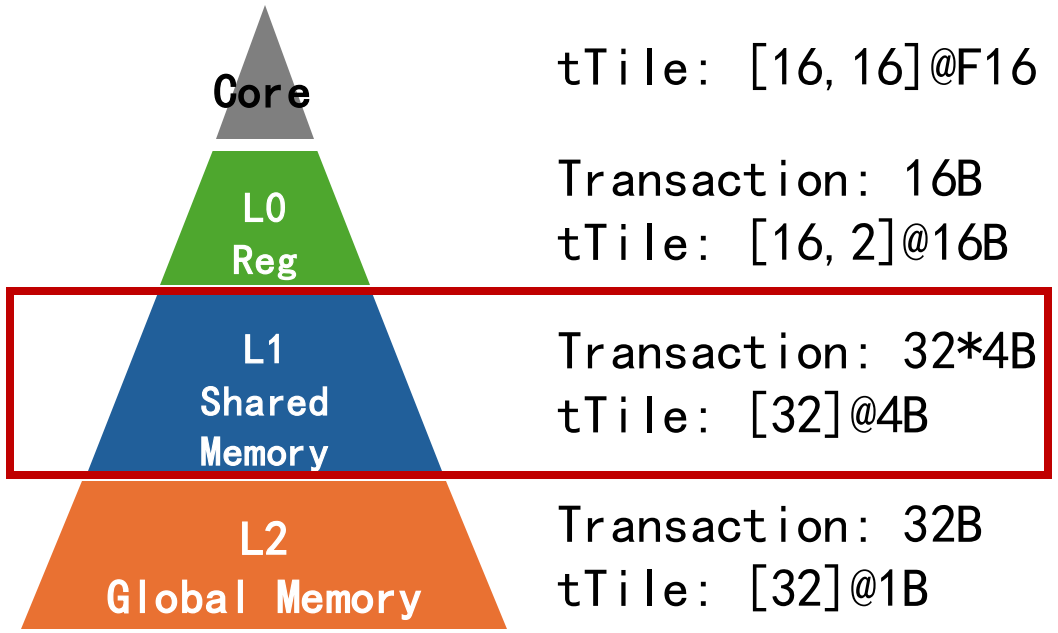
Example

目标：A[32,63]@FP16 和 B[32,63]@NF4的MMA

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

M=32, N=32, K=63

Device Hint: [4,8]@16B (size: 512B)



tTile-device for NVIDIA A100

target tTile: [16,16]@FP16 or [16,2]@16B

sliceB:[4,64]@FP16->[16,16]@FP16

B(0,0-15)	B(1,0-15)	B(2,0-15)	B(3,0-15)
B(4,0-15)	B(5,0-15)	B(6,0-15)	B(7,0-15)
B(8,0-15)	B(9,0-15)	B(10,0-15)	B(11,0-15)
B(12,0-15)	B(13,0-15)	B(14,0-15)	B(15,0-15)

64*16bit= 128B

B[4,64]@FP16

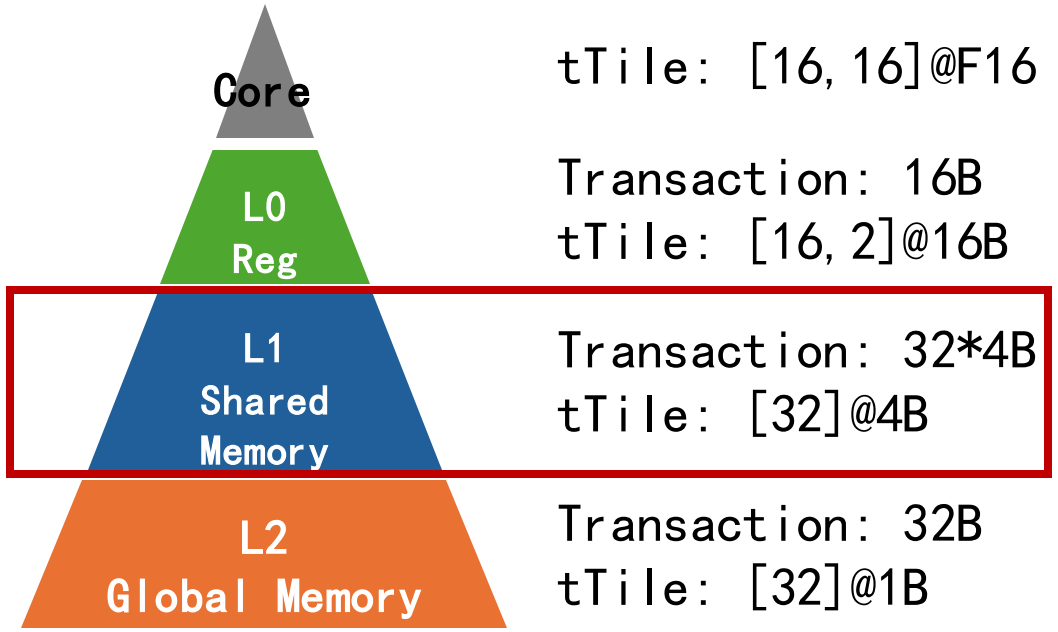
Example

目标：A[32,63]@FP16 和 B[32,63]@NF4的MMA

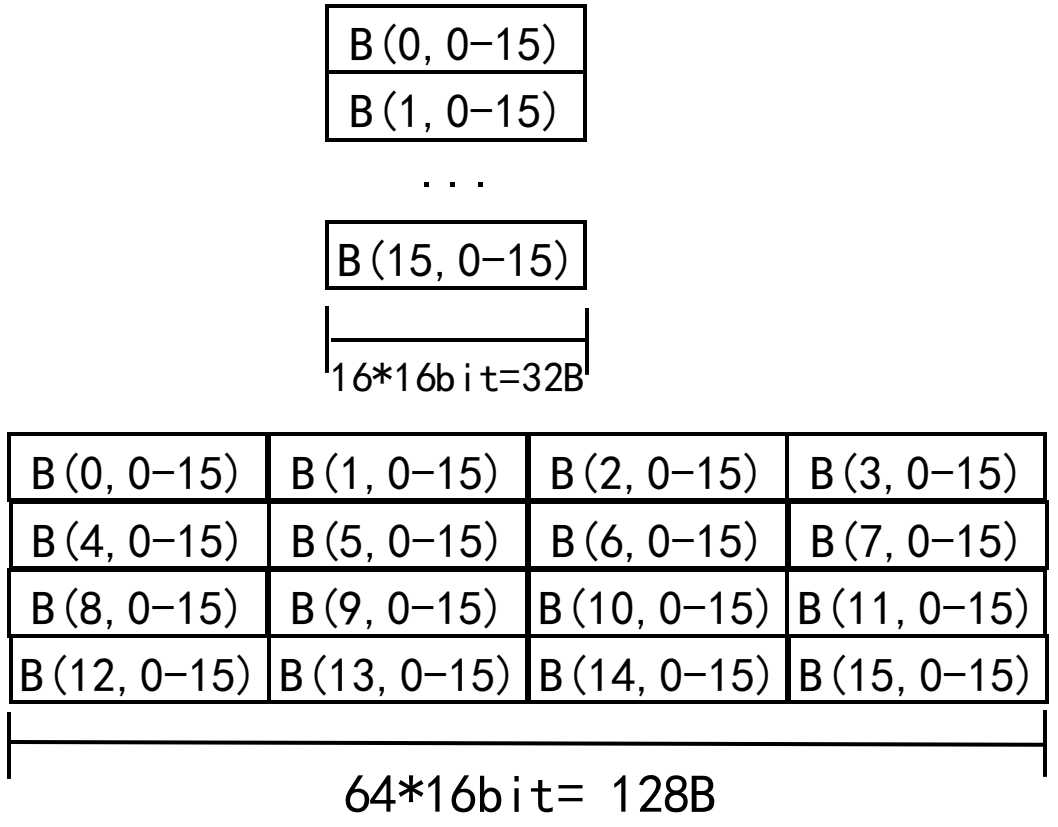
C=compute((M,N), lambda i,j:(sum((A[i,k]@FP16*B[j,k]@NF4)@FP32)@FP32)@FP16),

M=32, N=32, K=63

Device Hint: [4,8]@16B (size: 512B)



tTile-device for NVIDIA A100



B[4,64]@FP16

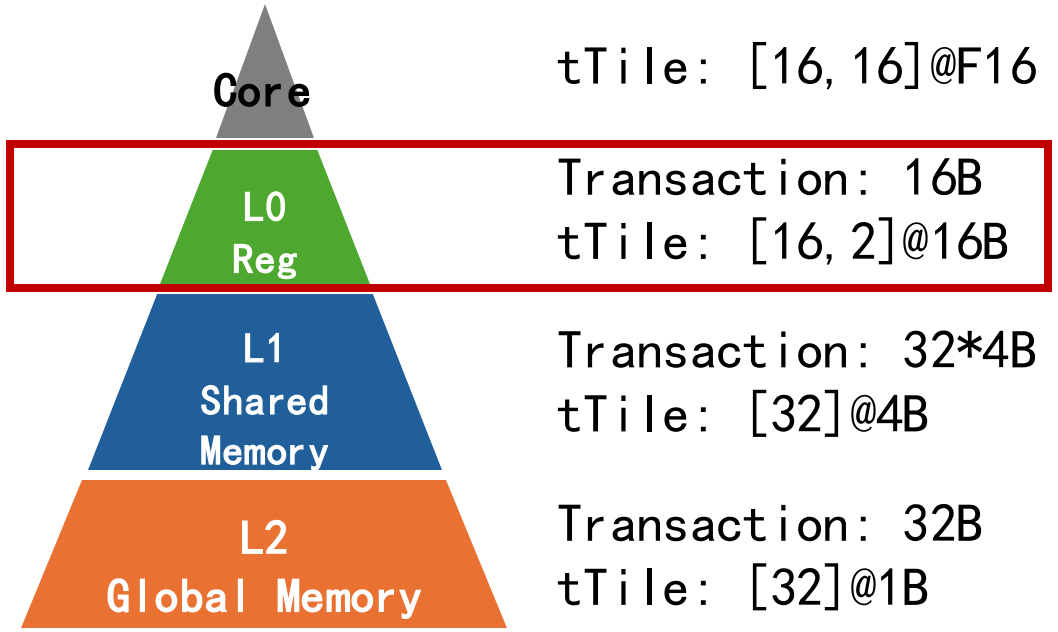
Example

目标：A[32,63]@FP16 和 B[32,63]@NF4的MMA

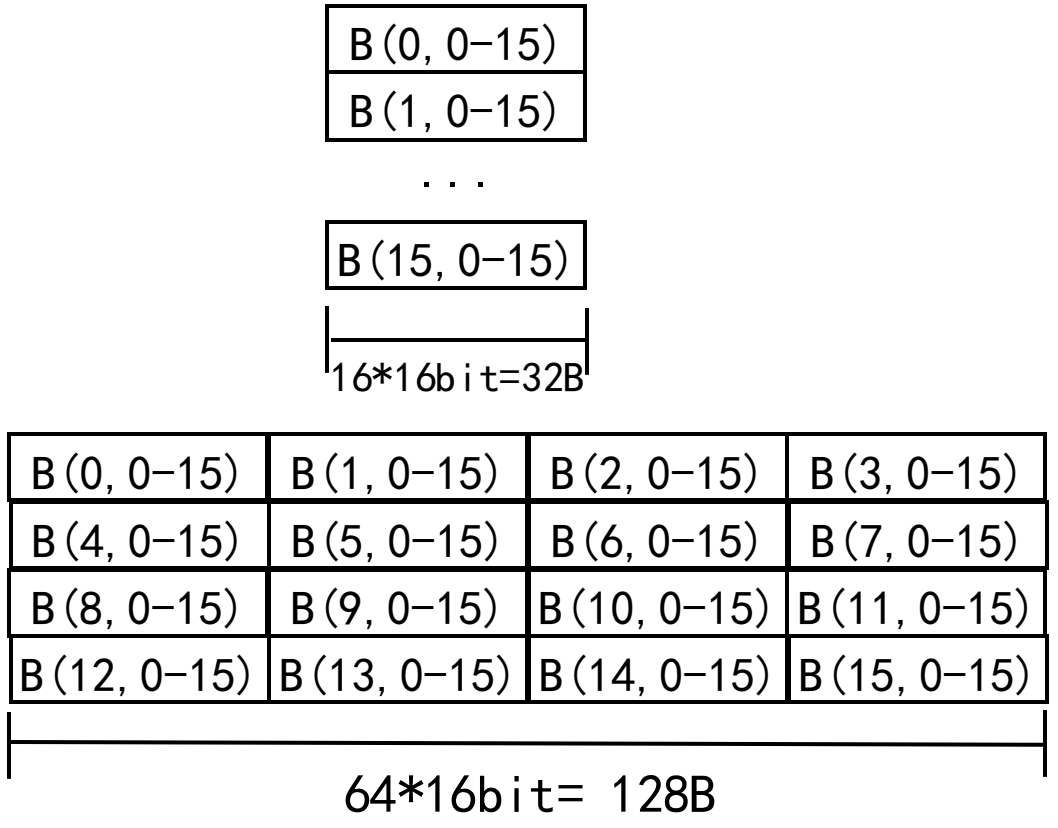
C=compute((M,N), lambda i,j:(sum((A[i,k]@FP16*B[j,k]@NF4)@FP32)@FP32)@FP16),

M=32, N=32, K=63

Device Hint: [4,8]@16B (size: 512B)



tTile-device for NVIDIA A100



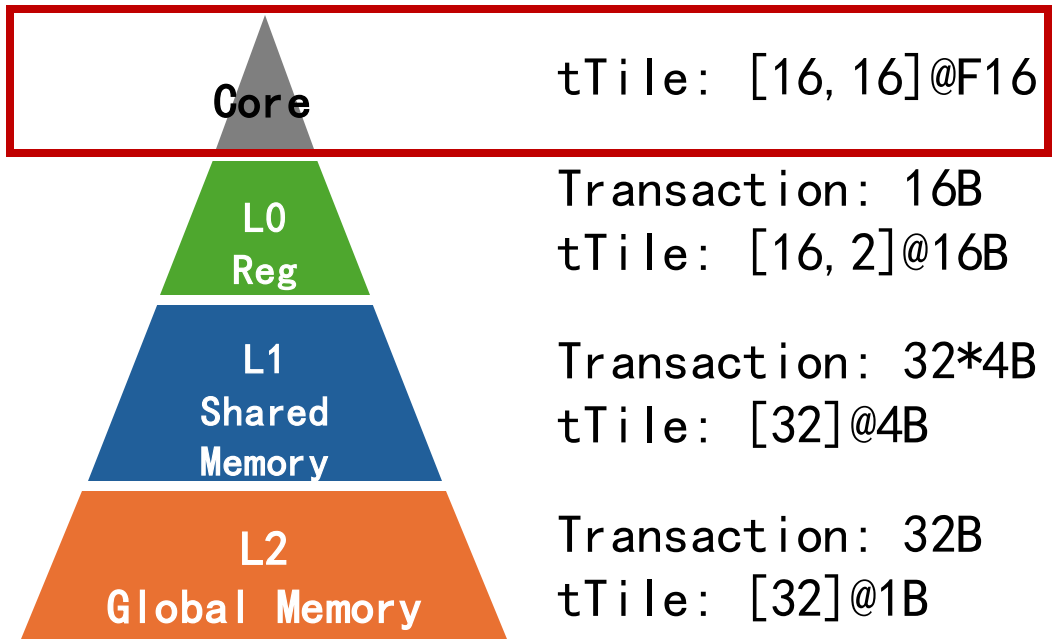
B[4,64]@FP16

Example

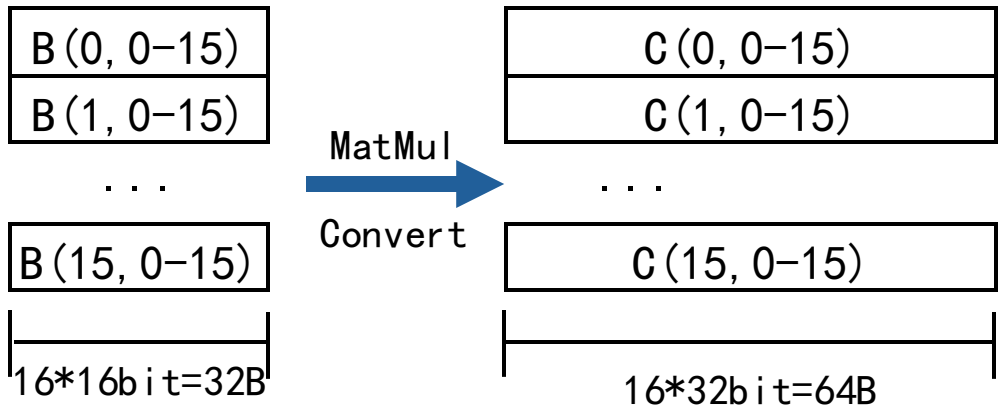
目标：A[32,63]@FP16 和 B[32,63]@NF4的MMA

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

Device Hint: [4,8]@16B (size: 512B)



进入Tensor Core进行运算
将结果C：
convertC: [16,16]@FP32



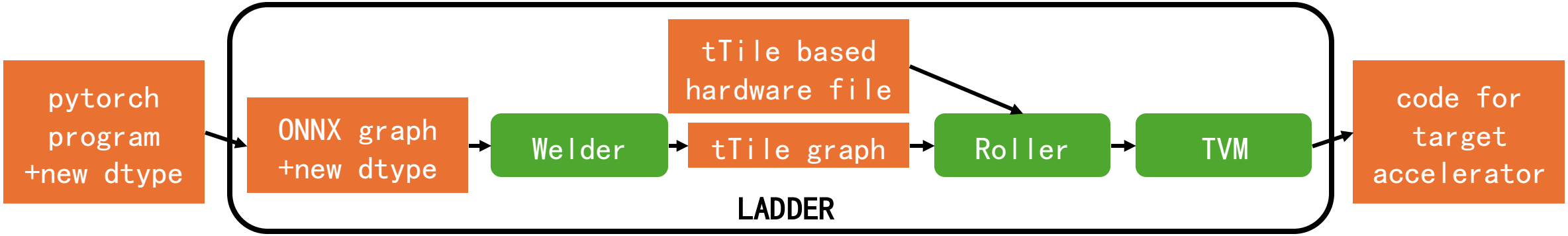
tTile-device for NVIDIA A100

B[16,16]@FP16

C[16,16]@FP32

Implementation

LADDER基于 TVM / Welder / Roller三项工作实现，支持低精度、非标准数据类型。



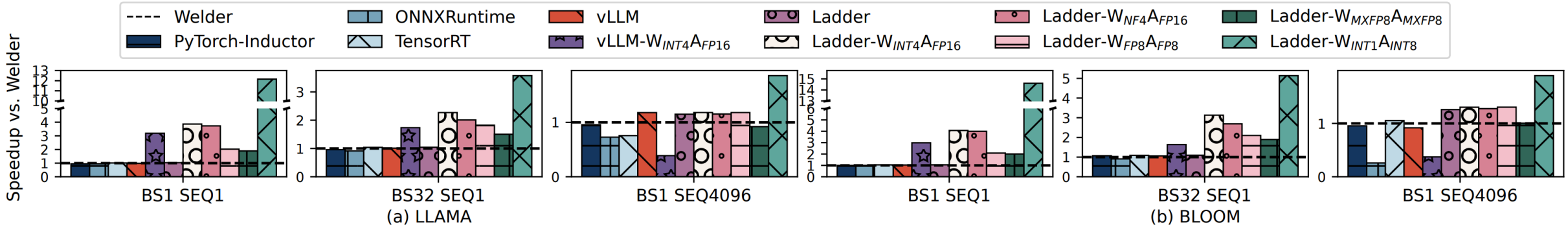
Welder: 端到端图优化，如算子融合，内存布局转换
Roller: tTile配置自动搜索
TVM: 内核调度与代码生成（LOP3优化等）

支持NVIDIA CUDA GPU和AMD ROCm GPU

技术细节：支持非2^n数据如3-bit，PTX指令控制，low bit转换优化等

Evaluation: End-to-end result on A100

LLM Inference

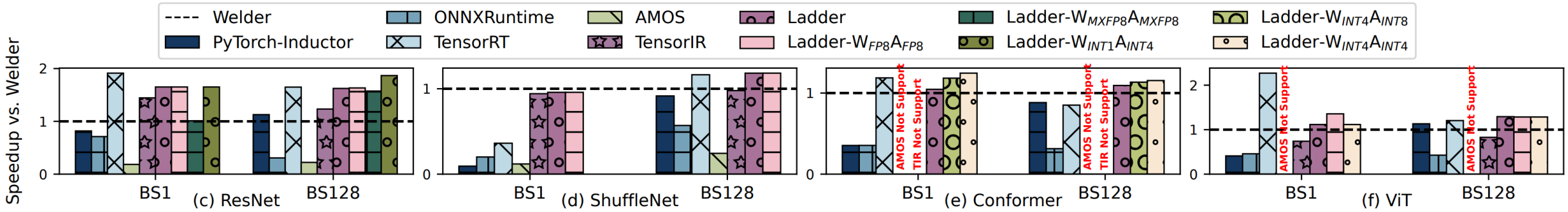


W@FP16 / A@FP16: 1.1x/1.1x avg. speedup over Welder/TensorRT

W@INT4 / A@FP16(GPTQ): 2.3x avg. speedup over vLLM

W@INT1 / A@INT8(BitNet): up to 8.8x speedup

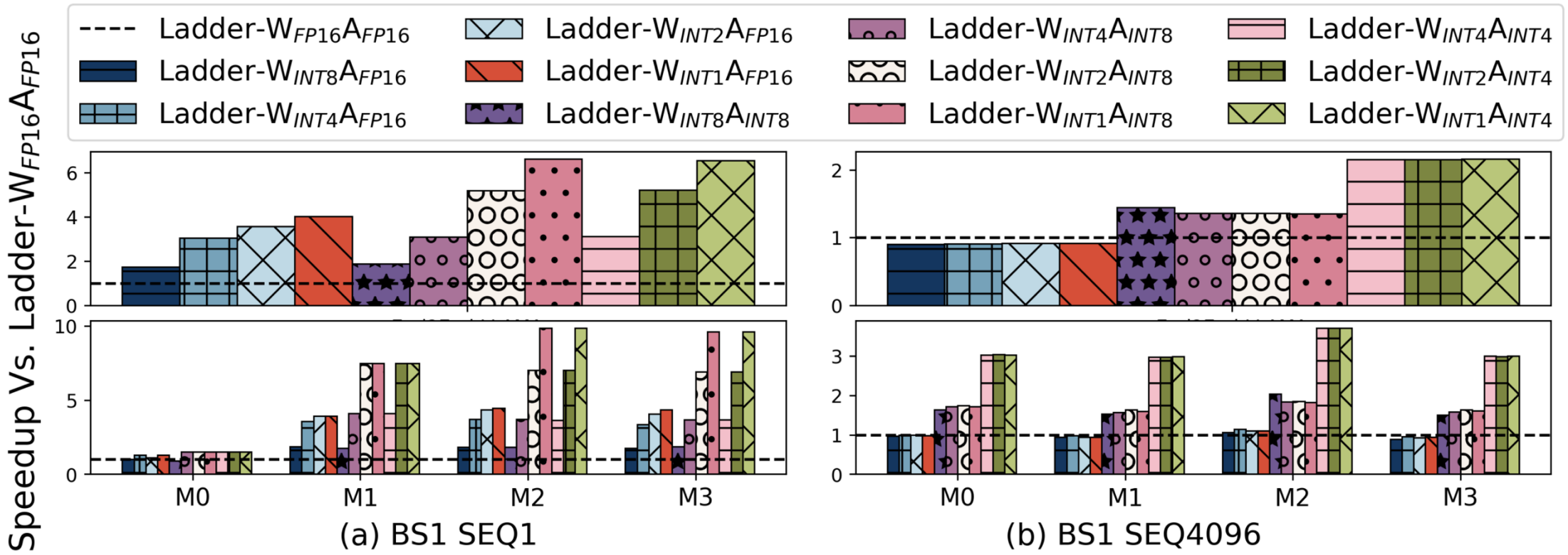
DNN Inference



W@FP16 / A@FP16: 1.4x/1.7x avg. speedup over Welder/TensorRT

Enable low-precision DNN computing with up to 3x speedup over Ladder- W@FP16 / A@FP16

Evaluation – Scaling Bit Width



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)

LADDER

Background：

- **低精度计算**不断演化
- 软硬件适配难以同步发展

LADDER提出了**TypeTile**抽象和**tensor transformation**的方法：

- 表示不断进化的新数据类型
- 为低精度计算提供系统化的机制
- 充分发挥模型和硬件的性能

谢谢！