UNIT: Unifying Tensorized Instruction Compilation

Jian Weng\textsuperscript{1}\textsuperscript{2}, Animesh Jain\textsuperscript{2}, Jie Wang\textsuperscript{1}\textsuperscript{2}, Leyuan Wang\textsuperscript{2}, Yida Wang\textsuperscript{2}, Tony Nowatzki\textsuperscript{1}

\textsuperscript{1}UCLA
\textsuperscript{2}Amazon Web Services

Mar. 1\textsuperscript{st}, 2020
Motivation: Mixed Precision

- Mixed precision
  - Low-precision Inputs
  - High-precision Outputs

Blindly using fp16 does not help the performance

![Graph comparing relative performance of different models using fp32 and fp16 with and without Tensor Core]
Motivation: Tensorization Idiom

- Reducing multiple low precision to high
- Horizontal reduction
- Mixed precision
- S/w Abstraction
  - Kernel Libraries
  - Manually Program Instrinsics
  - DSL Compiler
Unifying Tensorized Instruction Compilation

- Unified Instruction Abstraction
  - Instructions integrated by their semantics
- Unified Analysis of Applicability
  - Computation: Arithmetic isomorphism
  - Memory Access: Pattern isomorphism
- Unified Code Generation Interfaces
  - Reorganize the loops
  - Rewrite with the tensorized inst.
  - Tuning for favorable performance
Tensor Domain Specific Language

Convolution

// Convolution in tensor DSL
a,b = tensor((H,W,C), u8),tensor((R,S,K,C),i8)
k,rc = loop_axis(0,K), reduce_axis(0,C)
x,y = loop_axis(0,H-R+1), loop_axis(0,W-S+1)
r,s = reduce_axis(0,R), reduce_axis(0,S)
c[x,y,k] += i32(a[x+r,y+s,rc])*i32(b[r,s,k,rc])

• Tensor DSL [10, 31, 37]
  • Tensors
  • Loop Variables
    • Data-Parallel/Reduction
  • Expressions
  • Decoupled Loop Organization
Tensor Domain Specific Language

Convolution

// Convolution in tensor DSL
a, b = tensor((H, W, C), u8), tensor((R, S, K, C), i8)
k, rc = loop_axis(0, K), reduce_axis(0, C)
x, y = loop_axis(0, H - R + 1), loop_axis(0, W - S + 1)
r, s = reduce_axis(0, R), reduce_axis(0, S)
c[x, y, k] += i32(a[x + r, y + s, rc]) * i32(b[r, s, k, rc])

Split/Tile

for (i = 0; i < n; ++i)
  // expr uses i
for (io = 0; io < n/4; ++io)
  for (ii = 0; ii < 4; ++ii)
    // expr uses io*4+ii

Reorder

for (i = 0; i < n; ++i)
  for (j = 0; j < m; ++j)
    // expr uses i, j

Unroll

for (i = 0; i < 4; ++i)
  // expr uses i
// expr i replaced by 0
// expr i replaced by 1
// expr i replaced by 2
// expr i replaced by 3

• Tensor DSL [10, 31, 37]
  • Tensors
  • Loop Variables
    • Data-Parallel/Reduction
  • Expressions
  • Decoupled Loop Organization
Unifying Tensorized Instruction Compilation

• Unified Instruction Abstraction
  • Instructions integrated by their semantics
• Unified Analysis of Applicability
  • Computation: Arithmetic isomorphism
  • Memory Access: Pattern isomorphism
• Unified Code Generation Interfaces
  • Reorganize the loops
  • Rewrite with the tensorized inst.
  • Tuning for favorable performance
Unified Instruction Description

- Describe the instruction in *Tensor DSL*
  - Tensors are registers
  - Expr describes arithmetic behavior

- Expose this information for applicability analysis
  - Expression tree
  - Register shape
  - Loop axis
  - Data-Parallel/Reduction

**Intel VNNI**  
x86.avx512.pbpdusd

```
a, b = tensor((64,),u8), tensor((64,),i8)
c, d = tensor((16,), i32), tensor((16,), i32)
i, j = loop_axis(0,16), reduce_axis(0,4)
d[i] = c[i] + sum(i32(a[i*4+j])*i32(b[i*4+j]))
```

**Nvidia Tensor Core**  
nvvm.wmma.m16n16k16.mma.row.row.f32.f32

```
a, b = tensor((16,16),fp16), tensor((16,16),fp16)
i, j = loop_axis(0,16), loop_axis(0,16)
k = reduce_axis(0,16)
c[i,j] += fp32(a[i,k]) * fp32(b[k,j])
```
Unifying Tensorized Instruction Compilation

- Unified Instruction Abstraction
  - Instructions integrated by their semantics

- Unified Analysis of Applicability
  - Computation: Arithmetic isomorphism
  - Memory Access: Pattern isomorphism

- Unified Code Generation Interfaces
  - Reorganize the loops
  - Rewrite with the tensorized inst.
  - Tuning for favorable performance
Analysis: Arithmetic Isomorphism

Convolution

// Convolution in tensor DSL
a, b = tensor((H, W, C), u8), tensor((R, S, K, C), i8)
k, rc = loop_axis(0, K), reduce_axis(0, C)
x, y = loop_axis(0, H-R+1), loop_axis(0, W-S+1)
r, s = reduce_axis(0, R), reduce_axis(0, S)
c[x, y, k] += i32(a[x+r, y+s, rc]) * i32(b[r, s, k, rc])

Intel VNNI  x86.avx512.pbpdusd

a, b = tensor((64,), u8), tensor((64,), i8)
c, d = tensor((16,), i32), tensor((16,), i32)
i, j = loop_axis(0, 16), reduce_axis(0, 4)
d[i] = c[i] + sum(i32(a[i*4+j]) * i32(b[i*4+j]))
Analysis: Memory Isomorphism

**Convolution**

// Convolution in tensor DSL

```cpp
a, b = tensor((H,W,C), u8), tensor((R,S,K,C), i8)
k, rc = loop_axis(0, K), reduce_axis(0, C)
x, y = loop_axis(0, H-R+1), loop_axis(0, W-S+1)
r, s = reduce_axis(0, R), reduce_axis(0, S)
c[x, y, k] += i32(a[x+r, y+s, rc])*i32(b[r, s, k, rc])
```

- **k** -> **i**, **rc** -> **j**

**Intel VNNI**

```cpp
a, b = tensor((64,), u8), tensor((64,), i8)
c, d = tensor((16,), i32), tensor((16,), i32)
i, j = loop_axis(0, 16), reduce_axis(0, 4)
d[i] = c[i] + sum(i32(a[i*4+j])*i32(b[i*4+j]))
```
Analysis: Memory Isomorphism

Convolution

```c
// Convolution in tensor DSL
a, b = tensor((H,W,C), u8), tensor((R,S,K,C), i8)
k, rc = loop_axis(0,K), reduce_axis(0,C)
x, y = loop_axis(0,H-R+1), loop_axis(0,W-S+1)
r, s = reduce_axis(0,R), reduce_axis(0,S)
c[x,y,k] += i32(a[x+r,y+s,rc])*i32(b[r,s,k,rc])
```

- $k \rightarrow i$, $rc \rightarrow j$

```c
// Intel VNNI
x86.avx512.pbpdusd
a, b = tensor((64,), u8), tensor((64,), i8)
c, d = tensor((16,), i32), tensor((16,), i32)
i, j = loop_axis(0,16), reduce_axis(0,4)
d[i] = c[i] + sum(i32(a[i*4+j])*i32(b[i*4+j]))
```

```c
for (x=0; x<(H-R)+1; ++x)
  for (y=0; y<(W-S)+1; ++y)
    for (k=0; k<K; ++k)
      for (r=0; r<R; ++r)
        for (s=0; s<S; ++s)
          for (rc=0; rc<C; ++rc)
            c[x,y,k] += a[x+r,y+s,rc]*b[r,s,k,rc];
```
Analysis: Memory Isomorphism

Convolution

// Convolution in tensor DSL
a, b = tensor((H,W,C), u8), tensor((R,S,K,C), i8)
for (x=0; x<(H-R)+1; ++x)
    for (y=0; y<(W-S)+1; ++y)
        for (k=0; k<K; ++k)
            for (r=0; r<R; ++r)
                for (s=0; s<S; ++s)
                    for (rc=0; rc<C; ++rc)
                        c[x,y,k] += a[x+r,y+s,rc]*b[r,s,k,rc];

Intel VNNI  
x86.avx512.pbpdusd
for (x=0; x<(H-R)+1; ++x)
    for (y=0; y<(W-S)+1; ++y)
        for (k=0; k<K; ++k)
            for (r=0; r<R; ++r)
                for (s=0; s<S; ++s)
                    for (co=0; co<C; co+=4)
                        for (ki=0; ki<16; ++ki)
                            for (ci=0; ci<4; ++ci) {
                                k=ko+ki, rc=co+ci;
                                c[x,y,k] += a[x+r,y+s,rc]*b[r,s,k,rc];
                            }
Analysis: Memory Isomorphism

Convolution

```c
// Convolution in tensor DSL
a, b = tensor((H,W,C), u8), tensor((R,S,K,C), i8)
k, rc = loop_axis(0,K), reduce_axis(0,C)
x, y = loop_axis(0,H-R+1), loop_axis(0,W-S+1)
r, s = reduce_axis(0,R), reduce_axis(0,S)
c[x, y, k] += i32(a[x+r, y+s, rc]) * i32(b[r, s, k, rc])
```

```
// Convolution in tensor DSL
a, b = tensor((64,), u8), tensor((64,), i8)
c, d = tensor((16,), i32), tensor((16,), i32)
i, j = loop_axis(0,16), reduce_axis(0,4)
d[i] = c[i] + sum(i32(a[i*4+j])*i32(b[i*4+j]))
```

- **k** -> **i**, **rc** -> **j**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c[x, y, k]</td>
<td>d[i]</td>
<td>[16] = [16]</td>
<td></td>
</tr>
<tr>
<td>c[x, y, k]</td>
<td>c[i]</td>
<td>[16] = [16]</td>
<td></td>
</tr>
<tr>
<td>a[x+r, y+s, rc]</td>
<td>a[i*4+j]</td>
<td>[4] ⊆ [64] (Broadcast)</td>
<td></td>
</tr>
<tr>
<td>b[r, s, k, rc]</td>
<td>b[i*4+j]</td>
<td>[4x16] = [64] (Concatenate)</td>
<td></td>
</tr>
</tbody>
</table>

Intel VNNI x86.avx512.pbpdusd
Unifying Tensorized Instruction Compilation

- Unified Instruction Abstraction
  - Instructions integrated by their semantics
- Unified Analysis of Applicability
  - Computation: Arithmetic isomorphism
  - Memory Access: Pattern isomorphism
- Unified Code Generation Interfaces
  - Reorganize the loops
  - Rewrite with the tensorized inst.
  - Tuning for favorable performance

Diagram:

1. Tensor Operation Prog.
2. Analysis
4. Xform
5. Hardware Target
   - Intel x86
   - NVIDIA
   - ...
Transformation: Loop Reorg.

- \( k \rightarrow i \), \( rc \rightarrow j \)
- Transform loops to for rewriting
  - Tile loops by corresponding trip counts
  - Reorder to the inner most

```c
for (x=0; x<(H-R)+1; ++x)
    for (y=0; y<(W-S)+1; ++y)
        for (k=0; k<K; ++k)
            for (r=0; r<R; ++r)
                for (s=0; s<S; ++s)
                    for (rc=0; rc<C; ++rc)
                        c[x,y,k] += a[x+r,y+s,rc]*b[r,s,k,rc];
```
• Implement callback functions to generate each operand

def operand_generator(array, base, loops, coef):
    # implement the rule of codegen here
    # array: the array pointer of the memory operation
    # loops: chosen loops to be tensorized,
    #        from inner to outer
    # coef: the coefficient of each loop variable
    # base: the base addresses
    # thus, index = base + sum(loops[i] * coef[i])
    # return operand load intrinsic
Unified Code Generation

• Implement callback functions to generate each operand

```python
def operand_generator(array, base, loops, coef):
    # implement the rule of codegen here
    # array: the array pointer of the memory operation
    # loops: chosen loops to be tensorized,
    #        from inner to outer
    # coef: the coefficient of each loop variable
    # base: the base address
    # thus, index = base + sum(loops[i] * coef[i])
    # return operand load intrinsic
```

Generated:

Memory Operation:
```plaintext```
a[x+r, y+s, rc]
```
Flattened:
```plaintext```
a[(x+r)*d1+(y+s)*d0+rc]
```
Arguments:
```plaintext```
array: a
base: (x+r)*d1+(y+s)*d0
loop: [rc, k]
coef: [1, 0]```
Unified Code Generation

- Implement callback functions to generate each operand

```
def operand_generator(array, base, loops, coef):
    # implement the rule of codegen here
    # array: the array pointer of the memory operation
    # loops: chosen loops to be tensorized, from inner to outer
    # coef: the coefficient of each loop variable
    # base: the base address
    # thus, index = base + sum(loops[i] * coef[i])
    # return operand load intrinsic

    Generated: a[(x+r)*d1+(y+s)*d0+(0..4)]
```

Memory Operation:
```python
a[x+r,y+s,rc]
```

Flattened:
```python
a[(x+r)*d1+(y+s)*d0+rc]
```

Arguments:
- array: a
- base: (x+r)*d1+(y+s)*d0
- loop: [rc, k]
- coef: [1, 0]
Unified Code Generation

• Implement callback functions to generate each operand:

```python
def operand_generator(array, base, loops, coef):
    # implement the rule of codegen here
    # array: the array pointer of the memory operation
    # loops: chosen loops to be tensorized, from inner to outer
    # coef: the coefficient of each loop variable
    # base: the base addresss
    # thus, index = base + sum(loops[i] * coef[i])
    # return operand load intrinsic

    Generated: broadcast(a[(x+r)\*d1+(y+s)\*d0+(0..4)], 16)
```

Memory Operation:

\[ a[x+r, y+s, rc] \]

Flattened:

\[ a[(x+r)\*d1+(y+s)\*d0+rc] \]

Arguments:

- array: \( a \)
- base : \( (x+r)\*d1+(y+s)\*d0 \)
- loop : \([rc, k]\)
- coef : \([1, 0]\)
Unified Code Generation

• Implement callback functions to generate each operand

```python
def operand_generator(array, base, loops, coef):
    # implement the rule of codegen here
    # array: the array pointer of the memory operation
    # loops: chosen loops to be tensorized,
    #        from inner to outer
    # coef: the coefficient of each loop variable
    # base: the base address
    # thus, index = base + sum(loops[i] * coef[i])
    # return operand load intrinsic
```

• Invoke each callback function to plug in the operands

```python
def codegen(opcode, operands, callbacks):
    args = [func(arg) for arg, func in zip(operands, callbacks)]
    return inline_asm(opcode, args)
```
Unifying Tensorized Instruction Compilation

• Unified Instruction Abstraction
  • Instructions integrated by their semantics
• Unified Analysis of Applicability
  • Computation: Arithmetic isomorphism
  • Memory Access: Pattern isomorphism
• Unified Code Generation Interfaces
  • Reorganize the loops
  • Rewrite with the tensorized inst.
• Tuning for favorable performance
Idiom-Based Performance Tuning

- The outer loops are open to performance tuning

- **Data Parallel/Reduction**

- Parallelism
  - Coarse-Grain: Thread-level Parallelism
    - Distribute compute to proper #cores
  - Fine-Grain: Pipeline Parallelism
    - Achieve instruction-level parallelism by avoiding loop-carried penalty

```plaintext
for (s0=0; s0<d0; ++s0)
  for (s1=0; s1<d1; ++s1)
    for (s2=0; s2<d2; ++s2)
      ...
      for (r0=0; r0<rd0; ++r0)
        for (r1=0; r1<rd1; ++r1)
          tensorized inst.;
```
CPU Performance Tuning

- Coarse-Grain Parallelism: Distributing spatial loops to threads
- Find-Grain Parallelism: Avoiding loop carried dependences
- Reorder and unroll a spatial loop

```c
for (s0=0; s0<d0; ++s0) ...
   for (sn=0; sn<dn; ++sn) ...
      for (r0=0; r0<rd0; ++r0)
         for (r1=0; r1<rd1; ++r1)
            tensorized instruction;

parallel (fused=0; fused<fd; ++fused)
   for (serial=0; serial<sd; ++serial)
      for (r0=0; r0<extr0; ++r0)
         for (rm=0; rm<ext_rm; ++rm) {
            tensorized-instruction.0;
            tensorized-instruction.1;
            ...
         }
```
GPU Performance Tuning (Generic)

- Coarse Grain: Launch the CUDA kernel on multiple GPU blocks.

- No data reuse across the innermost reduction loop

- Loop-carried accumulation causes pipeline penalty

```cpp
// Direct accumulation
// a[n,k], b[k,m], c[n,m]
Buffer<fp16,16,16> A, B;
Buffer<fp32,16,16> C;
for (i=0; i<n; i+=16)
    for (j=0; j<m; j+=16)
        for (r=0; r<k; r+=16) {
            A = Load(a[i:16,r:16]);
            B = Load(b[r:16,j:16]);
            C += TensorCore(A, B);
        }
Store(c[i:16,j:16], C);
```
GPU Performance Tuning (Generic)

- Unroll 2 loops by \( pxp \)
- Loop-carried dependence avoided by the outer-product
- Each loaded sub-matrix are reused \( p \) times

```c
// pxp outer product
// a[n,k], b[k,m], c[n,m]
for (i=0; i<n; i+=16*p)
  for (j=0; j<m; j+=16*p)
    for (r=0; r<k; r+=16) {
      Buffer<fp16,16,16> A[p], B[p];
      Buffer<fp32,16,16> C[p][p];
      #pragma unroll
      for (x=0; x<p; ++x) {
        A[x] = Load(a[i+x*16:16,r:16]);
        B[x] = Load(b[r:16,j+x*16:16]);
      }
      #pragma unroll
      for (x=0; x<p; ++x)
        #pragma unroll
        for (y=0; y<p; ++y)
          C[x][y] += TensorCore(a[x],b[y]);
      for (x=0; x<p; ++x)
        for (y=0; y<p; ++y)
          Store(c[i+x*16,j+y*16], C[x][y]);
    }
```
CNN-Specialized Tuning on GPU

- Small width and height
- Deep channels
CNN-Specialized Tuning (Fuse Dim.)

- Tensors in DNN workloads often have small width and height

  ① Padding a perfect tiling size is wasting

  ② Fuse width and height to safe memory traffic

- Introduces software overhead of data rearrangement

①: More than 3/4 (25/32) traffic is wasted by padding

②: Less than 1/4 (15/64) traffic is wasted by padding.
CNN-Specialized Tuning (Split Red)

- Tensors in DNN workloads often have deep input channels
  ① Split the reduce loop across threads
  ② Store the partial accumulation in shared memory
  ③ Reduce the partial sum and write back

- A proper degree of splitting
  - Small: Too small to hide memory latency
  - Large: Overhead of sync; register pressure
Evaluation: Methodology

• **Hardware**
  - CPU: Amazon EC2 c5.12xlarge, with Intel Xeon Platinum 8275 CL @3.00G
  - GPU: Amazon EC2 p3.2xlarge, with Nvidia Tesla V100
  - ARM: Amazon EC2 m6g.8xlarge, with Amazon Graviton 2 ARM CPU

• **Software**
  - Compiler and Runtime: LLVM-10, and CUDA-10
  - Vendor Provided Libraries: cuDNN 7.6.5, and oneDNN v1.6.1
  - DNN Models: BS=1, MxNet models converted to TVM Relay [32] for
    - Padded data shape
    - Proper data layout NCHW[x]c and KCRS[y]k[x]c [23]
Evaluation: Goal

• Performance
  • End-to-end: How is the overall performance of UNIT?
    • 9 popular end-to-end DNN models
  • Ablation: How does each optimization help the performance?
    • 16 representative convolution layers

• Extensibility
  • Hardware Platform: ARM DOT
E2E Performance (Intel Xeon Platinum 8275CL)

- ResNet-18
- ResNet-50
- ResNet-50b
- Inception-bn
- Inception-v3
- ResNet-101
- ResNet-152
- Mobilenet-v1
- Mobilenet-v2
- GM

- MxNet (MKLDNN)
- UNIT
## Performance Impact of Tuning (CPU)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>288</td>
<td>160</td>
<td>1056</td>
<td>80</td>
<td>128</td>
<td>192</td>
<td>256</td>
<td>1024</td>
<td>128</td>
<td>576</td>
<td>96</td>
<td>1024</td>
<td>576</td>
<td>64</td>
<td>64</td>
<td>608</td>
</tr>
<tr>
<td>H=W(I)</td>
<td>35</td>
<td>9</td>
<td>7</td>
<td>73</td>
<td>16</td>
<td>16</td>
<td>14</td>
<td>16</td>
<td>14</td>
<td>16</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>384</td>
<td>224</td>
<td>192</td>
<td>192</td>
<td>128</td>
<td>192</td>
<td>256</td>
<td>512</td>
<td>160</td>
<td>192</td>
<td>128</td>
<td>256</td>
<td>128</td>
<td>96</td>
<td>128</td>
<td>192</td>
</tr>
<tr>
<td>R=S</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Stride</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>H=W(O)</td>
<td><strong>17</strong></td>
<td>7</td>
<td>7</td>
<td>7</td>
<td><strong>71</strong></td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>27</td>
<td>28</td>
<td>14</td>
</tr>
</tbody>
</table>

### Diagram

- **oneDNN**
- **Parallel**
- **+Unroll**
- **+Tune**

---

**Note:** The table and diagram represent the performance impact of tuning (CPU) with different configurations and optimization techniques.
## Performance Impact of Tuning (GPU)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>288</td>
<td>160</td>
<td>1056</td>
<td>80</td>
<td>128</td>
<td>192</td>
<td>256</td>
<td>1024</td>
<td>128</td>
<td>576</td>
<td>96</td>
<td>1024</td>
<td>576</td>
<td>64</td>
<td>64</td>
<td>608</td>
</tr>
<tr>
<td>H=W(I)</td>
<td>35</td>
<td>9</td>
<td>7</td>
<td>73</td>
<td>16</td>
<td>16</td>
<td>14</td>
<td>16</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>29</td>
<td>56</td>
<td>14</td>
</tr>
<tr>
<td>K</td>
<td>384</td>
<td>224</td>
<td>192</td>
<td>192</td>
<td>128</td>
<td>192</td>
<td>256</td>
<td>512</td>
<td>160</td>
<td>192</td>
<td>128</td>
<td>256</td>
<td>128</td>
<td>96</td>
<td>128</td>
<td>192</td>
</tr>
<tr>
<td>R=S</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Stride</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>H=W(O)</td>
<td>17</td>
<td>7</td>
<td>7</td>
<td>71</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>27</td>
<td>28</td>
<td>14</td>
</tr>
</tbody>
</table>

- **cuDNN**: White bars
- **Generic**: Light gray bars
- **+FuseDim**: Medium gray bars
- **+SplitK**: Dark gray bars
- **+Tune**: Black bars

![Graph showing performance impact of tuning](image)
E2E Performance (ARM Amazon Graviton 2)

• Describe ARM DOT in Tensor DSL

• Reuse Analysis, Xform, and Tuning

```python
a, b = tensor((16,), i8), tensor((16,), i8)
c, d = tensor((4,), i32), tensor((4,), i32)
i, j = loop_axis(0, 4), reduce_axis(0, 4)
d[i] = c[i] + sum(i32(a[i*4+j])*i32(b[i*4+j]))
```

![Performance Chart]

<table>
<thead>
<tr>
<th>Model</th>
<th>NEON</th>
<th>TVM</th>
<th>UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-18</td>
<td>7.5</td>
<td>10.0</td>
<td></td>
</tr>
<tr>
<td>ResNet-50</td>
<td>5.7</td>
<td>6.1</td>
<td></td>
</tr>
<tr>
<td>ResNet-50b</td>
<td>5.7</td>
<td>6.1</td>
<td></td>
</tr>
<tr>
<td>Inception-bn</td>
<td>5.8</td>
<td>6.8</td>
<td></td>
</tr>
<tr>
<td>Inception-v3</td>
<td>12.6</td>
<td>15.4</td>
<td></td>
</tr>
<tr>
<td>ResNet-101</td>
<td>5.7</td>
<td>6.2</td>
<td></td>
</tr>
<tr>
<td>ResNet-152</td>
<td>5.2</td>
<td>5.6</td>
<td></td>
</tr>
<tr>
<td>Mobilenet-v1</td>
<td>2.5</td>
<td>2.8</td>
<td></td>
</tr>
<tr>
<td>Mobilenet-v2</td>
<td>3.1</td>
<td>3.2</td>
<td></td>
</tr>
<tr>
<td>GM</td>
<td>5.4</td>
<td>5.8</td>
<td></td>
</tr>
</tbody>
</table>
Conclusion

• UNIT
  • A unified compilation flow for the emerging tensorization idiom
  • Tuning strategies for DNN workloads

• Future Work
  • Automated data layout transformation
  • A extension to vectorizer