Lecture 16: Domain Specific Language and IR

CSE599G1: Spring 2017

Where are we

High level Packages

User API

- Programming API
- Gradient Calculation (Differentiation API)

System Components

- Computational Graph Optimization and Execution
- Runtime Parallel Scheduling

Architecture

- GPU Kernels, Optimizing Device Code
- Accelerators and Hardwares

Where are we

Programming API

Gradient Calculation (Differentiation API)

Computational Graph Optimization and Execution

Runtime Parallel Scheduling / Networks

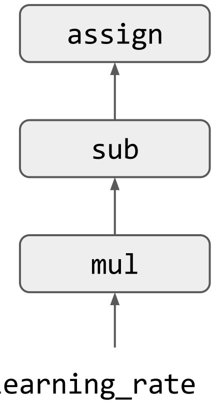
Gap between computation graph and hardware

GPU Kernels, Optimizing Device Code

Accelerators and Hardwares

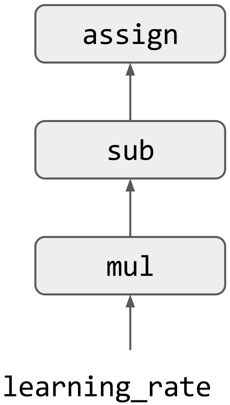
Question

```
w = w - lr * grad
```



Operator Fusion

Computation



Sequential Kernel Execution

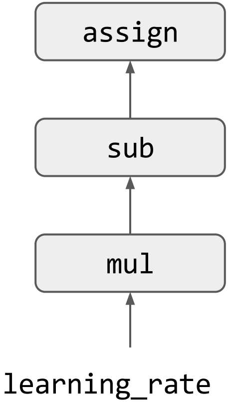
```
for (int i = 0; i < n; ++i) {
    temp1[i] = lr * grad[i]
}
for (int i = 0; i < n; ++i) {
    temp2[i] = w[i] - temp1[i]
}
for (int i = 0; i < n; ++i) {
    w[i] = temp2[i]
}
```

Fused Kernel Execution

```
for (int i = 0; i < n; ++i) {
    w[i] = w[i] - lr * grad[i]
}
```

More Backends

Computation



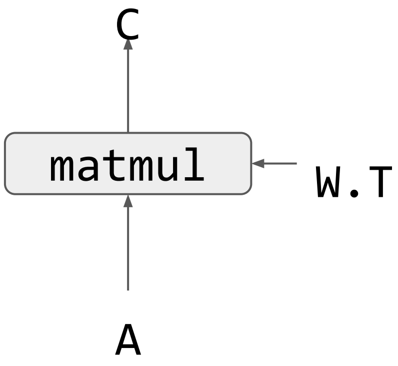
OpenCL (Arm devices)

```
__kernel void update(__global float *w,
                     __global float* grad,
                     int n) {
   int gid = get_global_id(0)
   if (gid < n) {
     w[gid] = w[gid] - lr * grad[gid];
   }
}
```

Metal (iOS devices)

```
kernel void update(float *w [[buffer(0)],
                   float* grad [[buffer(1)]],
                   uint gid [[thread_position_in_grid]]
                   int n) {
   if (gid < n) {
     w[gid] = w[gid] - lr * grad[gid];
   }
}
```

Computation and Data Layout

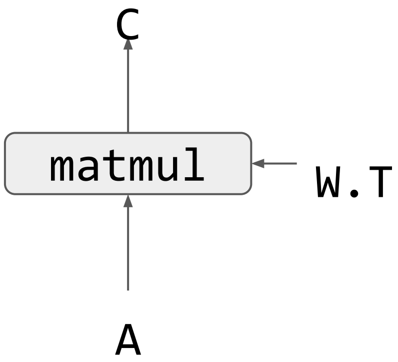


Vanilla Matrix Multiplication

```
float A[n][h], W[n][h], C[n][m];

for (int i = 0; i < n; ++i)
  for (int j = 0; j < m; ++j) {
    C[i][j] = 0;
    for (int k = 0; k < h; ++k) {
      C[i][j] += A[i][k] * W[j][k];
    }
  }
```

Challenge: Computation and Data Layout

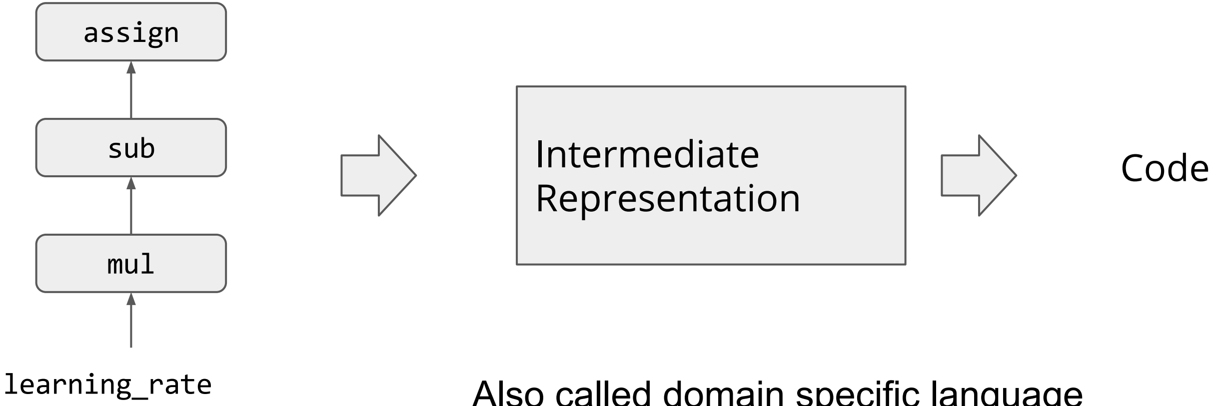


Data Packing `A[i][j] -> A[i/4][j/4][i%4][j%4]`

Code

```
float A[n/4][h/4][4][4];
float W[n/4][h/4][4][4];
float C[n/4][m/4][4][4];
for (int i = 0; i < n/4; ++i)
  for (int j = 0; j < m/4; ++j) {
    C[i][j] = 0
    for (int k = 0; k < h/4; ++k) {
      C[i][j] += dot(A[i][k], W[j][k]);
    }
  }
```

Bridge Layer for Code Generation



Also called domain specific language

Expression Template: Linear Algebra AST in C++

Expression Template:

- Expression returns AST
- Lazy evaluate the expression at assignment
- Generate one kernel per evaluation during compilation

```
float data_a[n] = {1, 2, 3};
float data_b[n] = {2, 3, 4};
float data_c[n] = {3, 4, 5};
float lr = 0.1;
Vec A(sa, n), B(sb, n), C(sc, n);


// run expression
A = B + C * lr;
```

Device Invariant Code via Templatization

```
template<typename xpu>
void UpdateSGD(Tensor<xpu, 2> weight,
               const Tensor<xpu, 2> &grad,
               float eta, float lambda) {
  weight -= eta * (grad + lambda * weight);
}
```

Expression Template in DL Frameworks

- Eigen: http://eigen.tuxfamily.org/
  - Used in TensorFlow
- mshadow: https://github.com/dmlc/mshadow
  - Used in MXNet

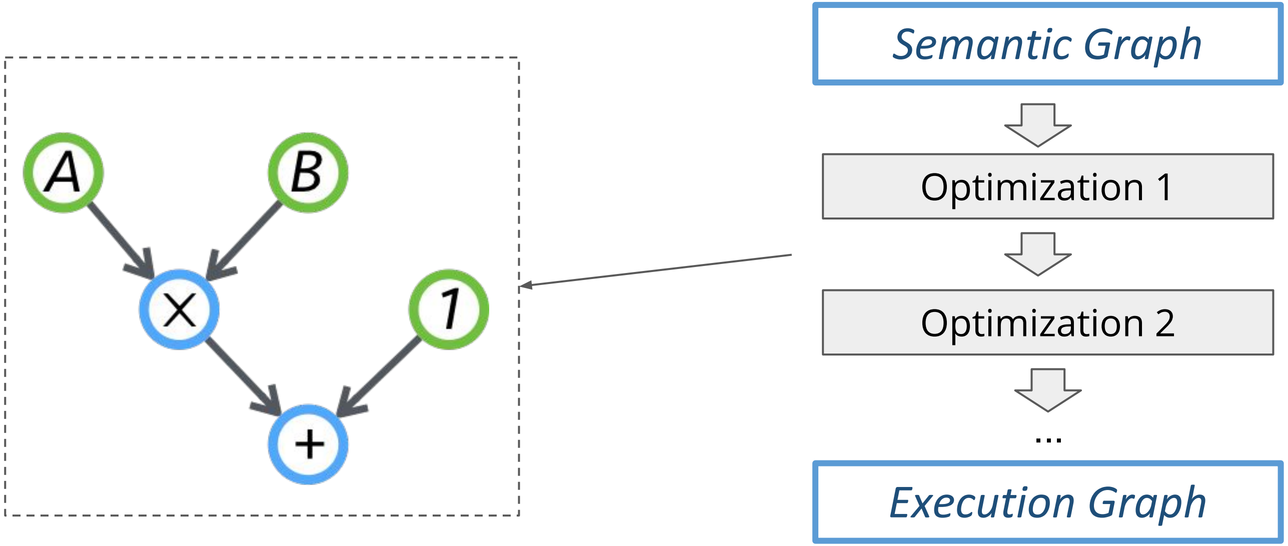
- Tutorial on how it works
  - https://github.com/dmlc/mshadow/tree/master/guide/exp-template

- Discussion: what are the drawbacks of expression template

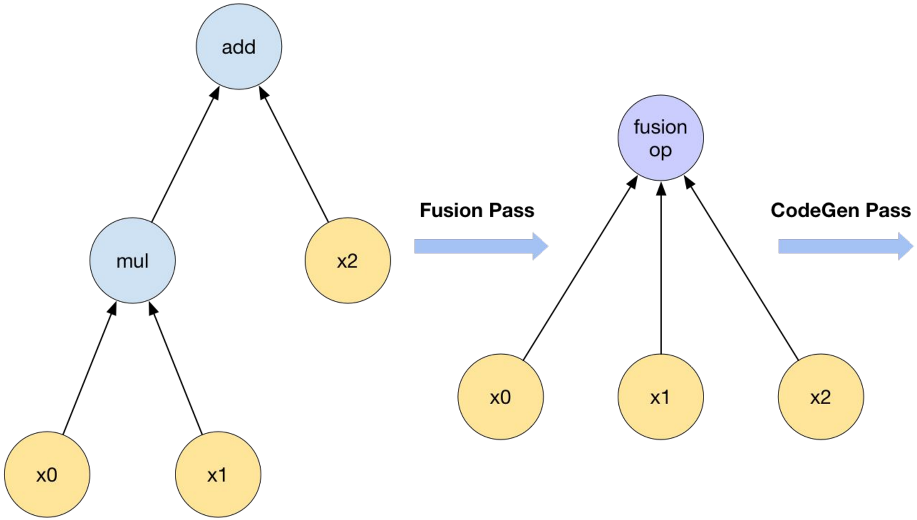
Computational Graph level IR

Computation Graph as IR

- Benefit from high level view
- Need code generation/interpretation rule for each op

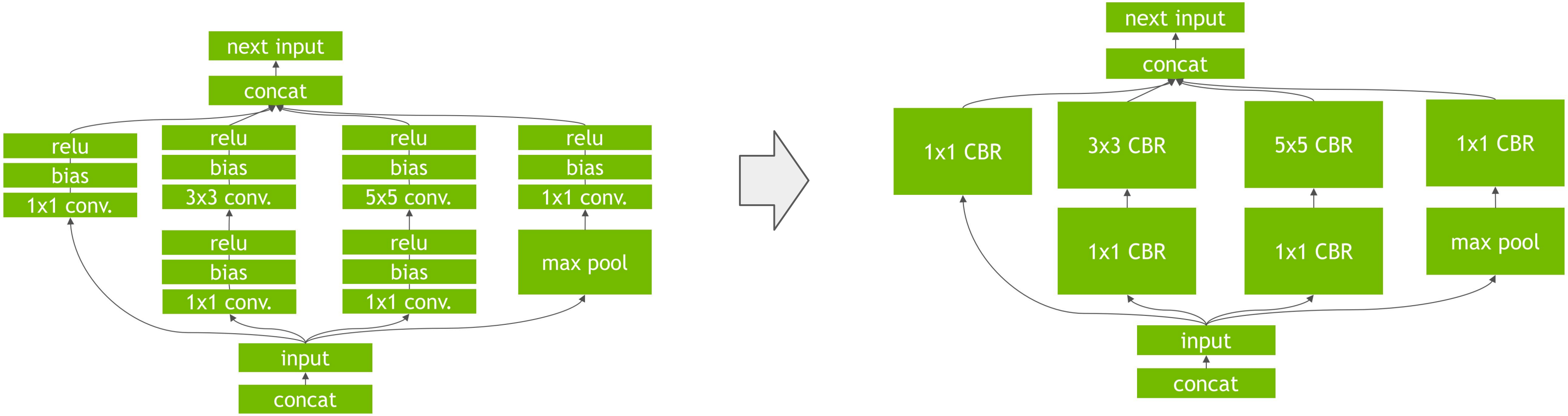


Codegen Rule for Elementwise Op




```
extern "C" __global__ fusion_kernel (uint32_t num_element,
      float *x0, float *x1, float *x2, float *y) {
  int global_idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (global_idx < num_element)
    y[global_idx] = (x0[global_idx] * x1[global_idx]) + x2[global_idx];
}
```

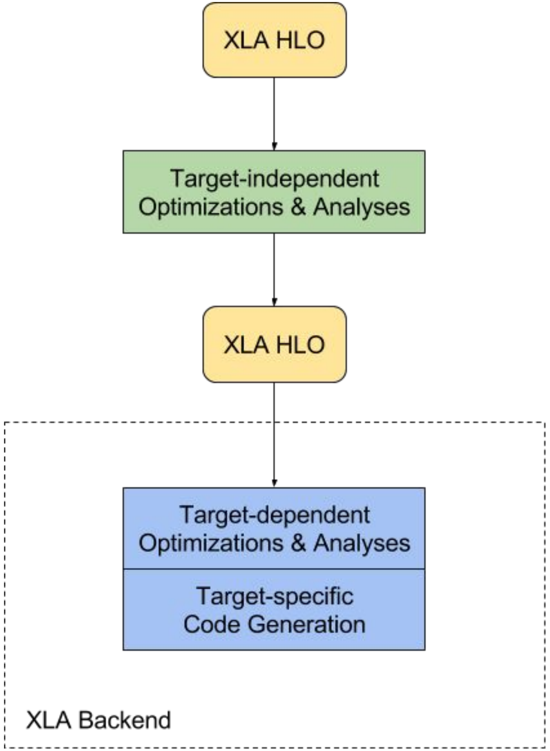
Nvdia TensorRT: Rule based Fusion



Source: Nvidia

XLA: Tensorflow Compiler Stack

- Constant shape dimension
- Data layout is specific
- Operations are low level tensor primitives
    - Map
    - Broadcast
    - Reduce
    - Convolution
    - ReduceWindow
    - …



Array Index based DSL

Index based Computation Description

Description of `C = A + B`

```
n = t.var('n')
m = t.var('m')
A = t.placeholder((m, n), name='A')
B = t.placeholder((m, n), name='B')
C = t.compute((m, n), lambda i, j: A[i, j] + B[i, j])
```

Computation Rule for index `i,j`

Computation Description for Matrix Multiplication

Description of `C = dot(A, B.T)`

```
A = t.placeholder((l, n), name='A')
B = t.placeholder((l, m), name='B')
k = t.reduce_axis((0, l), name='k')
C = t.compute((m, n),
        lambda i, j: t.sum(A[k, j] * B[k, i], axis=k));
```

Loop Transformation Rule

```
C = t.compute((m, n), lambda i, j: A[i, j] + B[i, j])
s = t.create_schedule(C.op)
```

```
for (int i = 0; i < n; ++i) {
  C[i] = A[i] + B[i];
}
```

Loop Transformation Rule

```
C = t.compute((m, n), lambda i, j: A[i, j] + B[i, j])
s = t.create_schedule(C.op)
bx, tx = s[C].split(C.op.axis[0], factor=64)
```

```
for (int bx = 0; bx < ceil(n / 64); ++bx) {
  for (int tx = 0; tx < 64; ++tx) {
    int i = bx * 64 + tx;
    if (i < n) {
      C[i] = A[i] + B[i];
    }
  }
}
```

Loop Transformation Rule

```
C = t.compute((m, n), lambda i, j: A[i, j] + B[i, j])
s = t.create_schedule(C.op)
bx, tx = s[C].split(C.op.axis[0], factor=64)
s[C].bind(bx, tvm.thread_axis("blockIdx.x"))
s[C].bind(tx, tvm.thread_axis("threadIdx.x"))
```

```
int i = blockIdx.x * 64 + threadIdx.x;
if (i < n) {
   C[i] = A[i] + B[i];
}
```

Key Characteristics of Array Index based DSLs

- Index based description
- Loop transformation rules to generate different programs

Summary: Challenges for IR

- Simple description language for computation
- Rich transformation for computation patterns
- Keep up with emerging hardware
- It is still an open question!