Lecture 6: Optimize for Hardware Backends

CSE599G1: Spring 2017
Where are we

**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
Goal: High Level Program to Bare Metal Code

\[ w = w - lr \times \text{grad} \]
Discussion

What are the tricks you can do to make your program run faster on CUDA/x86/any backend?
Time Complexity of Matrix Multiplication

Compute $C = \text{dot}(A, B.T)$

```cpp
float A[n][n], B[n][n], C[n][n];

for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) {
        C[i][j] = 0;
        for (int k = 0; k < n; ++k) {
            C[i][j] += A[i][k] * B[j][k];
        }
    }
```
Time Complexity of Matrix Multiplication

Compute $C = \text{dot}(A, B.T)$

```c
float A[n][n], B[n][n], C[n][n];

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

O(n^3)
```

Any information it did not capture?
Modern Memory Hierarchy

From: Latency numbers every programmer should know

- CPU Thread
- Registers
- L1 Cache
- L2 Cache
- DRAM

0.5 ns
7ns  14x L1 cache
200 ns  20x L2 cache, 200x L1 cache

There can also be L3 cache.
Example GPU Memory Hierarchy

Registers: R 0 cycle / R-after-W 20 cycles

L1 cache: 30 cycles
Shared memory: 28 cycles
Constant L1 cache: 28 cycles
Texture L1 cache: 92 cycles

L2 cache: 200 cycles

DRAM: 350 cycles
Architecture Aware Cost Analysis

```c
float A[n][n], B[n][n], C[n][n];
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        register float c = 0;
        for (int k = 0; k < n; ++k) {
            register float a = A[i][k];
            register float b = B[j][k];
            c += a * b;
        }
        C[i][j] = c;
    }
}
```

A’s dram->register time cost:
B’s dram->register time cost:
A’s register memory cost :
B’s register memory cost :
C’s register memory cost :
Architecture Aware Cost Analysis

```c
float A[n][n], B[n][n], C[n][n];
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        register float c = 0;
        for (int k = 0; k < n; ++k) {
            register float a = A[i][k];
            register float b = B[j][k];
            c += a * b;
        }
        C[i][j] = c;
    }
}
```

A’s dram->register time cost:  $n^3$
B’s dram->register time cost:  $n^3$
A’s register memory cost :  1
B’s register memory cost :  1
C’s register memory cost :  1

**load cost:**  $2 \times \text{dramspeed} \times n^3$
Register cost:  3
Tiled Matrix Multiplication

dram float A[n/v1][n/v3][v1][v3];
dram float B[n/v2][n/v3][v2][v3];
dram float C[n/v1][n/v2][v1][v2];

for (int i = 0; i < n/v1; ++i) {
  for (int j = 0; j < n/v2; ++j) {
    register float c[v1][v2] = 0;
    for (int k = 0; k < n / v3; ++k) {
      register float a[v1][v3] = A[i][k];
      register float b[v2][v3] = B[j][k];
      c += dot(a, b);
    }
    C[i][j] = c;
  }
}
Output Tiled Matrix Multiplication

dram float A[n/v1][n/v3][v1][v3];
dram float B[n/v2][n/v3][v2][v3];
dram float C[n/v1][n/v2][v1][v2];

for (int i = 0; i < n/v1; ++i) {
    for (int j = 0; j < n/v2; ++j) {
        register float c[v1][v2] = 0;
        for (int k = 0; k < n / v3; ++k) {
            register float a[v1][v3] = A[i][k];
            register float b[v2][v3] = B[j][k];
            c += dot(a, b);
        }
        C[i][j] = c;
    }
}

A’s dram->register time cost:
B’s dram->register time cost:
A’s register memory cost :
B’s register memory cost:
C’s register memory cost :
Output Tiled Matrix Multiplication

dram float A[n/v1][n/v3][v1][v3];
dram float B[n/v2][n/v3][v2][v3];
dram float C[n/v1][n/v2][v1][v2];

for (int i = 0; i < n/v1; ++i) {
    for (int j = 0; j < n/v2; ++j) {
        register float c[v1][v2] = 0;
        for (int k = 0; k < n/v3; ++k) {
            register float a[v1][v3] = A[i][k];
            register float b[v2][v3] = B[j][k];
            c += dot(a, b);
        }
        C[i][j] = c;
    }
}

A’s dram->register time cost:   n^3/v2
B’s dram->register time cost:   n^3/v1
A’s register memory cost :   v1*v3
B’s register memory cost of:   v2*v3
C’s register memory cost of :   v1*v2

load cost:  dramspeed * (n^3/v2 + n^3 / v1)
Register cost: v1*v3 + v2 * v3 + v1 * v2

Question: How to choose v1, v2, v3
Cache Line Aware Tiling

dram float A[n/b1][b1][n];
dram float B[n/b2][b2][n];
dram float C[n/b1][n/b2][b1][b2];
for (int i = 0; i < n/b1; ++i) {
    l1cache float a[b1][n] = A[i];
    for (int j = 0; j < n/b2; ++j) {
        l1cache b[b2][n] = B[j];
        C[i][j] = dot(a, b);
    }
}

sub matrix kernel, can apply output tiling
Cache Line Aware Tiling

dram float A[n/b1][b1][n];
dram float B[n/b2][b2][n];
dram float C[n/b1][n/b2][b1][b2];
for (int i = 0; i < n/b1; ++i) {
    l1cache float a[b1][n] = A[i];
    for (int j = 0; j < n/b2; ++j) {
        l1cache b[b2][n] = B[j];
        C[i][j] = dot(a, b);
    }
}

A’s dram->l1 time cost: $n^2$
B’s dram->l1 time cost: $n^3 / b1$

Constraints:
- $b1 \times n + b2 \times n < l1$ cache size
- To still apply output blocking on dot
  - $b1 \% v1 == 0$
  - $b2 \% v2 == 0$
Combine it Together

```c

dram float A[n/b1][b1/v1][n][v1];
dram float B[n/b2][b2/v2][n][v2];

for (int i = 0; i < n/b1; ++i) {
    l1cache float a[b1/v1][n][v1] = A[i];
    for (int j = 0; j < n/b2; ++j) {
        l1cache b[b2/v2][n][v2] = B[j];
        for (int x = 0; x < b/v1; ++x)
            for (int y = 0; x < b/v1; ++y) {
                register float c[v1][v2] = 0;
                for (int k = 0; k < n; ++k) {
                    register float ar[v1] = a[x][k];
                    register float br[v1] = b[y][k];
                    C += dot(ar, br)
                }
            }
    }
}
```

load cost:

\[
\text{l1speed} \times (n^3/v2 + n^3 / v1) + \text{dramspeed} \times (n^2 + n^3 / b1)
\]
The Key Ingredients: Memory Reuse

```c

dram float A[n/v1][n/v3][v1][v3];
dram float B[n/v2][n/v3][v2][v3];
dram float C[n/v1][n/v2][v1][v2];

for (int i = 0; i < n/v1; ++i) {
    for (int j = 0; j < n/v2; ++j) {
        register float c[v1][v2] = 0;
        for (int k = 0; k < n / v3; ++k) {
            register float a[v1][v3] = A[i][k];
            register float b[v2][v3] = B[j][k];
            c += dot(a, b);
        }
        C[i][j] = c;
    }
}
```

a get reused v2 times
b get reused v1 times

A’s dram->register time cost: \( \frac{n^3}{v2} \)
B’s dram->register time cost: \( \frac{n^3}{v1} \)
Generalize to GPU: Reuse among threads

Global memory

Shared memory

Cooperatively loaded by both thread 1 and 2

Each thread block computes a 64 x 64 area
Reuse and Invariant

float A[n][n];
float B[n][n];
float C[n][n];

C[i][j] = sum(A[i][k] * B[j][k], axis=k)

Access of A is independent of j,
tile the j dimension by v allows reuse A for v times.
Discussion: What about Convolution?

float Input[n][ci][h][w];
float Weight[co][ci][K][K];
float Output[n][co][h][w];

Output[b][co][y][x] =
    sum(Input[b][k][y+ry][x+rx] * 
        Weight[co][k][ry][rx], axis=[k, ry, rx])
Variants of GPU Backends

Computation

- assign
- sub
- mul

learning_rate

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)

```
kern 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];
    }
}
```
Operator Fusion

Computation

Sequential Kernel Execution

Fused Kernel Execution

```c
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]
}
```
Data Layout Matters: Locality in Access

**Data Packing**

\[ A[i][j] \rightarrow A[i/4][j/4][i\%4][j\%4] \]

**Code**

```c
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]);
        }
    }
```
Optimizations = Too Many Variant of Operators

- Different tiling patterns
- Different fuse patterns
- Different data layout
- Different hardware backends
Explore Code Generation Approach

Also called domain specific language

Intermediate Representation

assign
sub
mul
learning_rate

Code
TVM Stack: Next Lecture