# 10-414/714 – Deep Learning Systems: Algorithms and Implementation

# Hardware Acceleration

Fall 2021

J. Zico Kolter and Tianqi Chen (this time)
Carnegie Mellon University

# Outline

General acceleration techniques

Case study: matrix multiplication

# Outline

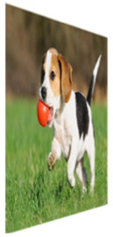
General acceleration techniques

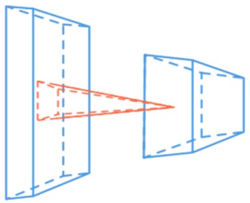
Case study: matrix multiplication

# Layers in machine learning frameworks

**ML Models**

$x$

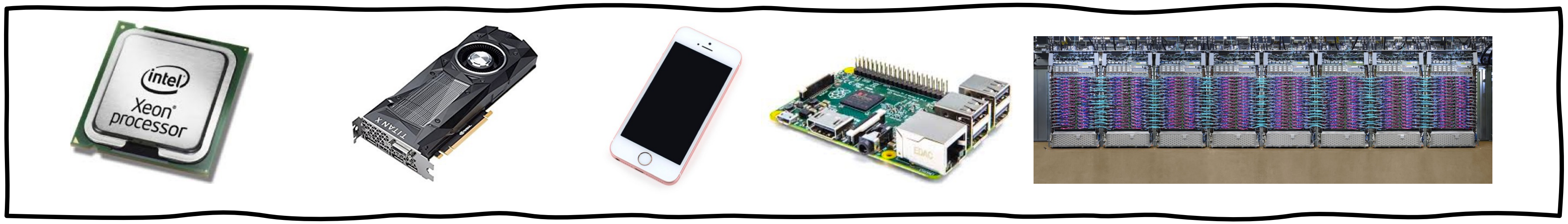




$h_\theta(x)$

Computational graph

Tensor linear algebra libraries



# Vectorization

Adding two arrays of length 128

```
void vecadd(float* A, float *B, float* C) {
   for (int i = 0; i < 64; ++i) {
      float4 a = load_float4(A + i*4);
      float4 b = load_float4(B + i*4);
      float4 c = add_float4(a, b);
      store_float4(C + i* 4, c);
  }
}
```

Additional requirements: memory (A, B, C) needs to be aligned to 128 bits

# Data layout and strides

Question: how to store a matrix in memory

Row major: `A[i, j] => Adata[i * A.shape[1] + j]`

Column major: `A[i, j] => Adata[j * A.shape[0] + i]`

Strides format: `A[i, j] => Adata[i * A.strides[0] + j * A.strides[1]]`

# Discussion about strides

**Advantages:** can perform transformation/slicing in zero copy way

- Slice: change the begin offset and shape
- Transpose: swap the strides
- Broadcast: insert a stride equals 0

**Disadvantages:** memory access becomes not continuous

- Makes vectorization harder
- Many linear algebra operations may require compact the array first

# Parallelization

```
void vecadd(float* A, float *B, float* C) {
   #pragma omp parallel for
   for (int i = 0; i < 64; ++i) {
      float4 a = load_float4(A + i*4);
      float4 b = load_float4(B + i*4);
      float4 c = add_float4(a, b);
      store_float4(C * 4, c);
  }
}
```

Executes the computation on multiple threads

# Outline

General acceleration techniques

Case study: matrix multiplication

# Vanilla matrix multiplication

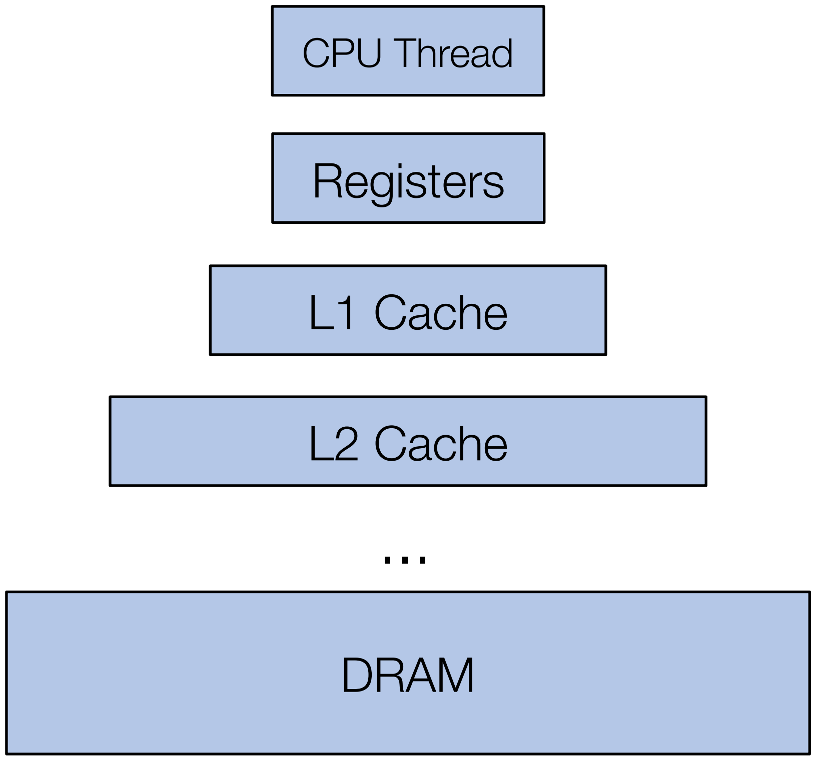
```
Compute C = dot(A, B.T)
```

```
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)

# Memory hierarchy on modern CPUs




**Latency**

Source: Latency numbers every programmer should know

0.5 ns

7ns 14x L1 cache

200ns 20x L2 cache, 200x L1 cache

# Architecture aware analysis

```
dram 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;
   }
}
```

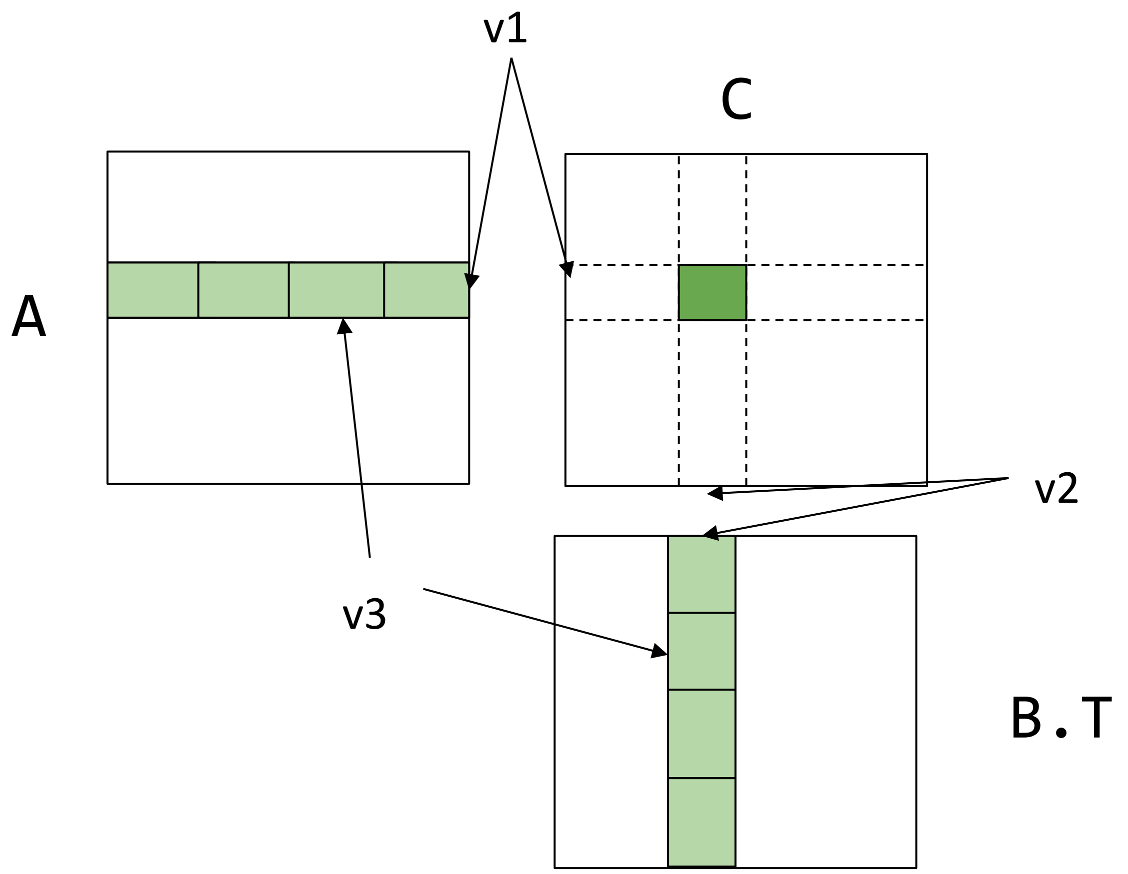
```
A’s dram->register time cost:  n^3
B’s dram->register time cost:  n^3
A’s register memory cost :  1
B’s register memory cost :  1
C’s register memory cost :  1
```

**Load cost:** 2 * dramspeed * n^3
**Register cost:** 3

# Register 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.T);
    }
    C[i][j] = c;
  }
}
```



# Register 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.T);
    }
    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:   v2*v3
C’s register memory cost:   v1*v2
```

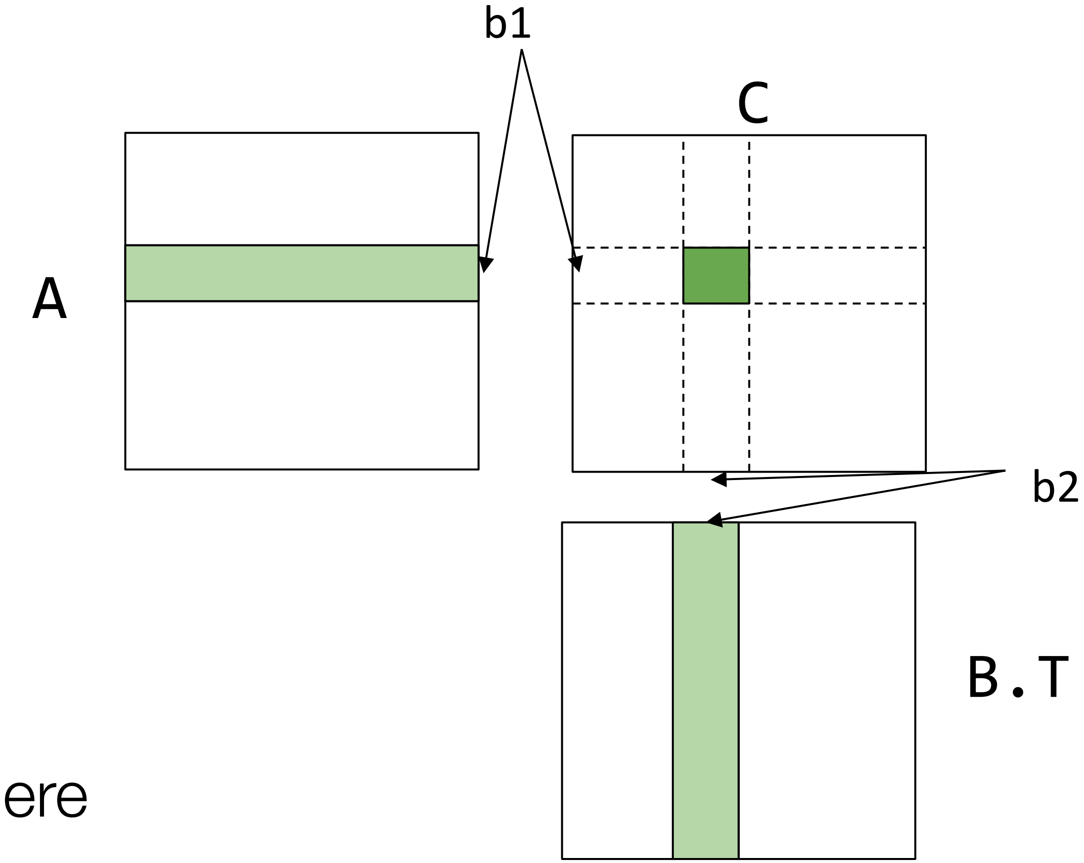
```
load cost:  dramspeed * (n^3/v2 + n^3/v1)
Register cost: v1*v3 + v2*v3 + v1*v2
```

# 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.T);
   }
}
```

Sub-procedure, can apply register tiling here



# 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.T);
   }
}
```

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

**Constraints:**

- b1 * n + b2 * n < l1 cache size
- To still apply register blocking on dot
  - b1 % v1 == 0
  - b2 % v2 == 0

# Putting it together

```
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; y < b/v2; ++y) {
         register float c[v1][v2] = 0;
         for (int k = 0; k < n; ++k) {
            register float ar[v1] = a[x][k];
            register float br[v2] = b[y][k];
            C += dot(ar, br.T)
         }
       }
   }
}
```

**load cost:**

```
l1speed * (n^3/v2 + n^3/v1) +
dramspeed * (n^2 + n^3/b1)
```

# Key insight: memory load reuse

```
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.T);
      }
      C[i][j] = c;
   }
}
```

```
a get reused v2 times
b get reused v1 times

A’s dram->register time cost:   n^3/v2
B’s dram->register time cost:   n^3/v1
```

# Common reuse patterns

```
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 enables reuse of A for v times.
```

# Discuss: possible reuse pattern in 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])
```

# Outline

General acceleration techniques

Case study: matrix multiplication