https://haoailab.com/cse291-s26/

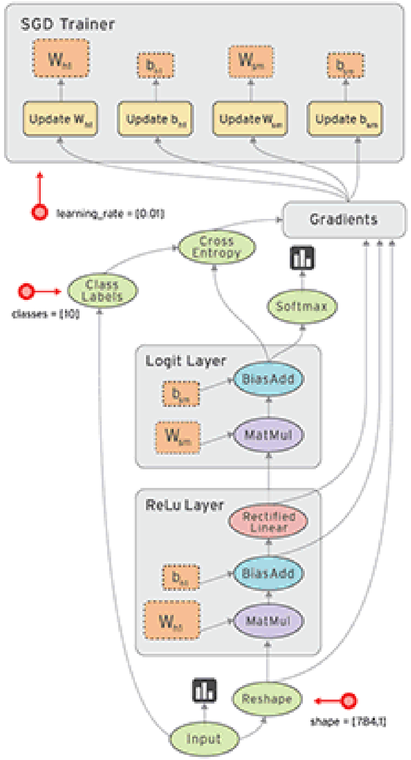
CSE/DSC 291: Deep Learning Systems
Spring 2026

LLM, diffusion, and case studies

Optimizations and Parallelization

Basics

ML System Overview



Dataflow Graph

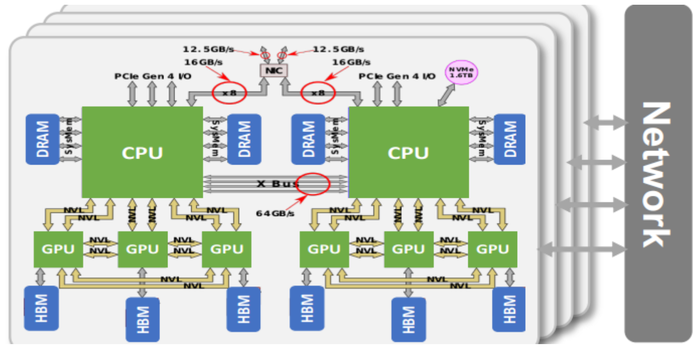
Autodiff

Graph Optimization

Parallelization

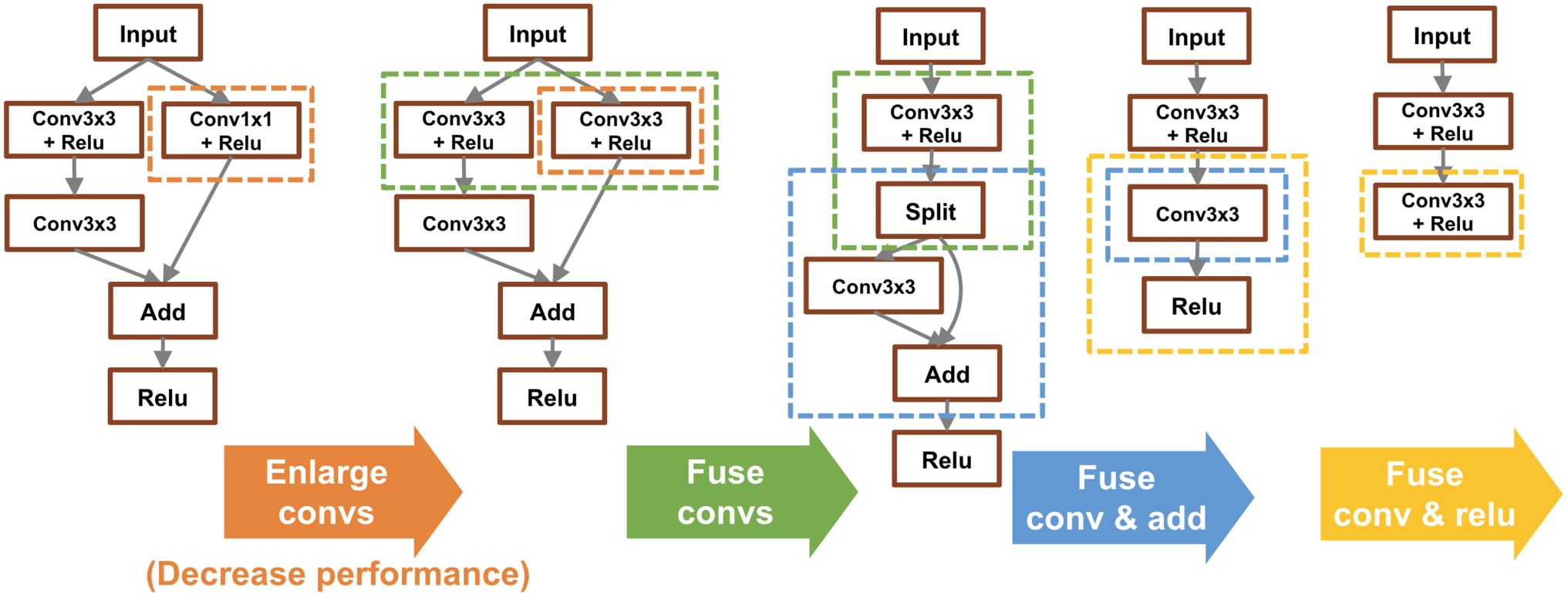
Runtime: schedule / memory

Operator optimization/compilation



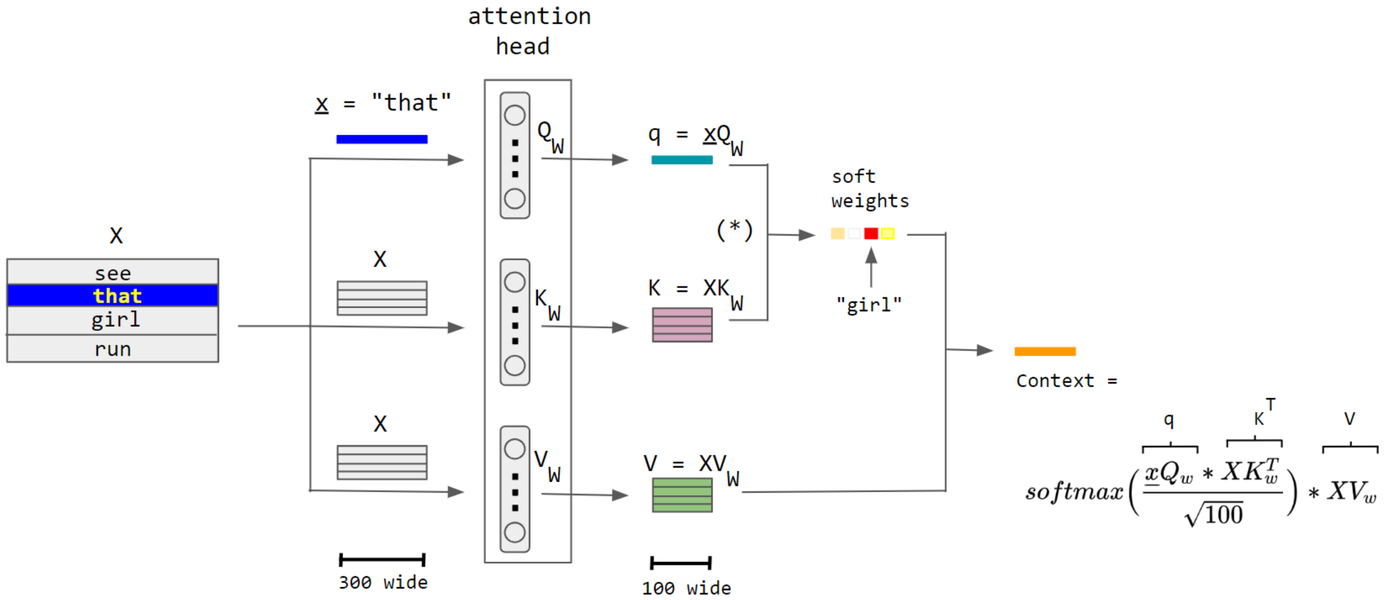
Dataflow Graph
Autodiff
Graph Optimization
Parallelization
Runtime: schedule / memory
Operator

Motivating Example: we can go further



- Does each step become faster than previous step?
- How does it perf on different hardware?

Motivating Example 2: Attention




```
# Original
Q = matmul(W_q, h)
K = matmul(W_k, h)
V = matmul(W_v, h)

# Merged QKV
QKV = matmul(concat(W_q, W_k, W_v), h)
```

- Why merged QKV is faster?

Arithmetic Intensity

$$AI = \#ops \,/\, \#bytes$$

Arithmetic Intensity

```
void add(int n, float*   A, float*   B, float*   C){
   for   (int   i=0; i<n;  i++)
      C[i]  = A[i]  + B[i];
}
```

Two loads, one store per math op
(arithmetic intensity = 1/3)

1. Read A[i]
2. Read B[i]
3. Add A[i]+B[i]
4. Store C[i]

Which program performs better? Program 1

```
void add(int n, float* A, float* B, float* C){
   for (int i=0; i<n; i++)
      C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
   for (int i=0; i<n; i++)
      C[i] = A[i] * B[i];
}

float* A, *B, *C, *D, *E, *tmp1, *tmp2;
//  assume arrays are allocated here
//  compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

Two loads, one store per math op (arithmetic intensity = 1/3)

Two loads, one store per math op (arithmetic intensity = 1/3)

Overall arithmetic intensity = 1/3

Which program performs better? Program 2

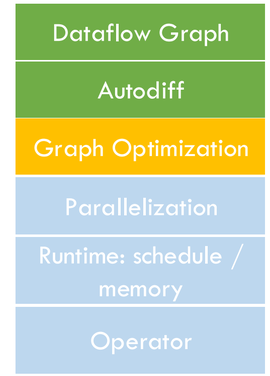
```
float* A, *B, *C, *D, *E, *tmp1, *tmp2;
// assume arrays are allocated here
// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);



void fused(int n, float* A, float* B, float* C, float* D,
   float* E) {
   for (int i=0; i<n; i++)
      E[i] = D[i] + (A[i] + B[i]) * C[i];
}
// compute E = D + (A + B) * C
fused(n, A, B,C, D, E);
```

Overall arithmetic intensity = 1/3

Four loads, one store per 3 math ops
arithmetic intensity = 3/5



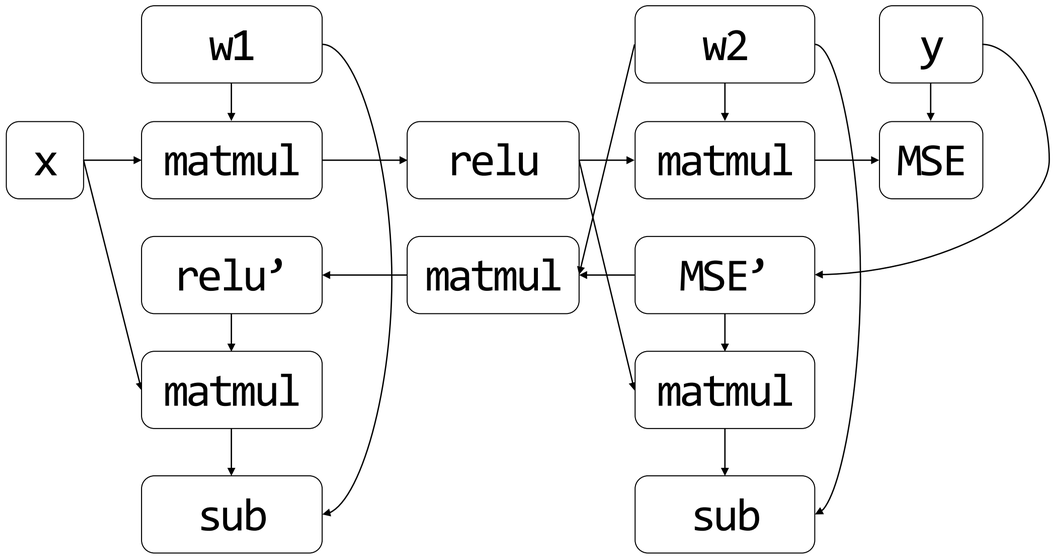
How to perform graph optimization?

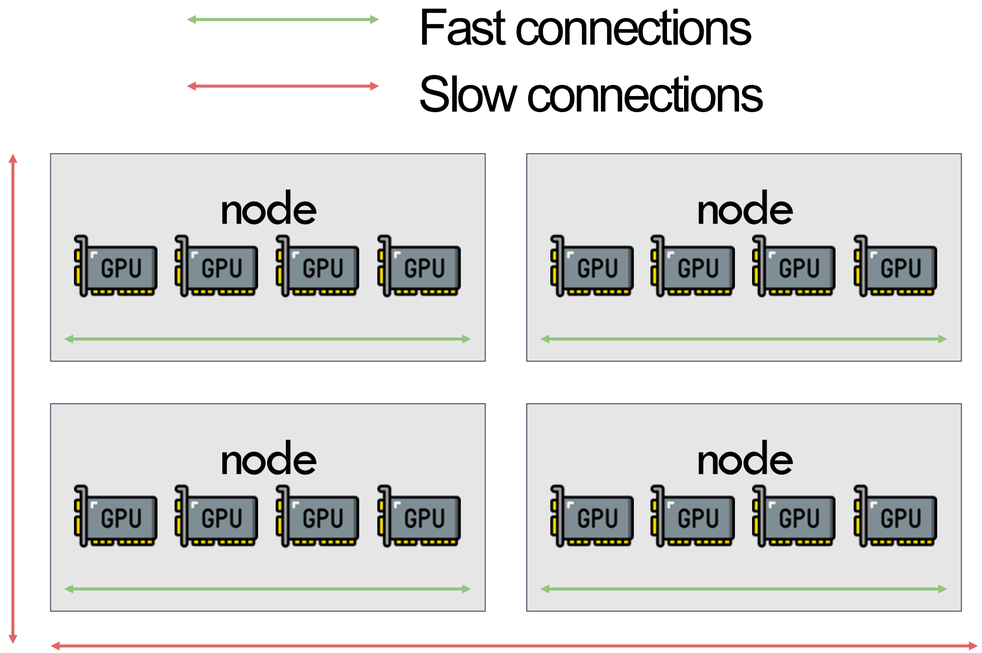
- Writing rules / template
- Auto discovery

Parallelization

- Goal: parallelize the graph compute over multiple devices

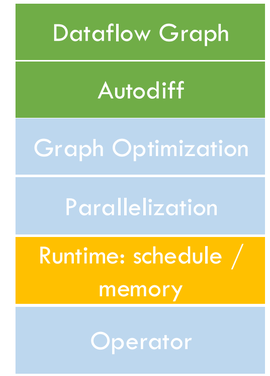
How to partition the computational graph on the device cluster?





Parallelization Problems

- How to partition
- How to communicate
- How to schedule
- Consistency
- How to auto-parallelize?



Runtime and Scheduling

- Goal: schedule the compute/communication/memory in a way that
  - As fast as possible
  - Overlap communication with compute
  - Subject to memory constraints

Operator Implementation

- Goal: get the fastest possible implementation of
  - Matmul
  - Conv2d?
- For different hardware: V100, A100, H100, phone, TPU
- For different precision: fp32, fp16, fp8, fp4
- For different shape: conv2d_3x3, conv2d_5x5, matmul2D, 3D, attention

High-level Picture

Data

Model

Compute

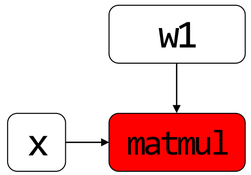
❓ $\{x_i\}^n{}_{i=1}$

✅ Math primitives (mostly matmul)

❓ Make them run on (clusters of) different kinds of hardware

✅ A repr that expresses the computation using primitives

Next: How to make operators run (fast) on devices?



Dataflow Graph

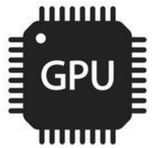
Autodiff

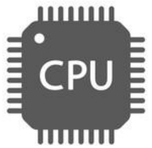
Graph Optimization

Parallelization

Runtime: schedule / memory

Operator optimization/compilation





Our Goal in This Layer: Maximize Arithmetic Intensity

$$\max \text{AI} = \#\text{ops} \,/\, \#\text{bytes}$$

Next

- How we can make operator fast in general
- Case study: Matmul
- GPU architecture and programming

How we can make operators fast in general

- Vectorization
- Data layout
- Parallelization

Using vectorized operations: array add

Why vectorized is faster than unvectorized?

```
Float A[256], B[256], C[256]
For (int i = 0; i < 256; ++i) {
   C[i] = A[i] + B[i]
}
```

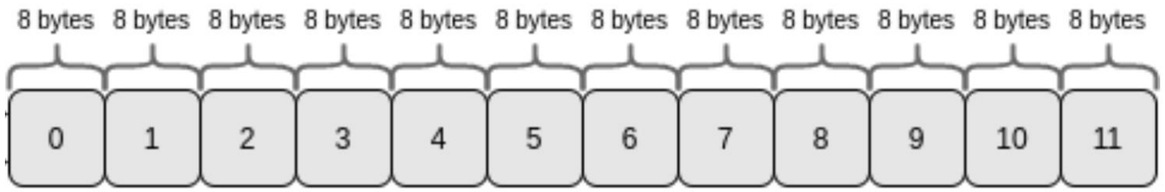
unvectorized

```
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);
```

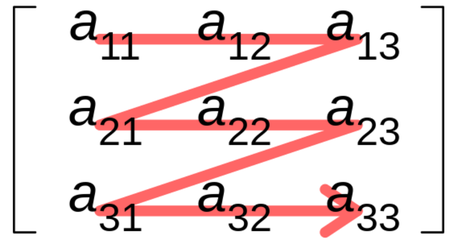
vectorized

Data Layout: make read/write faster

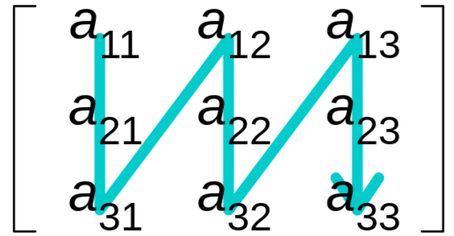
- How to store a matrix in memory
  - Data in memory are stored sequentially (no tensor awareness)
- Row Major: A[i, j] = A.data[i*A.shape[1] + j]
- Column major: A[i, j] = A.data[j*A.shape[0] + i]



Row-major order



Column-major order

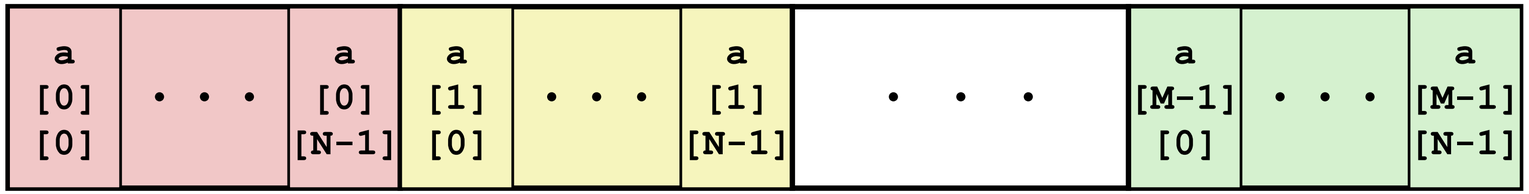


Be aware of your data layout

Assuming row-major array

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



How to improve the above program?

MCQ Time

Data

❓ $\{x_i\}^n{}_{i=1}$

ML Systems Store Data in:

A. Row major

B. Col major

C. Strides format: A[i, j] = A.data[offset + i*A.strides[0] + j * A.strides[1]]

Strides in High-dimension

Offset: the offset of the tensor relative to the underlying storage

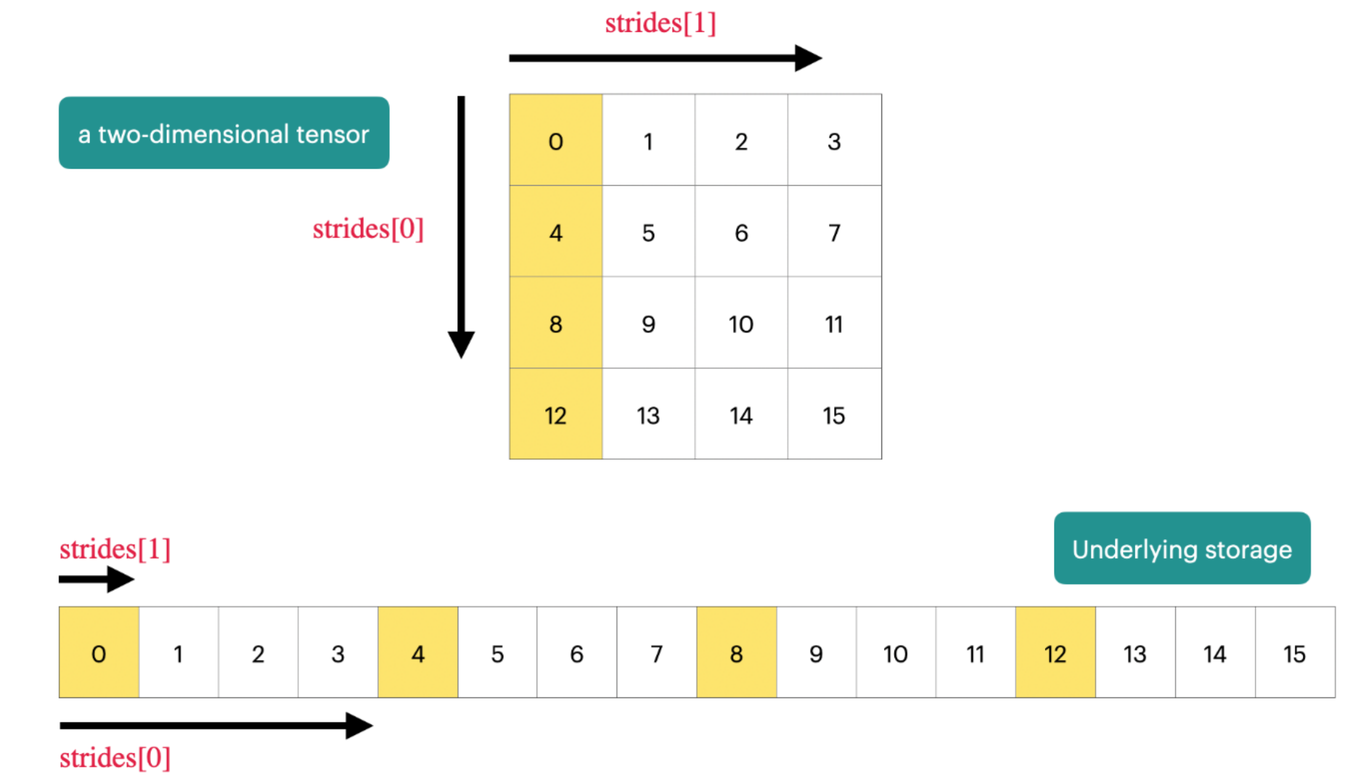
Strides: strides[i] indicates how many "elements" need to be skipped in memory to move "one unit" in the i-th dimension of the tensor

Python

```
A[i0][i1][i2]... = A_internal[
    stride_offset
    + i0 * A.strides[0]
    + i1 * A.strides[1]
    + i2 * A.strides[2]
    + ...
    + in-1 * A.strides[n-1]
]
```

Strides format

- What we have when:
  - A.strides[0] = 1,
  - A.strides[1] = A.shape[0]?
- What we have when:
  - A.strides[0] = A.shape[1]
  - A.strides[1] = 1,
- Strides offers more flexibility



Questions

- If a tensor of shape [1, 2, 3, 4] is stored contiguous in memory following row Major, write down its strides?

```
torch.arange(0, 24).reshape(1, 2, 3, 4)
print(t)
# tensor([[[[ 0,  1,  2,  3],
#           [ 4,  5,  6,  7],
#           [ 8,  9, 10, 11]],

#          [[12, 13, 14, 15],
#           [16, 17, 18, 19],
#           [20, 21, 22, 23]]]])

print(t.stride())
# (24, 12, 4, 1)
```

Why we bother saving “strides” when saving tensors

- Strides can separate the underlying storage and the view of the tensor

torch.Tensor.view

Tensor.view(*shape) → Tensor

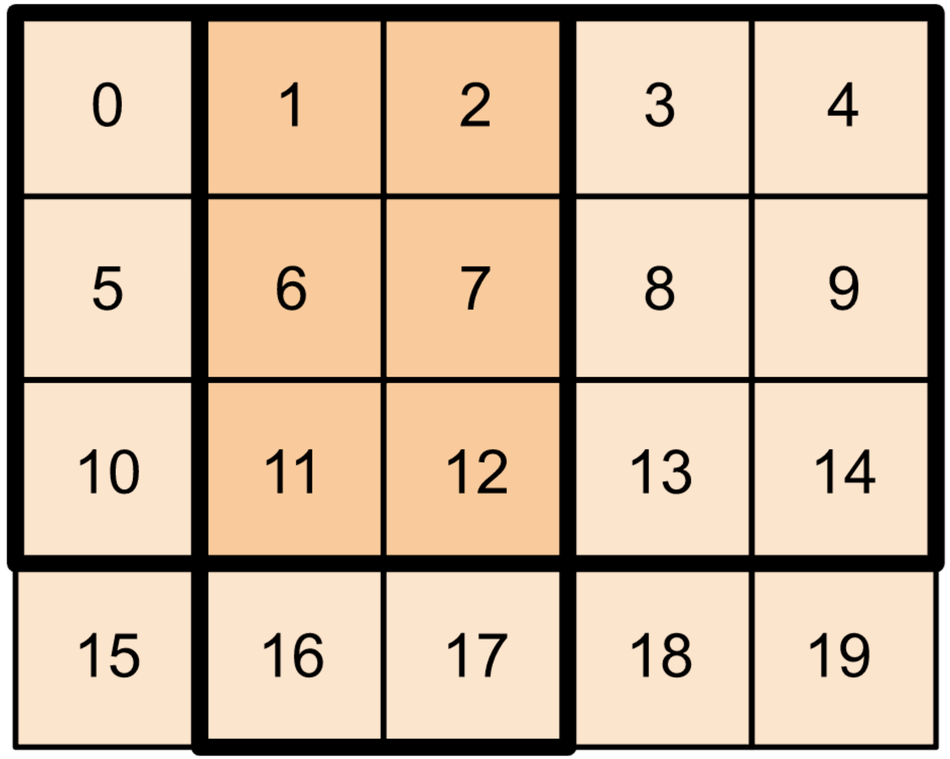
```
>>> x = torch.randn(4, 4)
>>> x.size()
torch.Size([4, 4])
>>> y = x.view(16)
>>> y.size()
torch.Size([16])
>>> z = x.view(-1, 8)  #
>>> z.size()
torch.Size([2, 8])
```

- Enable zero-copy of some very frequently used operators

Operator: Slice

- Change the offset by +1
- Reduce the shape to [3, 2]

- Note: zero copy



Operator: Transpose

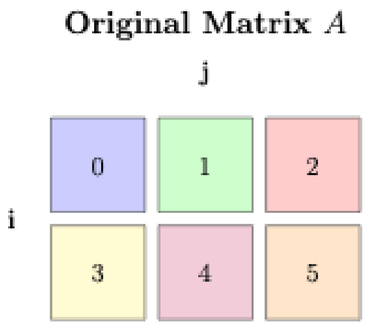
- How to do Transpose?

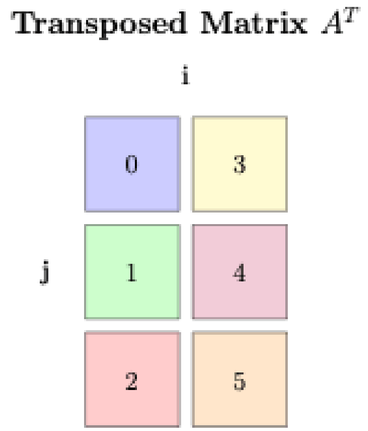
```python
print(t.stride())
# (24, 12, 4, 1)

print(t.permute((1, 2, 3, 0)).is_contiguous())
# True

print(t.permute((1, 2, 3, 0)).stride())
# (12, 4, 1, 24)

print_internal(t.permute((1, 2, 3, 0)))
# tensor([0,  1,  2,  3,
#         4,  5,  6,  7,
#         8,  9, 10, 11,
#         12, 13, 14, 15,
#         16, 17, 18, 19,
#         20, 21, 22, 23])
```

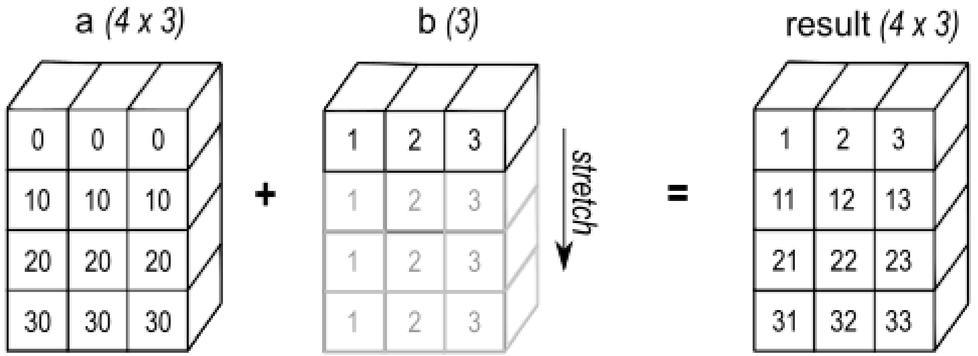




- Note 1: zero copy
- Note 2: underlying storage is unchanged

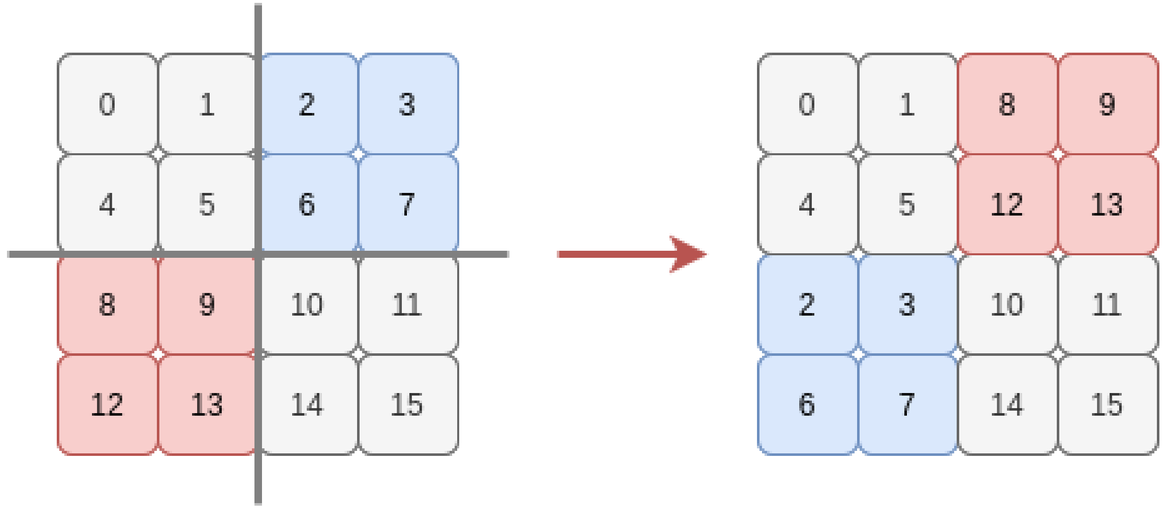
Broadcast

- Question: how to do broadcast?



- b.strides=[1], b.shape=[1,3], b.data=[1, 2, 3]
- After broadcast: b.shape=[4,3], b.data=[1, 2, 3], b.strides=?
- Recall the def of strides: A[i, j] = A.data[offset + i*A.strides[0] + j * A.strides[1]]

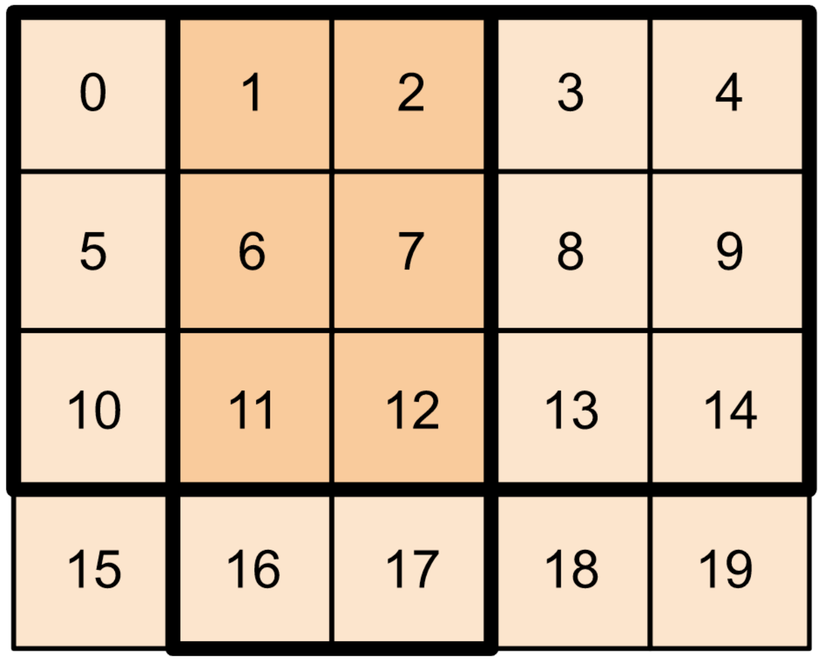
Home Exercise: Swapping tiles




```
>>> a = np.arange(16).reshape(4, 4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> np.vstack((np.hstack((a[0:2, 0:2], a[2:4, 0:2])), np.hstack((a[0:2, 2:4], a[2:4, 2:4]))))
array([[ 0,  1,  8,  9],
       [ 4,  5, 12, 13],
       [ 2,  3, 10, 11],
       [ 6,  7, 14, 15]])
```

Problems of Strides

- Memory Access may become not continuous
  - Many vectorized ops requires continuous storage
  - What's the underlying storage after slice?



TORCH.TENSOR.CONTIGUOUS

Tensor.contiguous(*memory_format=torch.contiguous_format*) → Tensor

Returns a contiguous in memory tensor containing the same data as `self` tensor. If `self` tensor is already in the specified memory format, this function returns the `self` tensor.

Parameters

memory_format (`torch.memory_format` , optional) – the desired memory format of returned Tensor. Default: `torch.contiguous_format` .

Parallelization (Elementise)

We'll com back to this later

- How to parallelize the loop?

```
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);
```

vectorized

```
#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);
}
```

Vectorized & parallelized

Summary

- Vectorization
  - Leverage platform-specific vectorized functions
  - reduce seek time
- Data layout
  - Stride format
  - Zero copy
  - Enable fast array-manipulation: slice, transpose, broadcast, etc.
- Parallelization on CPUs

Next

- How to make operators fast in general?
  - ✅ Vectorize
  - ✅ Data layout
  - ✅ Parallelization (at the operator level)
- ❓Matmul-specific optimization
- GPUs and accelerators
  - High-level Idea
  - The accelerator market

What is Matmul in Code?

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];
    }
  }
```

- What is the time complexity of 2D matmul?
- O(n^3)

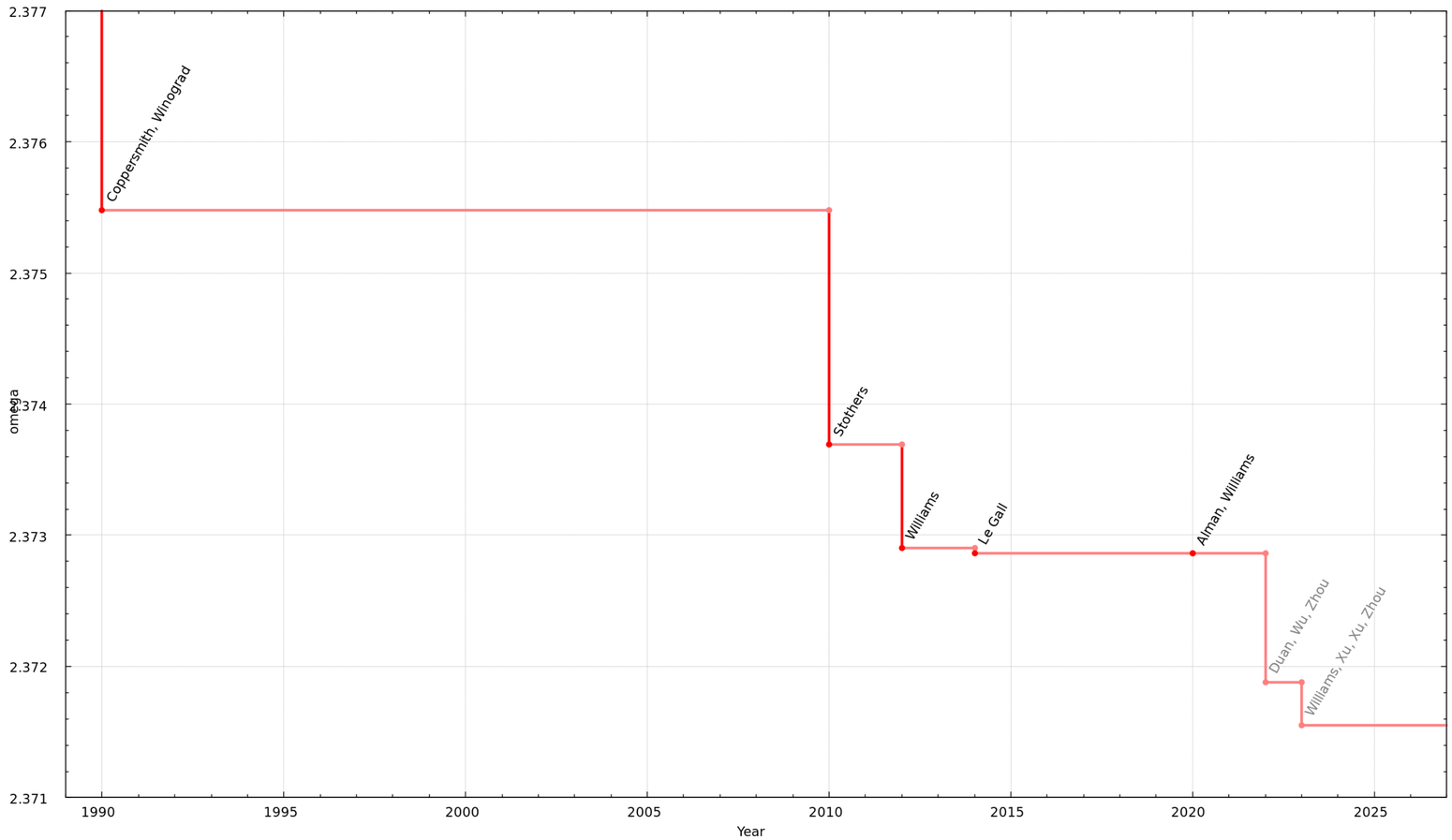
- What is the best complexity we can achieve?
- O(n^2.371552)

Matmul Complexity

Not a good area to do research ☺

Timeline of matrix multiplication exponent

Year	Bound on omega	Authors
1969	2.8074	Strassen[1]
1978	2.796	Pan[10]
1979	2.780	Bini, Capovani [it], Romani[11]
1981	2.522	Schönhage[12]
1981	2.517	Romani[13]
1981	2.496	Coppersmith, Winograd[14]
1986	2.479	Strassen[15]
1990	2.3755	Coppersmith, Winograd[16]
2010	2.3737	Stothers[17]
2012	2.3729	Williams[18][19]
2014	2.3728639	Le Gall[20]
2020	2.3728596	Alman, Williams[21][22]
2022	2.371866	Duan, Wu, Zhou[23]
2024	2.371552	Williams, Xu, Xu, and Zhou[2]

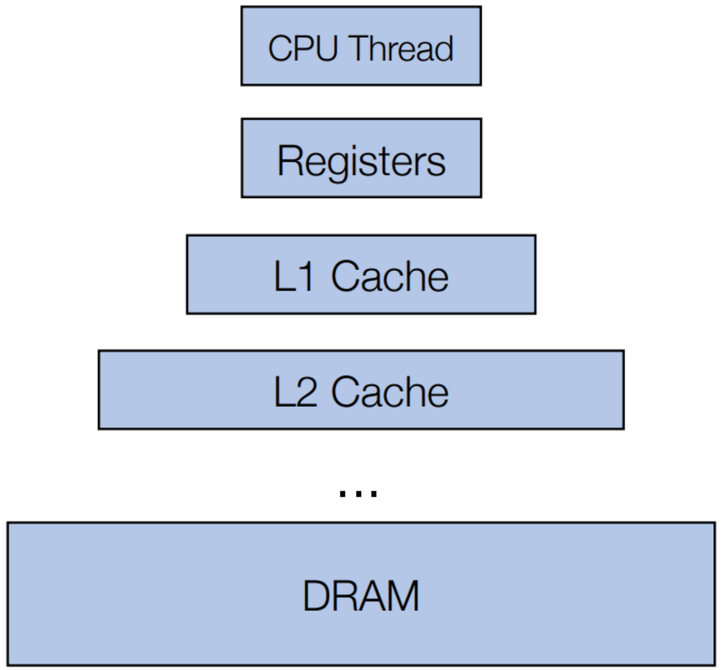


How to Make Matmul Fast

$$\max \text{AI} = \#\text{ops} / \#\text{bytes}$$

Recall: Memory Hierarchy

- Ideally: we want everything to be local to processors (In registers)
- But registers are expensive and small, hence memory hierarhcy



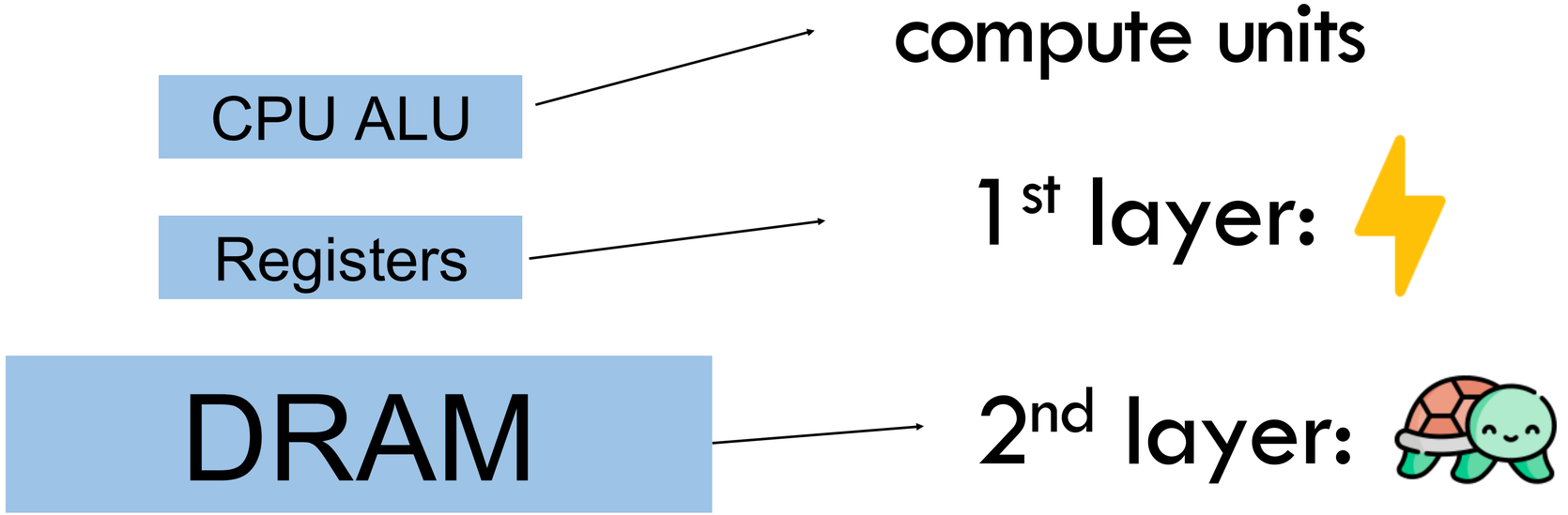
Latency
Source: Latency numbers every programmer should know

0.5 ns

7ns 14x L1 cache

200ns 20x L2 cache, 200x L1 cache

Simplify It a bit



Recall How to Estimate AI: count loads

```
float*  A, *B,  *C,  *D,  *E,  *tmp1, *tmp2;
//   assume arrays are allocated here
//   compute  E = D + ((A  + B) *  C)
add(n, A, B, tmp1);
mul(n,  tmp1,  C, tmp2);
add(n, tmp2,  D, E);



void fused(int  n, float*  A, float*  B, float*  C, float*  D,
   float*  E) {
   for  (int  i=0; i<n; i++)
      E[i]  = D[i]  + (A[i] + B[i]) *  C[i];
}
//   compute  E = D + (A + B) *  C
fused(n,   A,  B,C,    D, E);
```

Overall arithmetic intensity = 1/3

Four loads, one store per 3 math ops
arithmetic intensity = 3/5

Review Matmul loop

```
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;
   }
}
```

Read a n^3

Read b n^3

Write c n^2

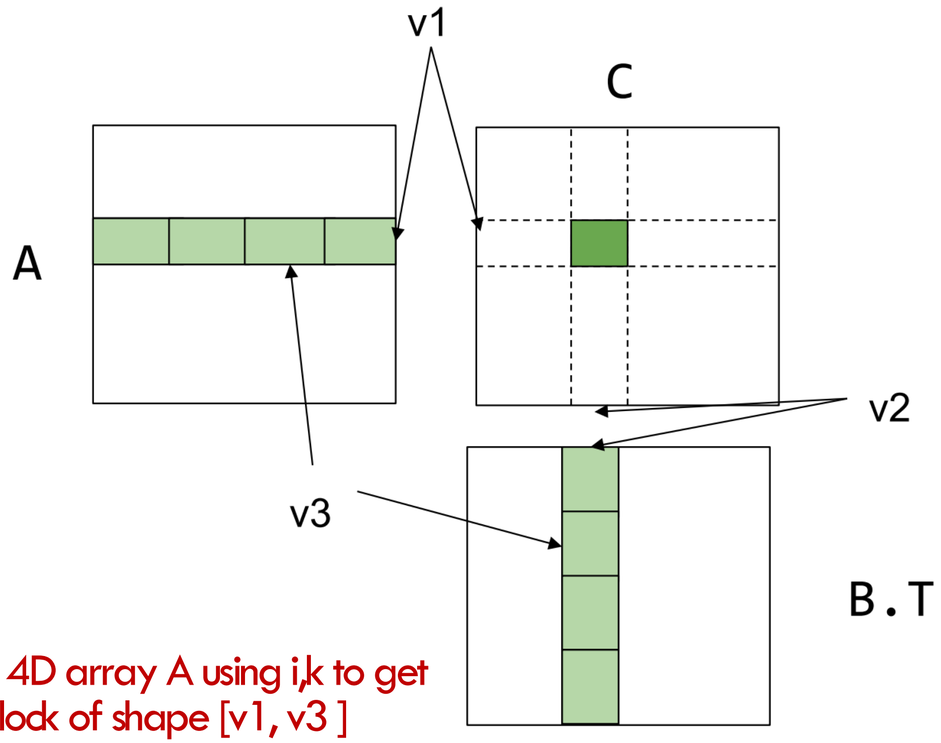
#reigsters needed:
1 + 1 + 1 =3

Read cost:
2 * n^3 * speed(dram -> register)

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;
   }
}
```



Indexing 4D array A using i,k to get a block of shape [v1, v3]

- Assigning the right block to a register block of shape [v1, v3]
- It should be: a[0:v1, 0:v3] = A[i][k]
- RHS is indexing, but LHS is not!

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;
   }
}
```

Read a	N^3 / v_2
Read b	N^3 / v_1
Write c	N^2

#reigsters needed:
v1*v3 + v2*v3 + v1*v2

Read cost:
(n^3/v2 + n^3 / v1) * speed(dram -> register)

Register Tiled Matrix Multiplication

- Q: is the load cost related to v3?

Read a N^3 / v_2

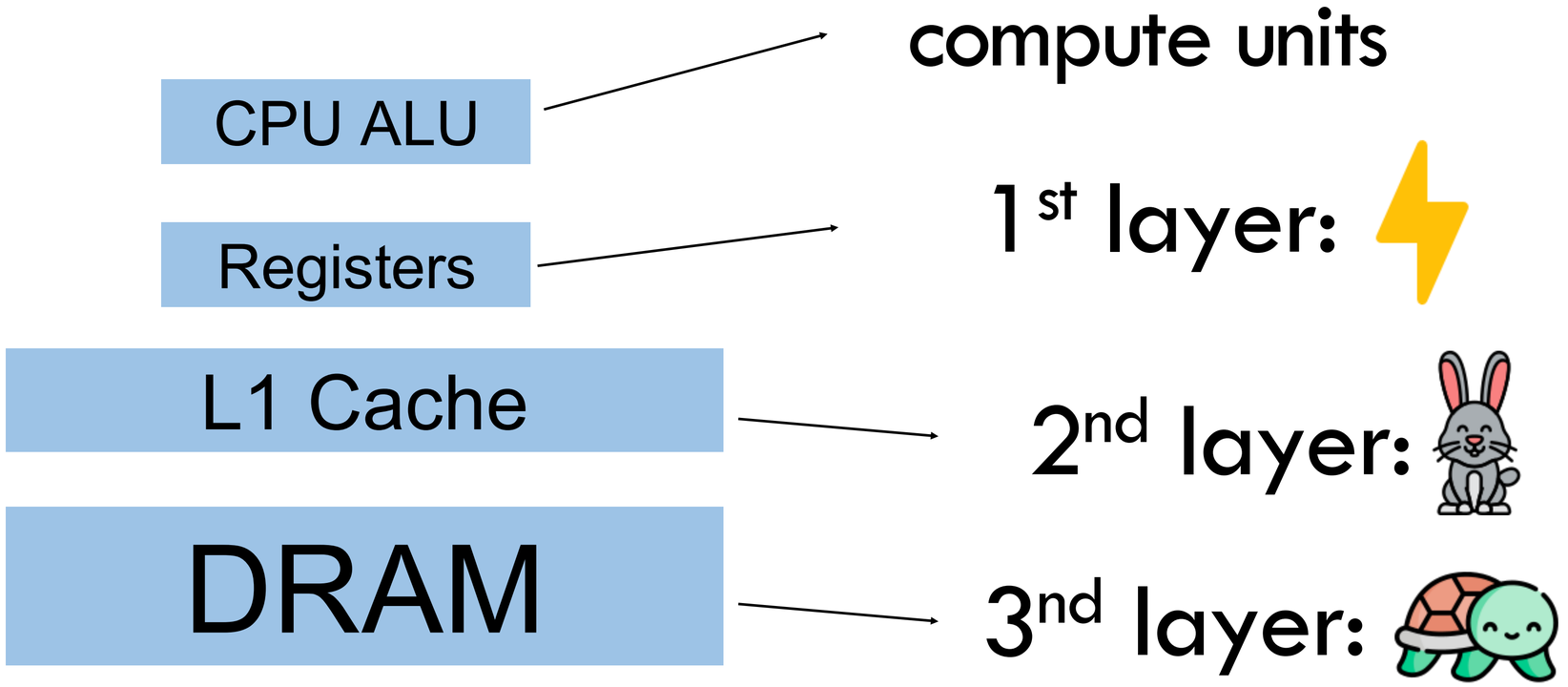
Read b N^3 / v_1

- Q: How to set v1 / v2?
  - What are the constraints?

#reigsters needed:
v1*v3 + v2*v3 + v1*v2

- Q: Why essentially can tiling reduce read cost?

Make it more complicated: Consider L1 cache



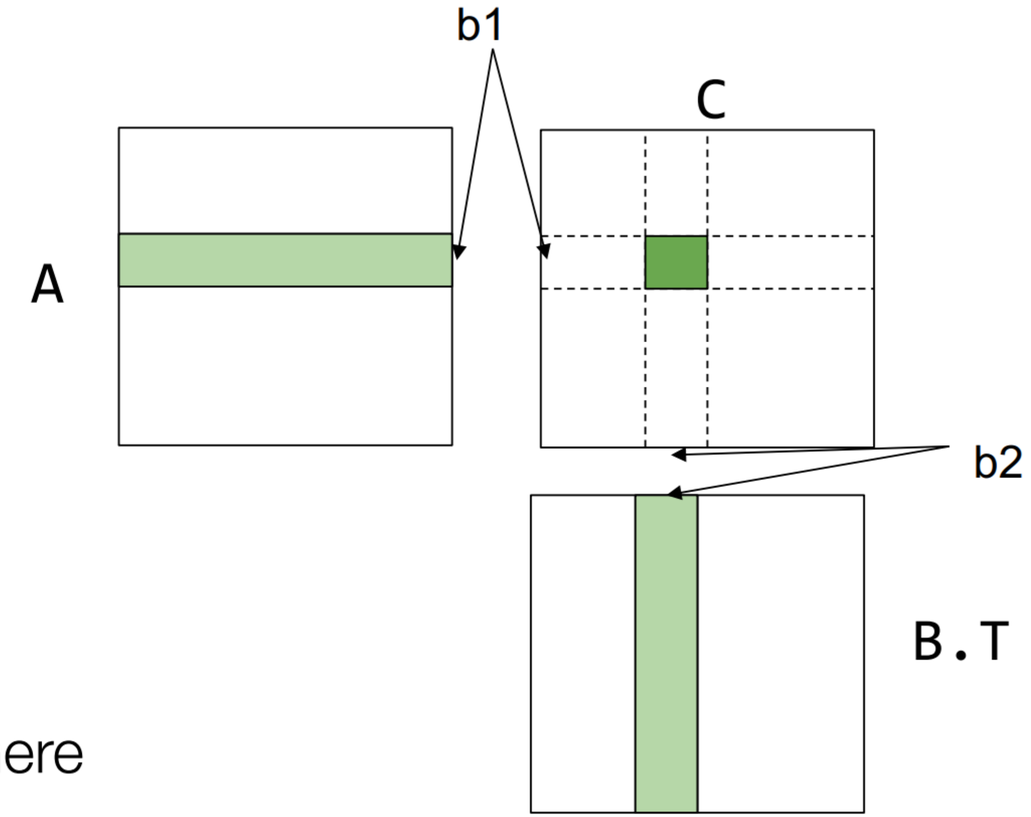
Cache-aware tiling

- We can further tile the array [b1][n] or [b2][n] using at the L1-> register level
- What's the required condition?

```
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);
   }
}
```

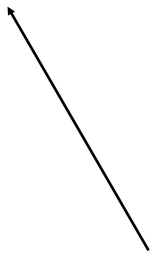
Sub-procedure, can apply register tiling here



Cache-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);
   }
}
```



Later we apply register tiling here

Data movement path:

1. Dram
2. Dram -> l1cache (cache tiling)
3. L1cache -> register (reg tiling)

Cache-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 cost:

n / b1 * n * b1 = n^2

B's dram -> l1 time cost:

n / b1 * n / b2 * b2 * n = n^3 / b1

Vs. previous untiled version?

s.t.

- b1 * n + b2 * n < L1 cache size

Putting Things 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 < b1/v1; ++x)
       for (int y = 0; y < b2/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)
         }
       }
   }
}
```

We set v3 = 1 (we know it does not matter)

Outside: cache tiling
Inside: register tiling

- Cache tiling using b1 and b2
- DRAM → L1 cache reads here

- Register tiling using v1 and v2
- L1→ register cache reads here

Putting Things 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 < b1/v1; ++x)
      for (int y = 0; y < b2/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)
        }
      }
  }
}
```

Outside: cache tiling
Inside: register tiling

Cost: dram -> l1

- n/b1*b1/v1*n*v1 = n^2
- n/b1*n/b2*b2/v2*n*v2=n^2/b1

Cost: l1 -> register:

- n / b1 * n / b2 * b1 / v1 * b2 / v2 * n * v1 = n^3 / v2
- n^3 / v1

In practice

- On CPUs: We have disk -> dram -> L2 -> L1 -> Register
- How to choose v1, v2, b1, b2, c1, c2?
- While we are reading from dram -> L2, can we concurrently read:
  - L2 -> L1
  - L1 -> register
- S.t. sizes of L2, L1, registers

Why tiling works: **reuse** loading

```
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 the dimension of j

Tile the j dimension by v1 enables resue of A for v1 times

Homework Candidate?

- Q: How to tile?

```
for n in range(0, N):
  for co in range(0, CO):
    for h in range(0, H):
      for w in range(0, W):
        for ci in range(0, CI):
          for kh in range(0, KH):
            for kw in range(0, KW):
              C[n,co,h,w] += A[n,co,h+kh,w+kw] x B[kh,kw,co,ci]
```

Simple spatial loops.

Stencil computation loops.

Reduction loop.

Reduction loops. But usually too small (<= 5) for parallelization.