https://hao-ai-lab.github.io/dsc291-s24/

DSC 291: ML Systems
Spring 2024

LLMs

Parallelization

Single-device Optimization

Basics

In-Class Quiz

Consist of 2 Components:

- Attendance check-in on iClicker app
- 15 minute quiz on Gradescope (UCSD email)
  - Will go over quiz in class after

Need to complete both to get credit

Quiz will open at 5:00PM and close at 5:15PM.

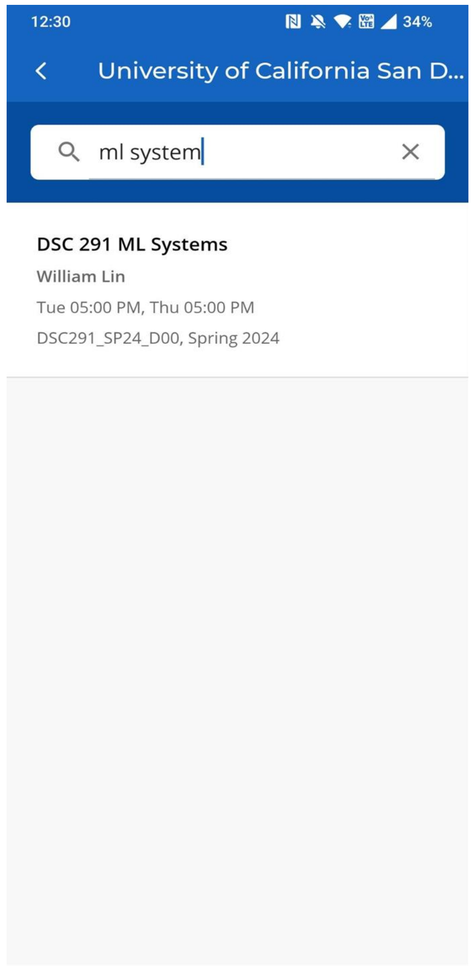
Without checking in on iClicker you cannot get credit!

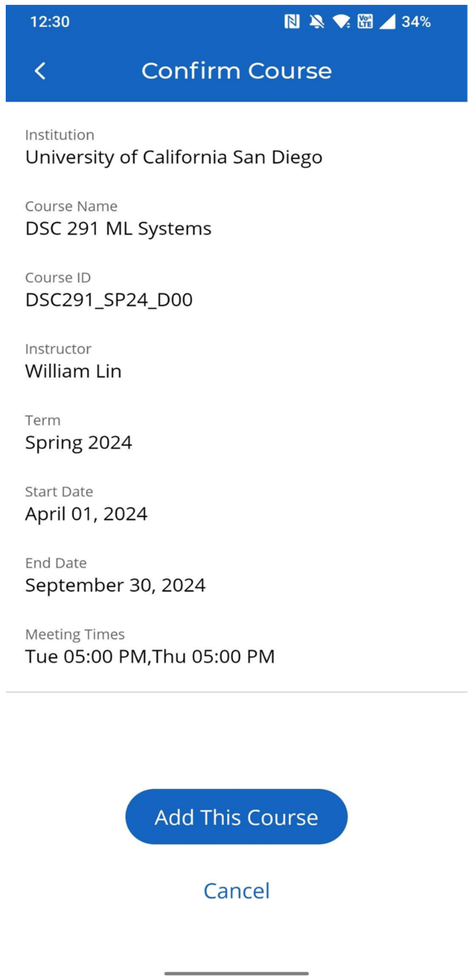
Try to check-in now

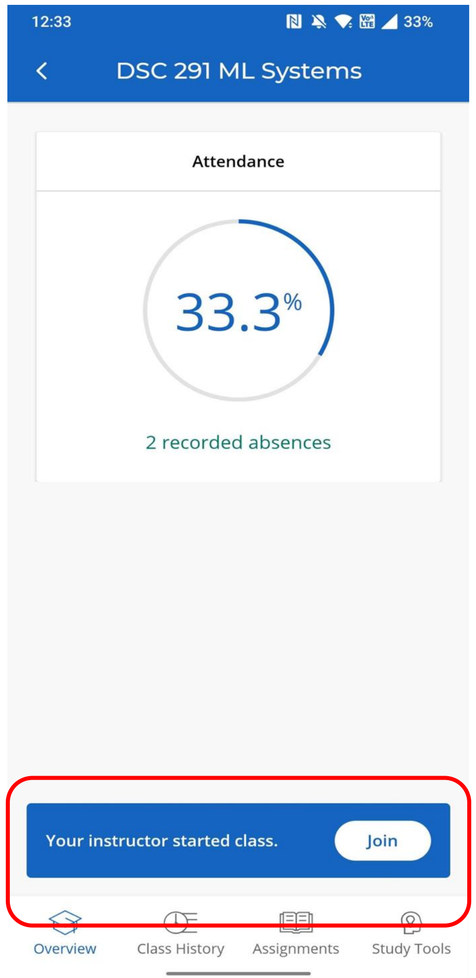
We are using iClicker App for attendance!

Try to check-in now

- Check-in to DSC 291 ML Systems on iclicker app

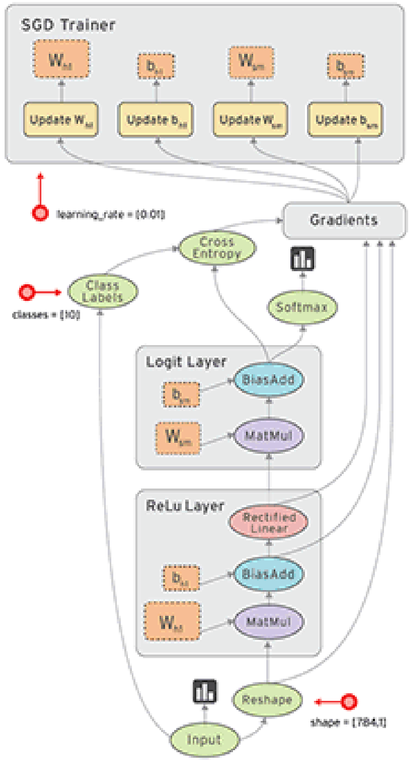






SCAN ME

Recap



Dataflow Graph

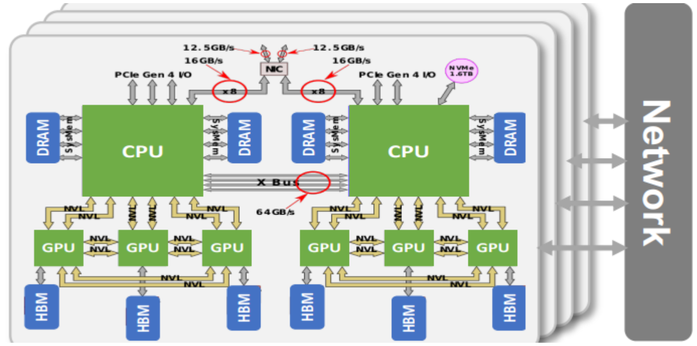
Autodiff

Graph Optimization

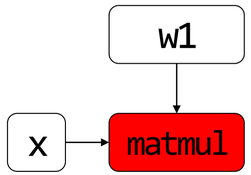
Parallelization

Runtime: schedule / memory

Operator optimization/compilation



Today



Dataflow Graph

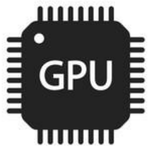
Autodiff

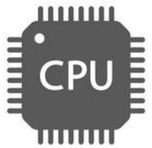
Graph Optimization

Parallelization

Runtime: schedule / memory

Operator optimization/compilation





Next 2 – 3 lectures

- Fundamentals: why/how we can make operator fast
- Case study 1: Matmul
- GPU architecture
- CUDA programming
- Case study 2: Matmul
- Roofline model

Big Goal: Maximize Arithmetic Intensity

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

Example: Program 2 performs better (graph-level)

```
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

General Op-level Techniques (on CPUs)

- Vectorization
- Data layout
- Parallelization
- Matmul-specific tricks
  - Tiling

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]
}
```

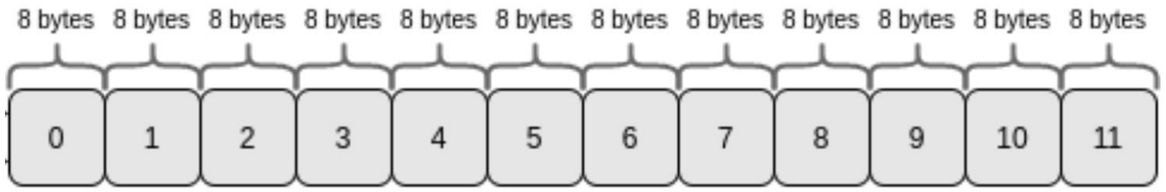
unvectorized

```
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);
```

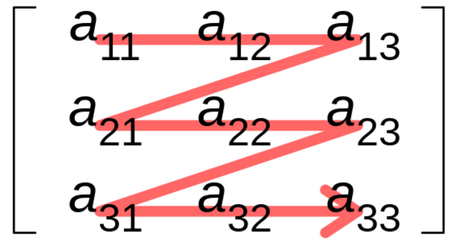
vectorized

Data Layout: make read/write faster

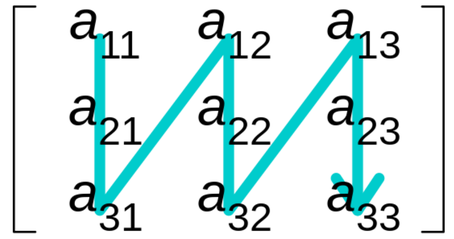
- How to store a matrix in memory
  - Data in memory are stored sequentially (no tensor awareness)
- Row Major: A[i, j] = A.data[i*A.shape[1] + j]
- Column major: A[i, j] = A.data[j*A.shape[0] + i]



Row-major order



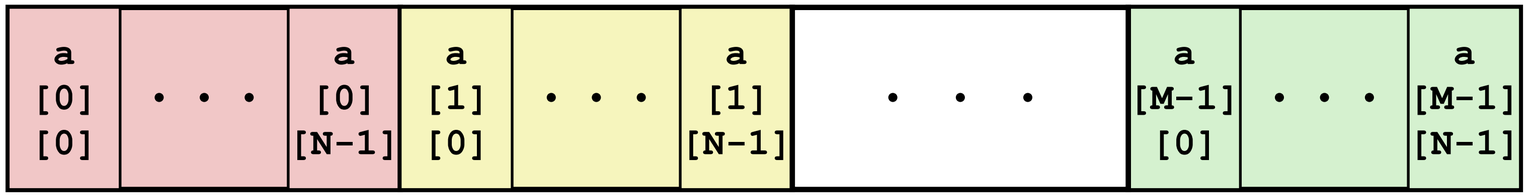
Column-major order



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?

Data Layout and Strides

- Row Major: A[i, j] = A.data[i*A.shape[1] + j]
- Column major: A[i, j] = A.data[j*A.shape[0] + i]
- 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

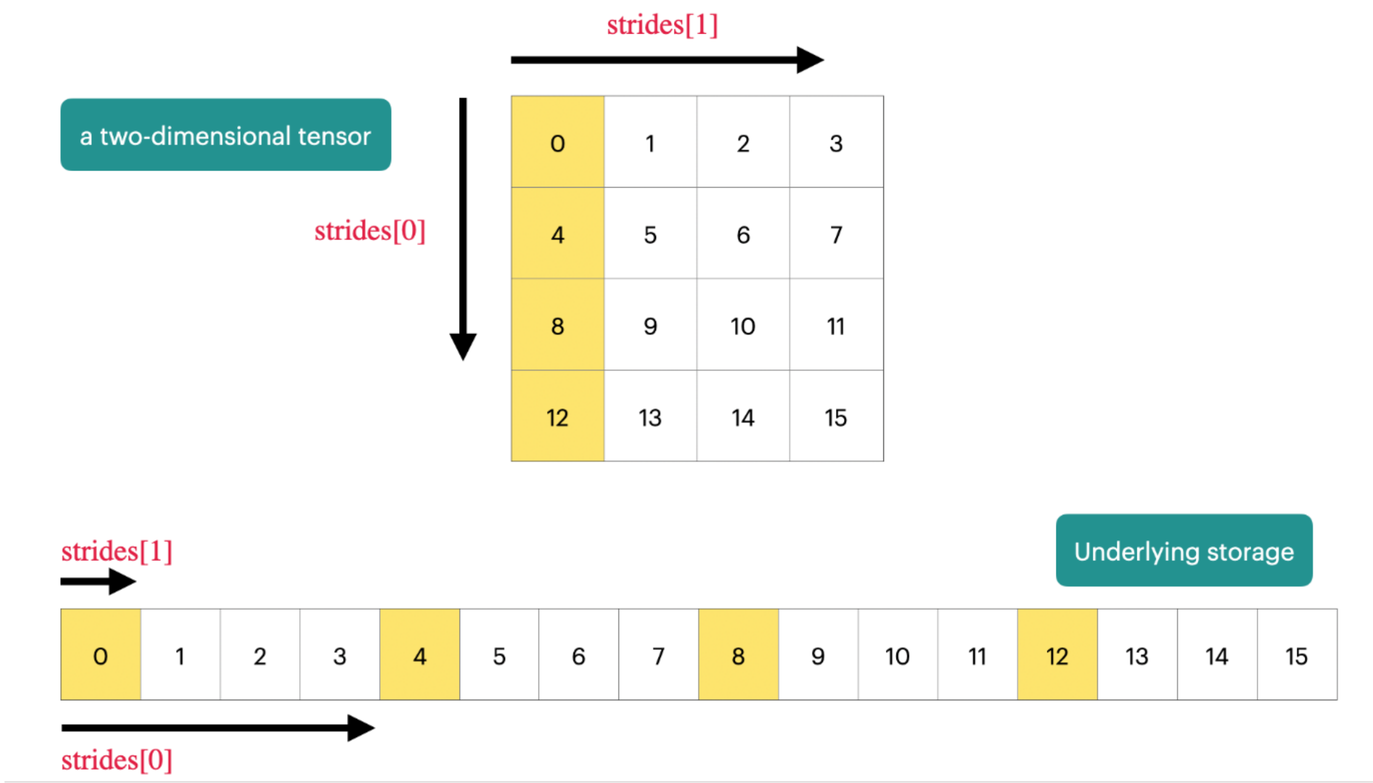
Strides: strides[i] indicates how many "elements" need to be skipped in memory to move "one unit" in the iii-th dimension of the tensor

Python

```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?

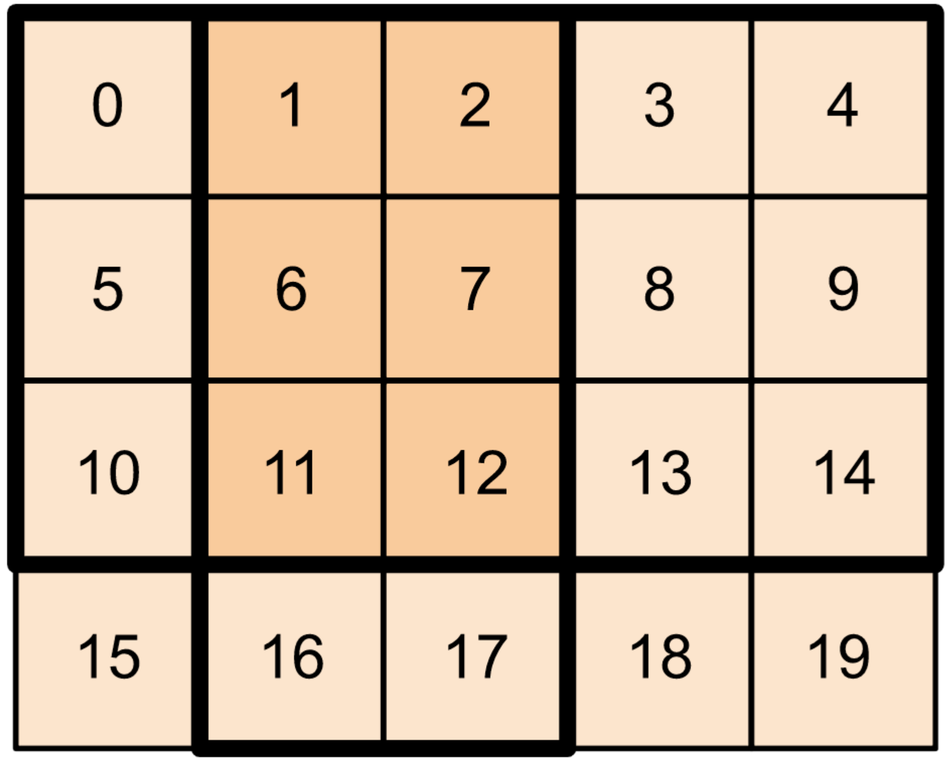
```
torch.arange(0, 24).reshape(1, 2, 3, 4)
print(t)
# tensor([[[[ 0,  1,  2,  3],
#           [ 4,  5,  6,  7],
#           [ 8,  9, 10, 11]],

#          [[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 -> Enable zero-copy of some very frequently used ops (recall “tensor.view” in pytorch)
  - Slice
  - Transpose (or reshape)
  - Broadcast

How to do Slice with strides

- Change the offset by +1
- Reduce the shape to [3, 2]
- Note: zero copy



Transpose

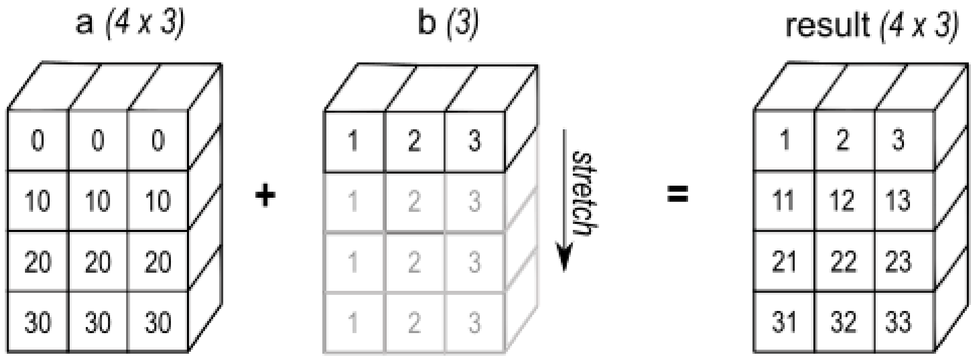
- How to do Transpose?
- Note: it is zero copy
- Note: What are the underlying Storage looking like?

Python

```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])
```

Broadcast

- Question: how to do broadcast?

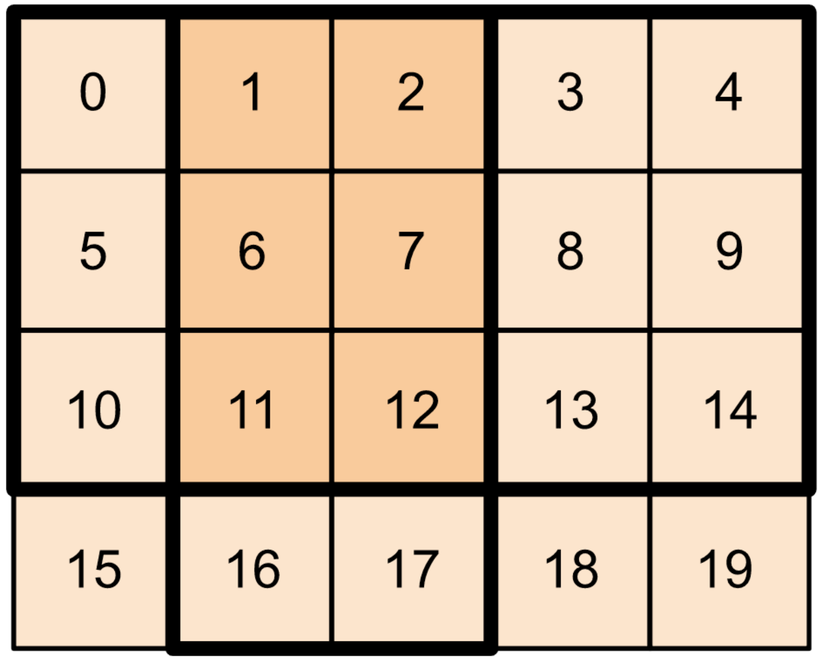


Python

```python
print(t.broadcast_to((2, 2, 3, 4)).is_contiguous())
# False

print(t.broadcast_to((2, 2, 3, 4)).shape)
# torch.Size([2, 2, 3, 4])

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

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

print_internal(t.broadcast_to((2, 2, 3, 4)))
# 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])
```

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

- 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
  - reduce seek time
- Data layout / strides
  - Convenient
  - Zero copy
- Parallelization

Next

- Fundamentals: why/how we can make operator fast
- **Case study: Matmul**
- GPU architecture and programming
- Roofline model

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];
    }
  }
```

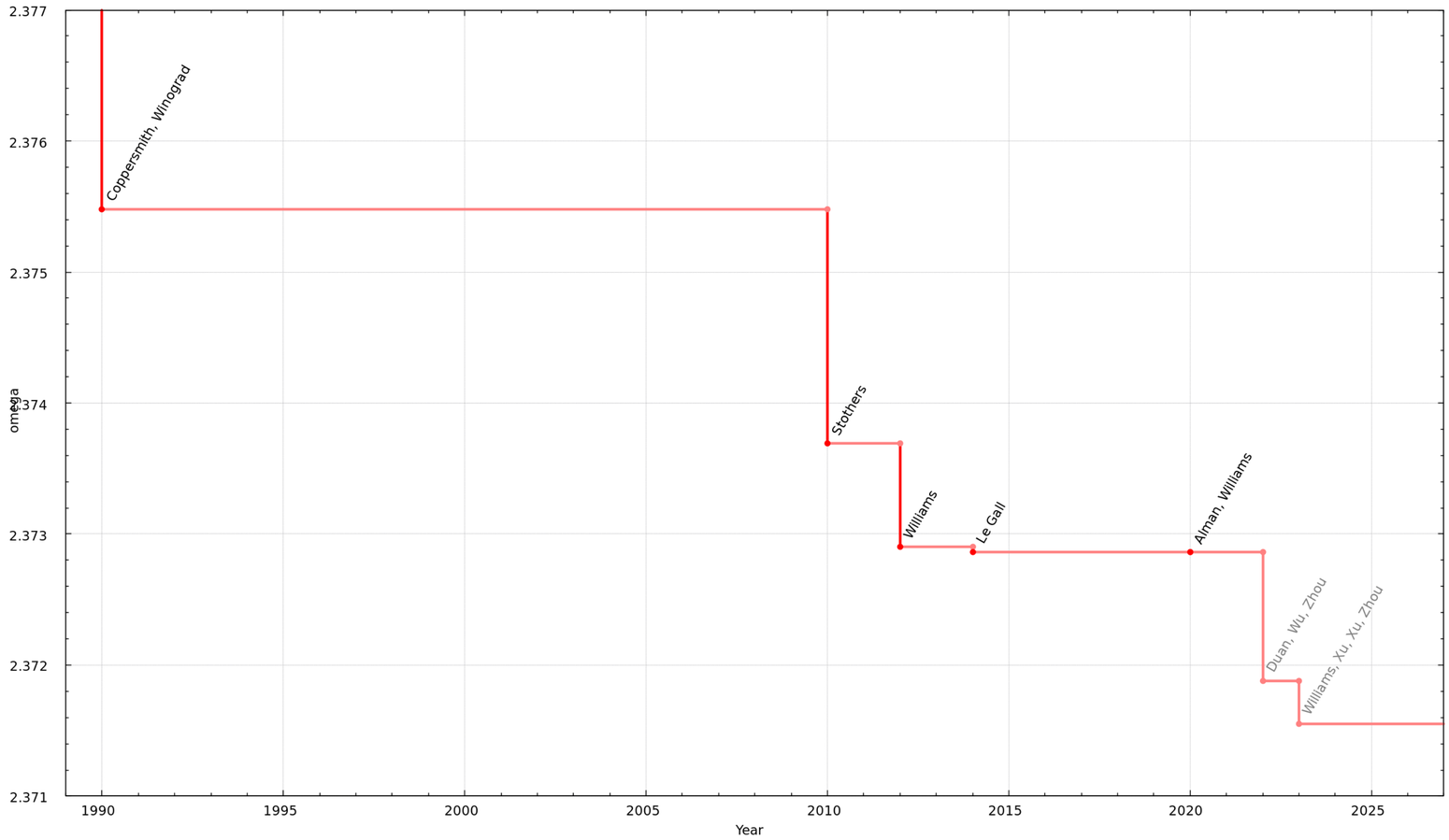
- What is the time complexity of 2D matmul?
- O(n^3)

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

Matmul Complexity

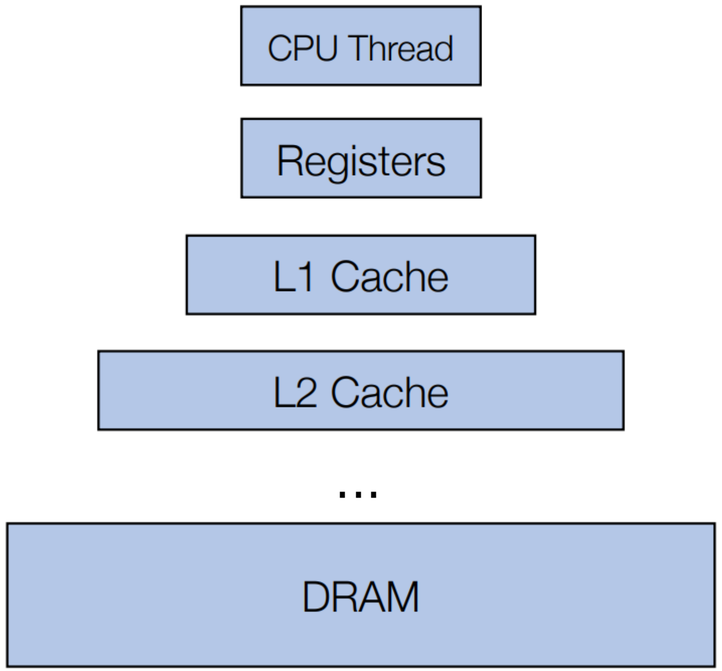
Timeline of matrix multiplication exponent

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]



We also care about I/O Because:

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



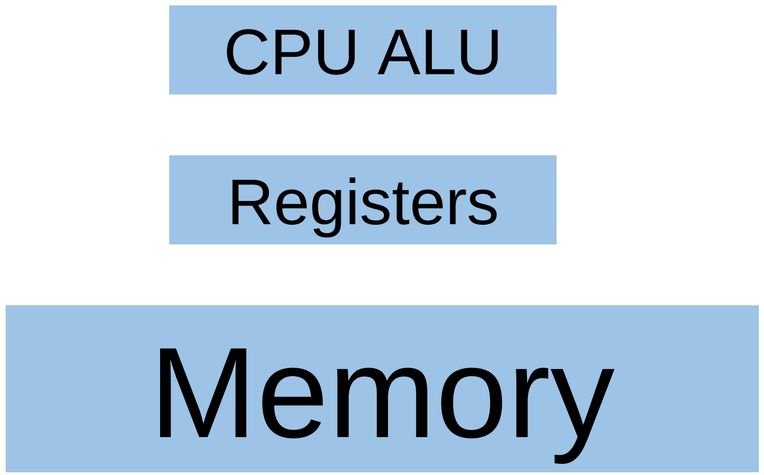
Latency
Source: Latency numbers every programmer should know

0.5 ns

7ns 14x L1 cache

200ns 20x L2 cache, 200x L1 cache

Simplify It a bit



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

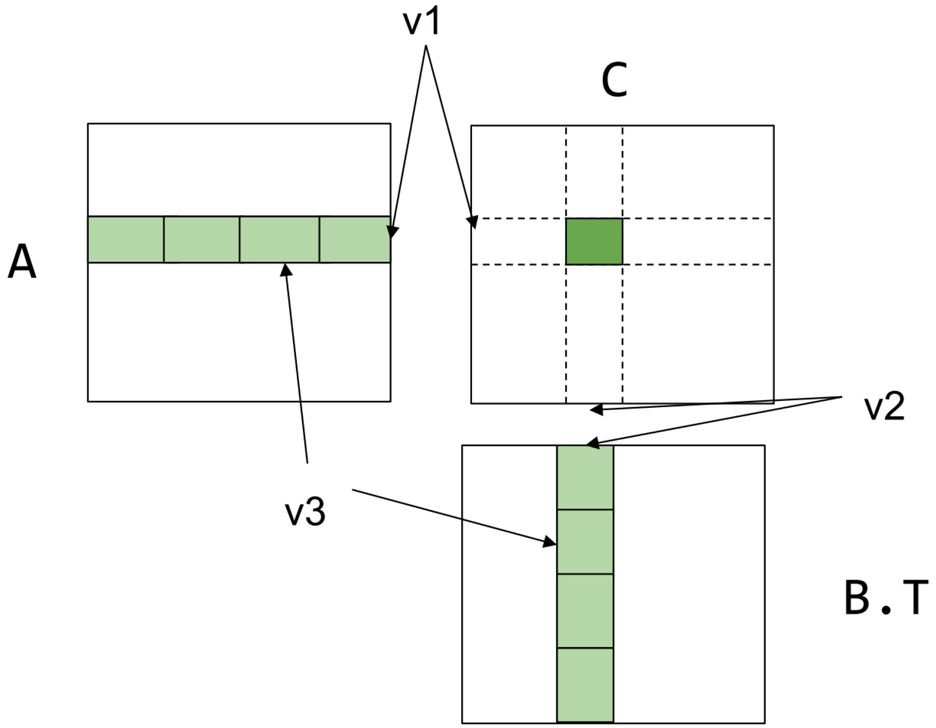
Write c n^3

#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;
   }
}
```



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^3 / v_3

#reigsters needed:
V1*v3 + v2*v3 + v1*v2

Read cost:
(n^3/v2 + n^3 / v1 + n^3 / v3) * speed(dram -> register)

Register Tiled Matrix Multiplication

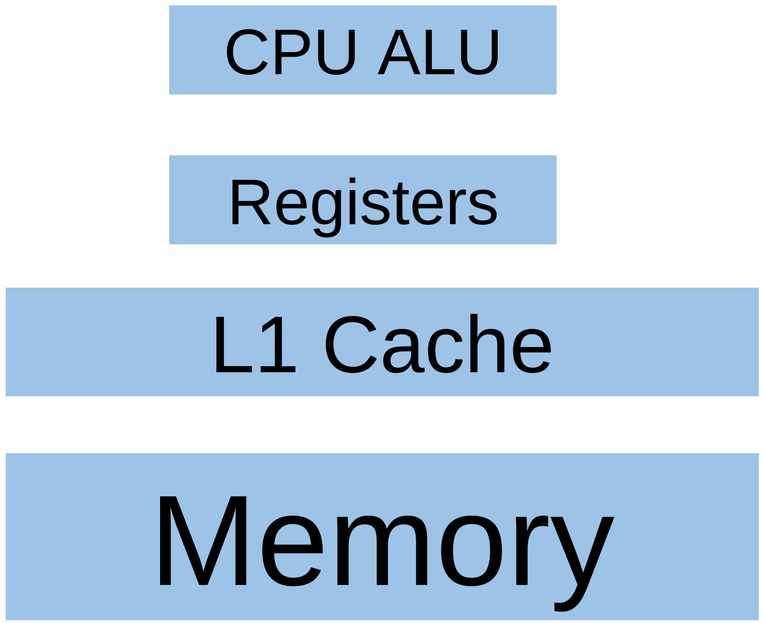
- Q: is the load cost related to v3?
- Q: How to set v1 / v2?
  - What are the constraints?
- Q: Why essentially can tiling reduce read cost?

Read a	N^3 / v_2
Read b	N^3 / v_1
Write c	N^3 / v_3

#reigsters needed:
V1*v3 + v2*v3 + v1*v2

Read cost:
(n^3/v2 + n^3 / v1 + n^3 / v3) * speed(dram -> register)

Make it more complicated: Consider L1 cache

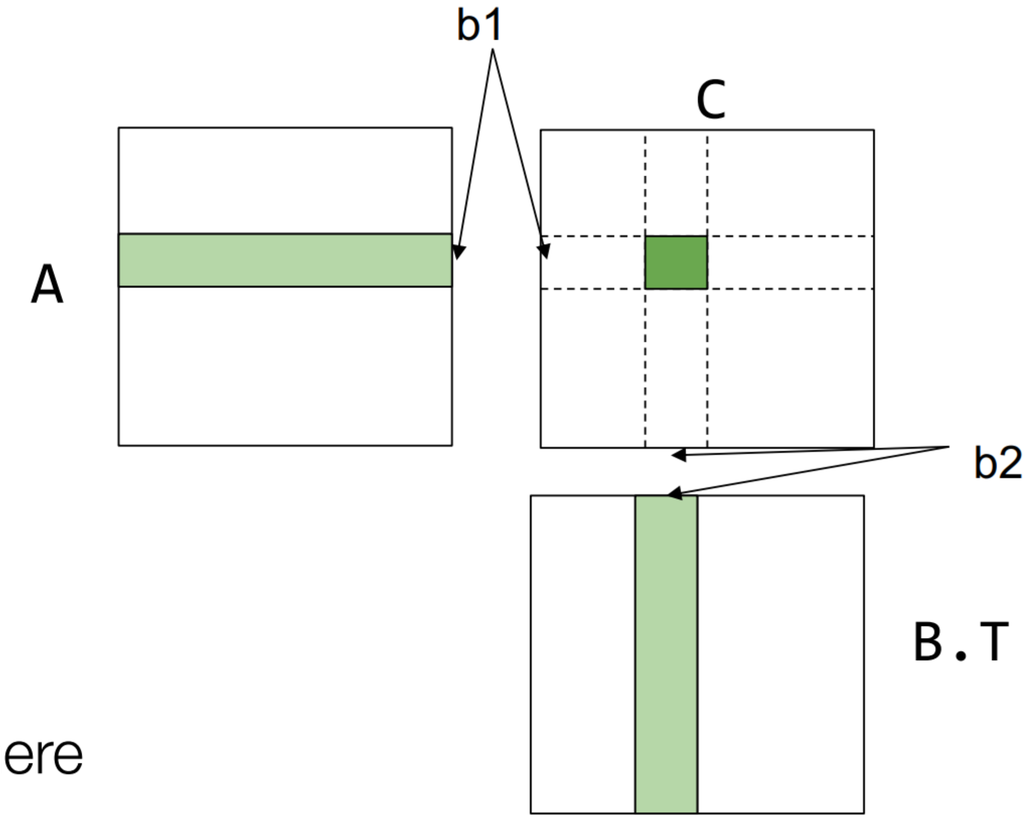


Cache-aware tiling

- We can further tile [b1][n] / [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);
   }
}
```

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 time cost:

n / b1 * n * b1 = n^2

B's dram -> l1 time cost:

n / b1 * n / b2 * b2 * n = n^2 / b1

Vs. previous untiled version?

s.t.

- b1 * n + b2 * n < L1 cache size
- b1 % v1 == 0
- b2 % v2 == 0

Putting Things Together

We set v3 = 1

```
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:

l1 -> cache:

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

dram -> l1

- n^2 + n^3 / b1

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
- On GPUs:
  - We have dram -> HBM -> sram -> L1 -> L1 -> register?

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

One of the Most Complicated Case

- 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.

Next 2 – 3 lectures

- Fundamentals: why/how we can make operator fast
- Case study: Matmul
- **GPU architecture and programming**
- Roofline model

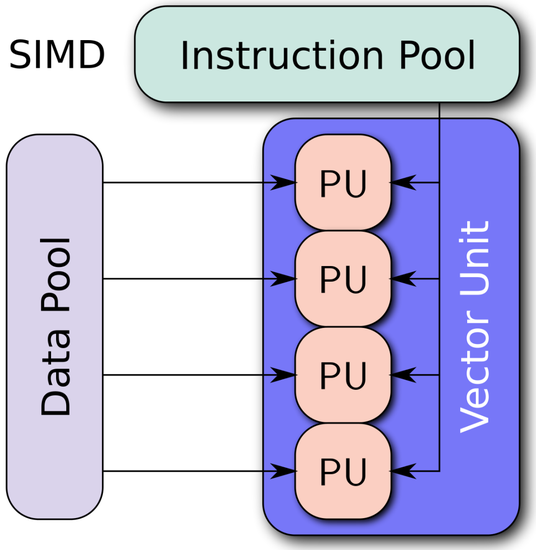
Recap CPU parallelization

- We can parallelize this loop using CPU threads
- We can parallelize this using many concurrent cores

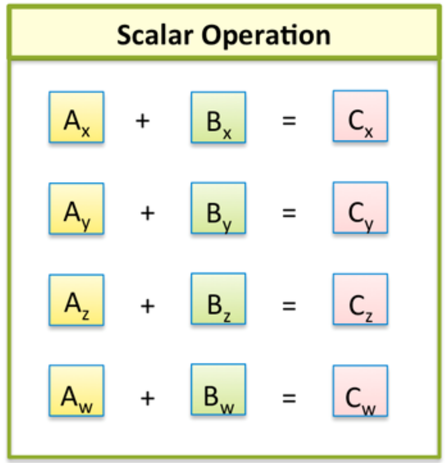
```
#pragma omp parallel for
for (int i = 0; i < 64; ++i) {
      float4 a = load_float4(A + i*4);
      float4 b = load_float4(B + i*4);
      float4 c = add_float4(a, b);
      store_float4(C * 4, c);
}
```

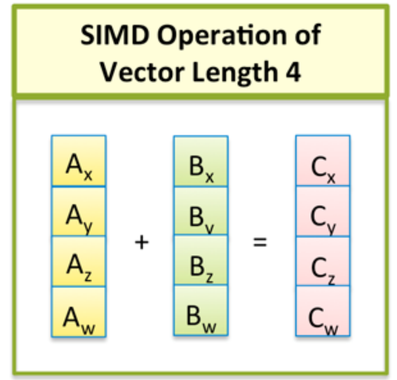
Vectorized & parallelized

Single-Instruction Multiple-Data



Example for SIMD in data science:

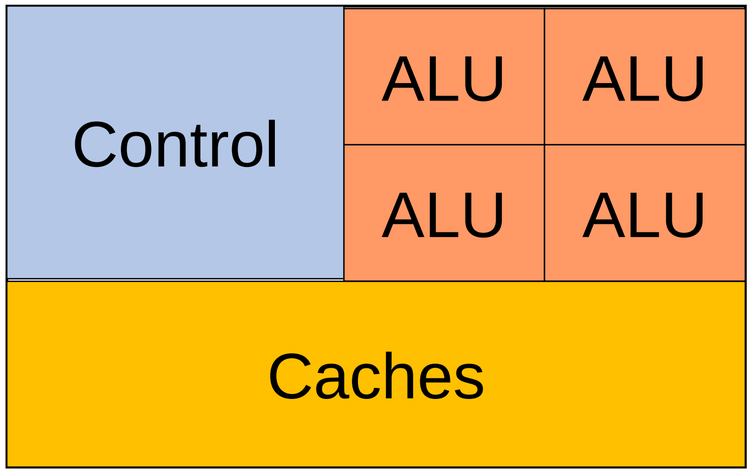


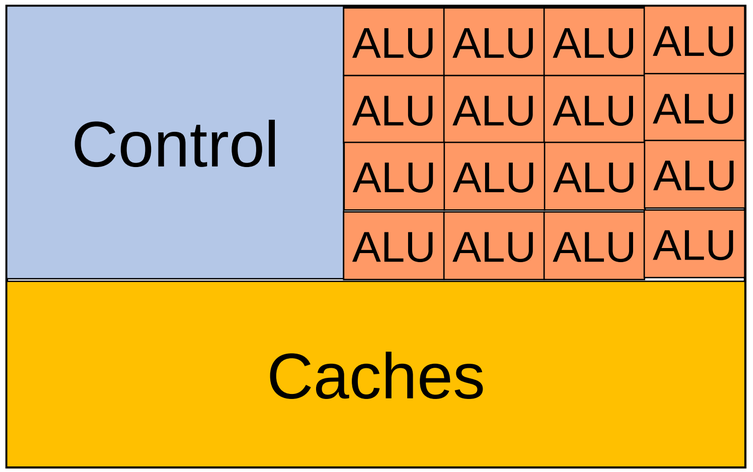


Intel® Architecture currently has SIMD operations of vector length 4, 8, 16

Chip Design Trajectory: SIMD

If we're able to reduce to size of ALU while keeping its power

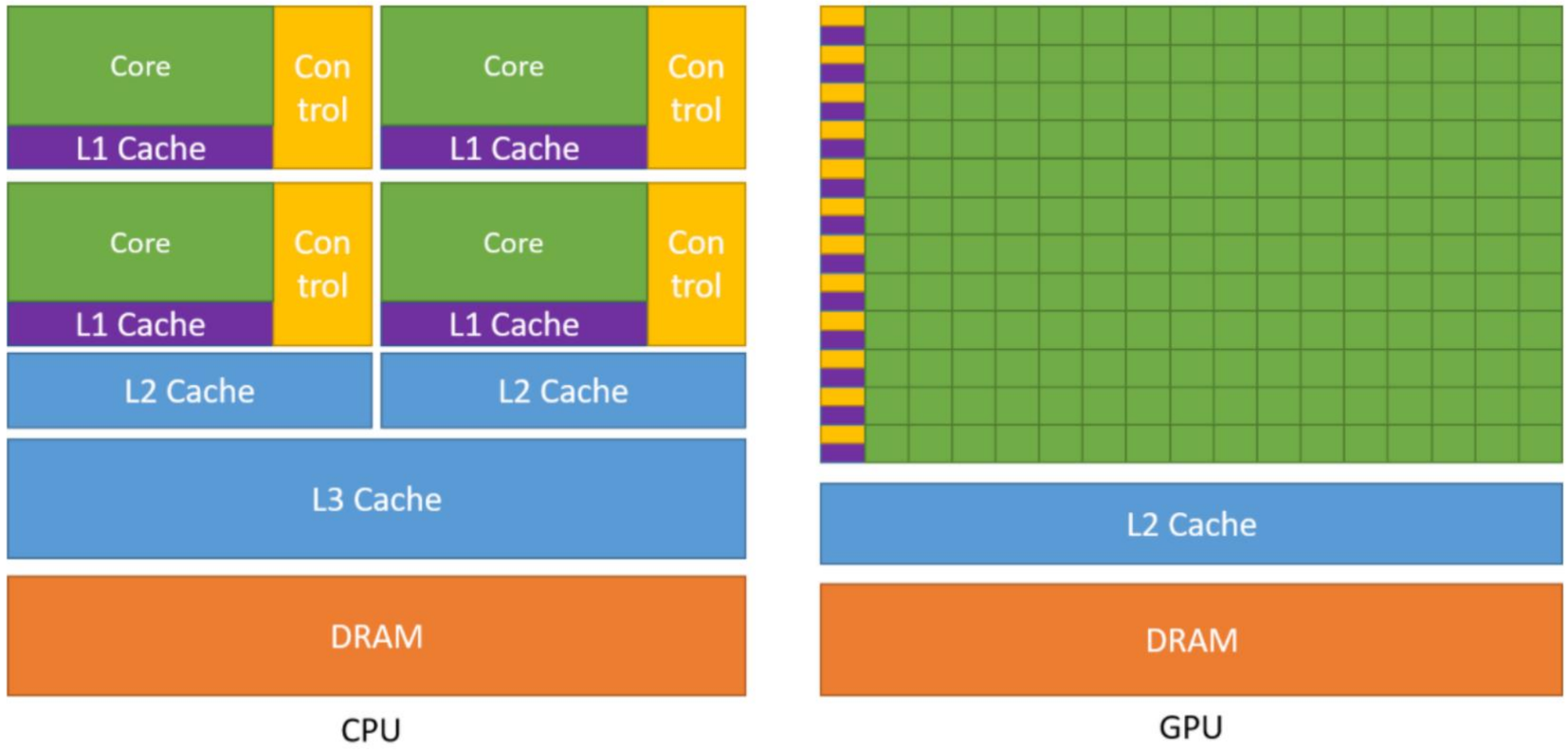




That's why you see trends: 70nm -> 60nm -> 50nm -> … -> what is the best now?

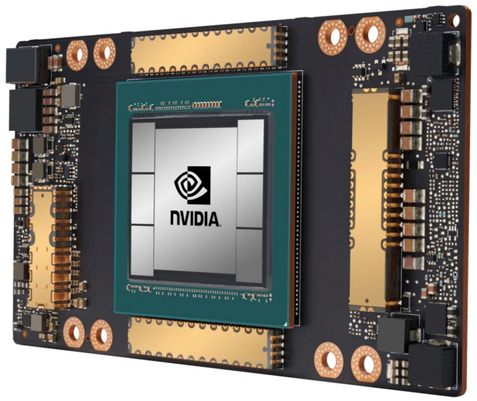
Problem: this is not substantiable; there are also power/heat issues when you put more ALUs in

Idea: How about we use a lot of weak/specialized cores

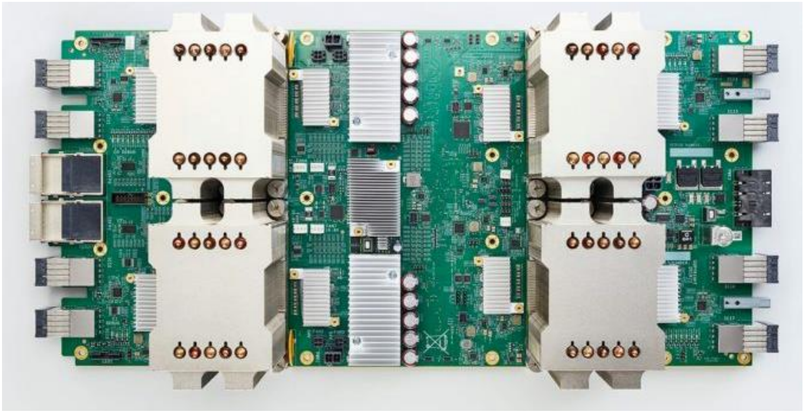


Hardware Accelerators: GPUs

- **Graphics Processing Unit (GPU)**: Tailored for matrix/tensor ops
- Basic idea: Use tons of ALUs (but weak and more specialized); massive data parallelism (SIMD on steroids); now H100 offers ~980 TFLOPS for FP16!
- Popularized by NVIDIA in early 2000s for video games, graphics, and multimedia; now ubiquitous in DL
- CUDA released in 2007; later wrapper APIs on top: CuDNN, CuSparse, CuDF (RapidsAI), NCCL, etc.



Other Hardware Accelerators

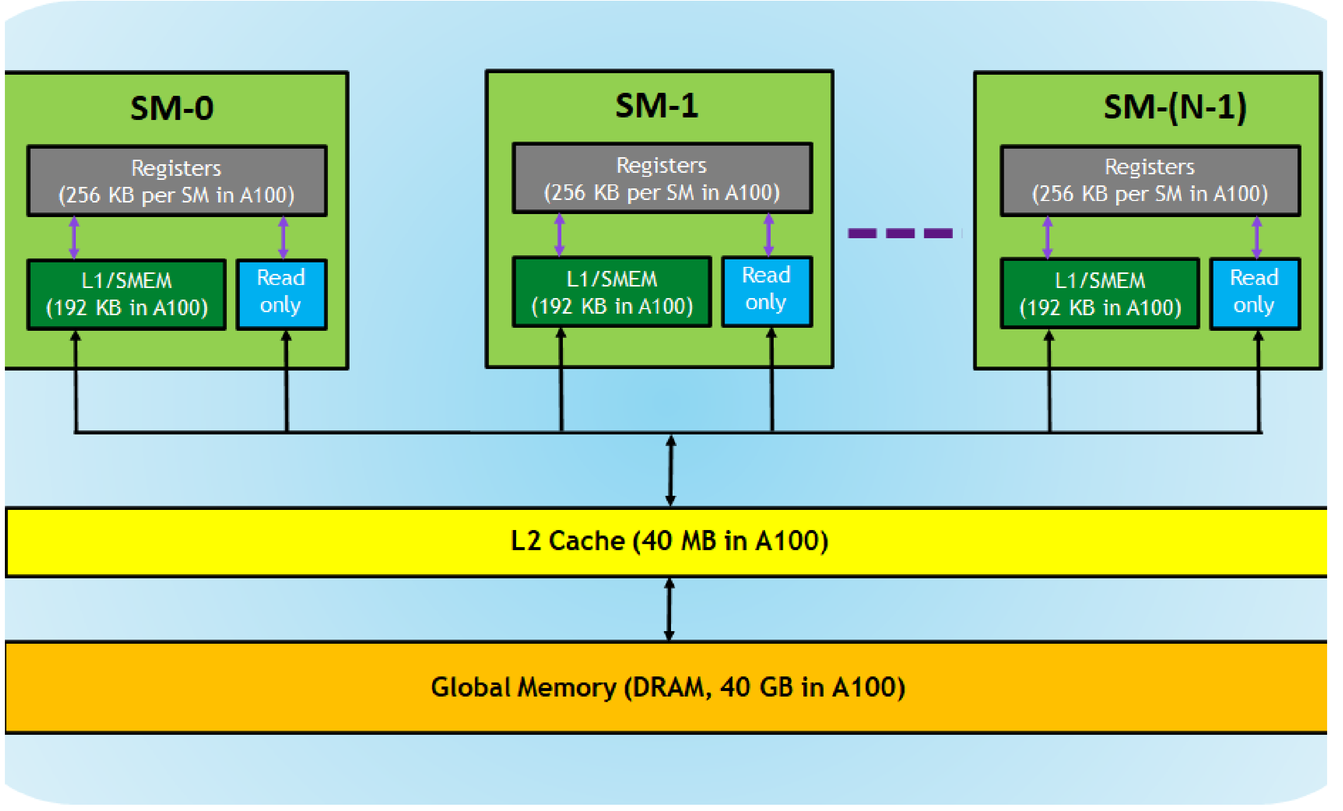


- General Trajectory:
  - Use more specialized core
  - Reduce precision
  - Mixing specialized and general-purpose cores
- E.g.
  - Tensor Processing Unit (TPU)
  - An “application-specific integrated circuit” (ASIC) created by Google in mid 2010s; used for AlphaGo
- E.g.
  - B200 (projected release 2025): fp4 / fp8 Tensorcore
- E.g.
  - M3 max: mixing tensorcore and normal core

GPU and CUDA

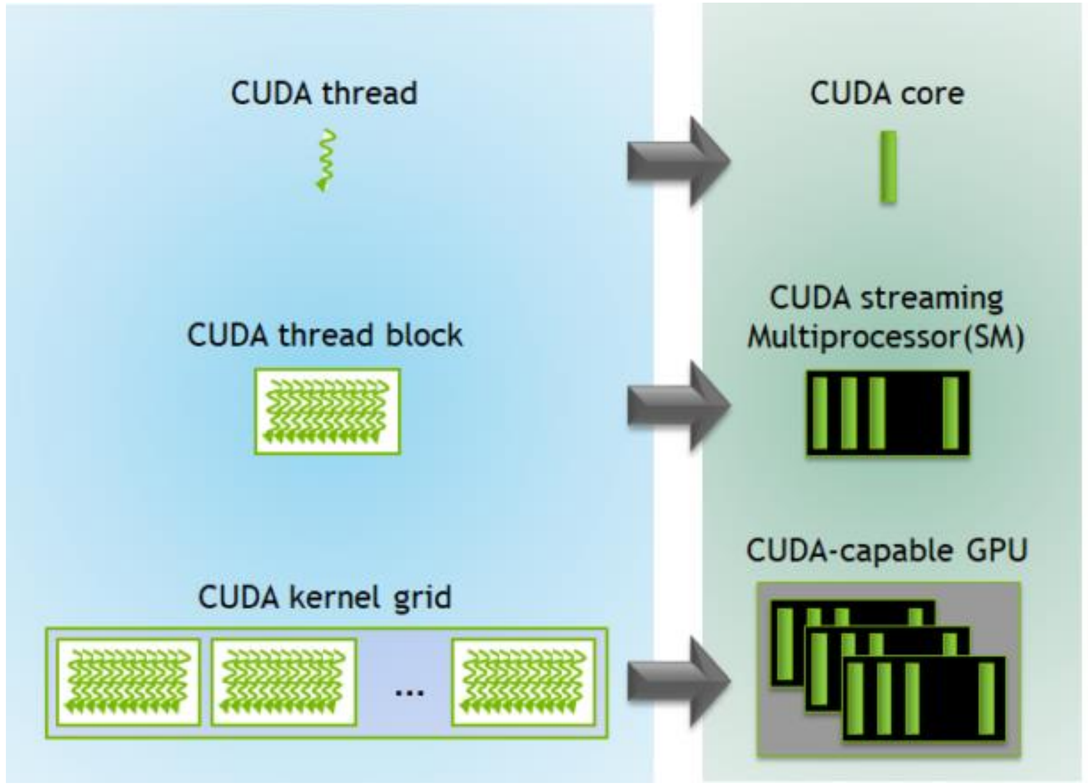
- Basic concepts and Architecture
- Programming abstraction
- Case study: Matmul

GPU Overview



Threads, Blocks, Grids

- Threads: smallest units to process a chunk of data
- Blocks: A group of threads that share memory
- Grid: A collection of blocks that execute the same kernel



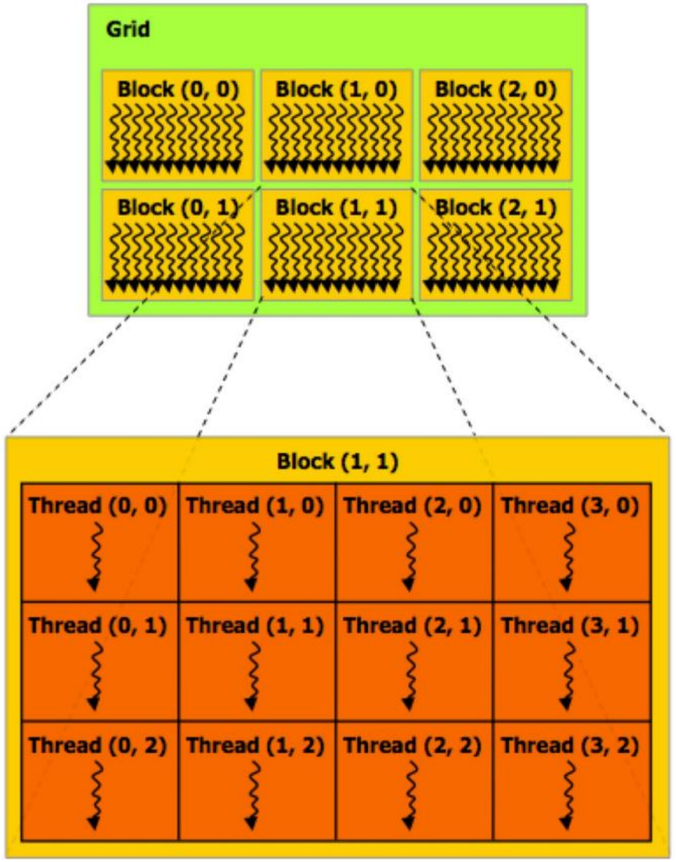
How many SMs/Threads we have?

- V100 (2018 - Now): 80 SMs, 2048 threads/SM
- A100 (2020 - Now): 108 SMs, 2048 threads/SM
- H100 (2022 - Now): 144 SMs, 2048 threads/SM
- B100 (2025 -): go surveying the numbers

CUDA

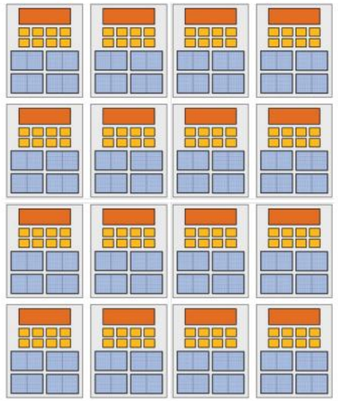
- Introduced in 2007 with NVIDIA Tesla architecture
- C-like languages for programming GPUs
- CUDA's design matches the grid/block/thread concepts in GPUs

CUDA Programs contain A Hierarchy of Threads



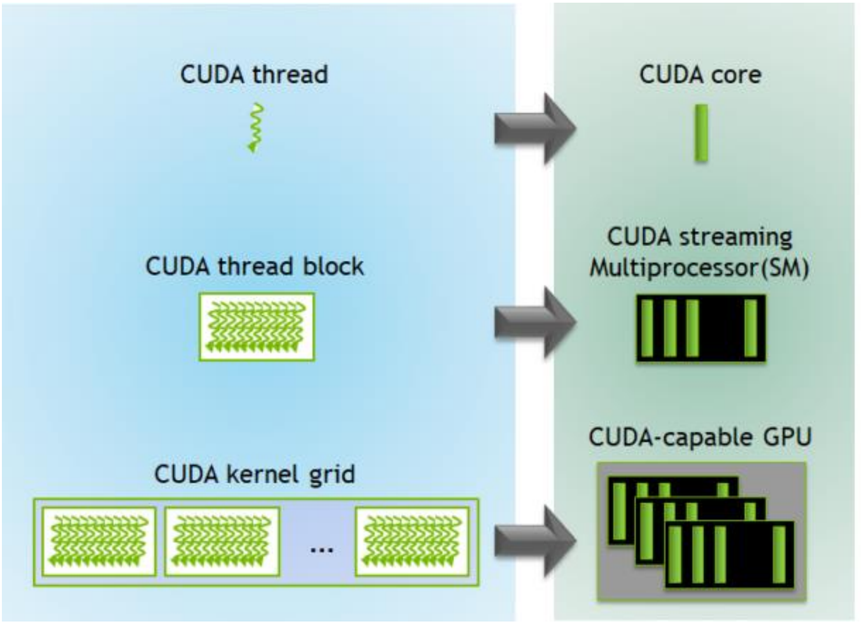

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Run on CPU



How Many threads/Blocks it runs on?

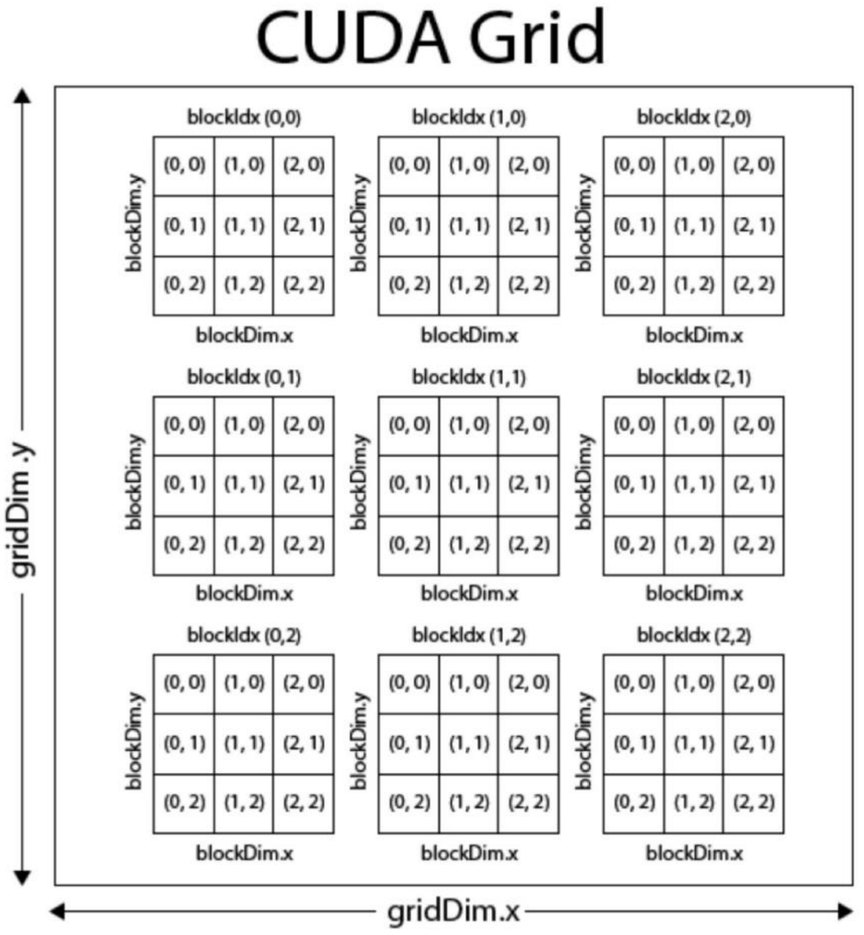
```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```



Grid, Block, and Thread

- GridDim: The dimensions of the grid
- blockIdx: The block index within the grid
- blockDim: The dimensions of a block
- threadIdx: The thread index within a block

- What About GridId?
- What about threadDim?



An Example CUDA Program

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will cause execution of 72 threads
// 6 blocks of 12 threads each
matrixAddDoubleB<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

- "launch a grid of CUDA thread
- blocks" Call returns when all threads have terminated

```
__device__ float doubleValue(float x)
{
   return 2 * x;
}

// kernel definition
__global__ void matrixAddDoubleB(float A[Ny][Nx],
                                 float B[Ny][Nx],
                                 float C[Ny][Nx])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;

   C[j][i] = A[j][i] + doubleValue(B[j][i]);
}
```

- __global__ denotes a CUDA kernel function runs on GPU
- Each thread indexes its data using blockIdx, blockDim, threadIdx and execute the compute

Separation CPU and GPU Execution

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will cause execution of 72 threads
// 6 blocks of 12 threads each
matrixAddDoubleB<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

```
__device__ float doubleValue(float x)
{
   return 2 * x;
}

// kernel definition
__global__ void matrixAddDoubleB(float A[Ny][Nx],
                                 float B[Ny][Nx],
                                 float C[Ny][Nx])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;

   C[j][i] = A[j][i] + doubleValue(B[j][i]);
}
```

- Host code: serial execution on CPU
- Device code: SIMD parallel execution on GPUs

#Threads is Explicit and Static in Programs

```
const int Nx = 11; // not a multiple of threadsPerBlk.x
const int Ny = 5; // not a multiple of threadsPerBlk.y

dim3 threadsPerBlk(4, 3, 1);
dim3 numBlocks(3, 2, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlk>>>(A, B, C);
```

```
// kernel definition
__global__ void matrixAdd(float A[Ny][Nx],
                          float B[Ny][Nx],
                          float C[Ny][Nx])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;
   // guard against out of bounds array access
   if (i < Nx && j < Ny)
      C[j][i] = A[j][i] + B[j][i];
}
```

Developers to:

- To provide CPU/GPU code separation
- Statically declare blockDim, shapes.
- Map data to blocks/threads
- Check boundary conditions

SIMD Constraints: how to handle control flow?

SIMD requires all ALUs/Core Must proceed in the same pace

```
// kernel definition
__global__ void f(float A[N])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   float x = A[i];
   if (x > 0) {
      x = 2.0f * x;
   } else {
      x = exp(x, 5.0f);
   }
   A[i] = x;
}
```