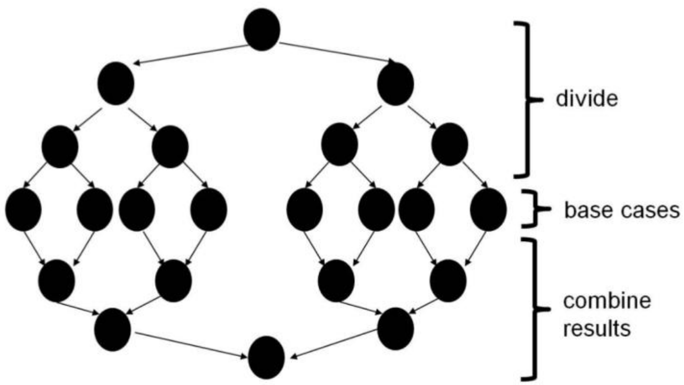
CSE 332: Data Structures & Parallelism
Lecture 26: More Parallelism
(just for fun)

UNIVERSITY of WASHINGTON

332 Recap: Fork-Join Pattern

> Split in 2+ subproblems
> Solve recursively in parallel
> Join and reduce together




```java
class SumTask extends RecursiveTask<Integer> {
  ...
  SumTask(int[] arr, int start, int end) { … }

  int computeSequential() { … }

  protected int compute() {
    if (end - start < THRESHOLD)
      return computeSequential();

    int mid = start + (end - start) / 2;
    SumTask left  = new SumTask(in, out, start, mid);
    SumTask right = new SumTask(in, out, mid, end);

    left.fork();
    int rightResult = right.compute();
    int leftResult =  left.join();

    return leftResult + rightResult;
  }
}
```

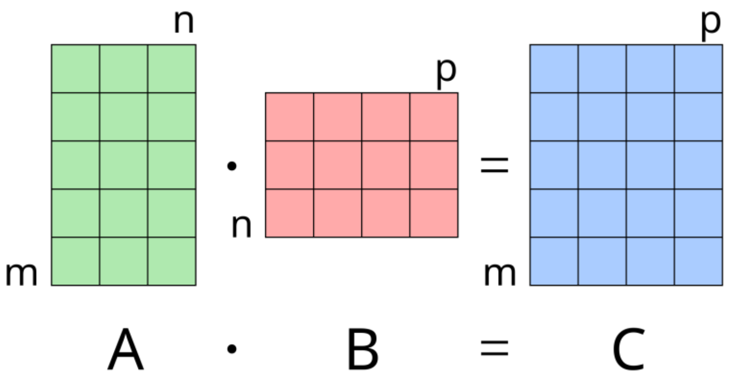
Can we do better?

Parallelism Beyond Fork-Join

- In 332, only tested on Fork-Join
- But there's so much more
- Today is just for fun!
- *Please don't write code like this on your final*

Linear algebra problems:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$



?

Convolutions

- 1d or multidimensional
- Signal Processing
  - Cleaning audio streams
  - Image filters (blur, sharpen)
  - CSE 455
- Machine Learning
  - Convolutional Neural Networks
  - Object detection in images
  - CSE 446

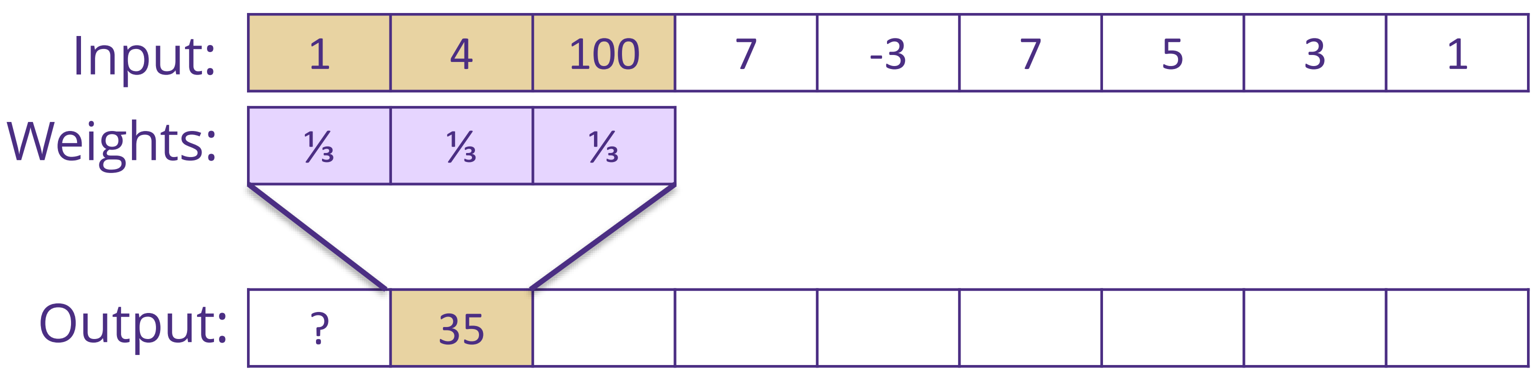
Convolutions

- Slide a window over an array
- Multiply each windowed value by a weight
- Sum and put the result in an output array
- Each output value is a weighted sum of nearby inputs
- Many ways to deal with edges

Input:	1	4	100	7	-3	7	5	3	1
Weights:	⅓	⅓	⅓						
Output:									

Convolutions

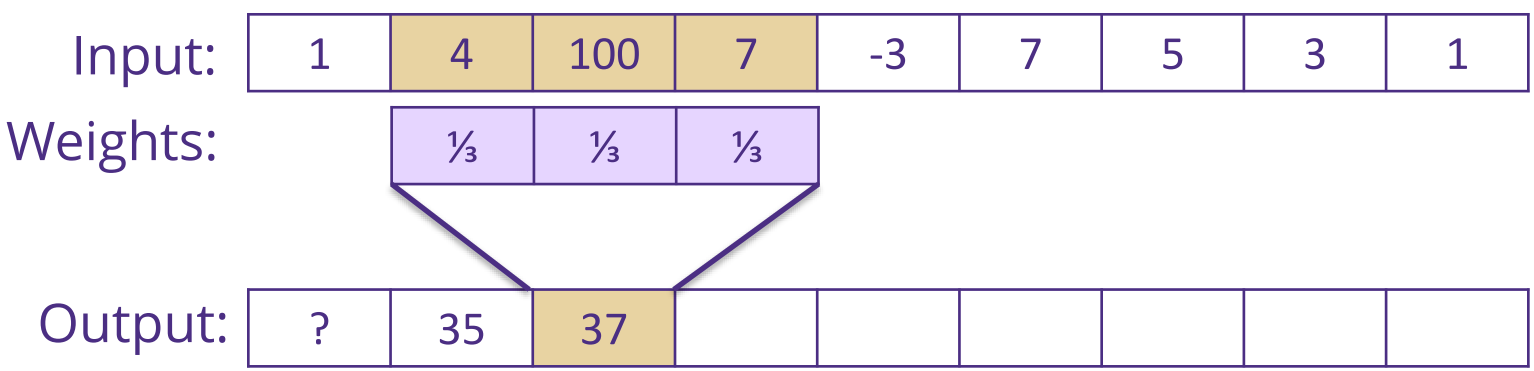
- Slide a window over an array
- Multiply each windowed value by a weight
- Sum and put the result in an output array
- Each output value is a weighted sum of nearby inputs.
- Many ways to deal with edges



$$1 \cdot \frac{1}{3} + 4 \cdot \frac{1}{3} + 100 \cdot \frac{1}{3} = 35$$

Convolutions

- Slide a window over an array
- Multiply each windowed value by a weight
- Sum and put the result in an output array
- Each output value is a weighted sum of nearby inputs.
- Many ways to deal with edges



$$4 \cdot \frac{1}{3} + 100 \cdot \frac{1}{3} + 6 \cdot \frac{1}{3} = 37$$

Convolutions

- Slide a window over an array
- Multiply each windowed value by a weight
- Sum and put the result in an output array
- Each output value is a weighted sum of nearby inputs.
- Many ways to deal with edges

Input:	1	4	100	7	-3	7	5	3	1
Weights:			⅓	⅓	⅓				
Output:	?	35	37	34.7					

$$100 \cdot \frac{1}{3} + 7 \cdot \frac{1}{3} + (-3) \cdot \frac{1}{3} = 34.\overline{66}$$

Convolutions

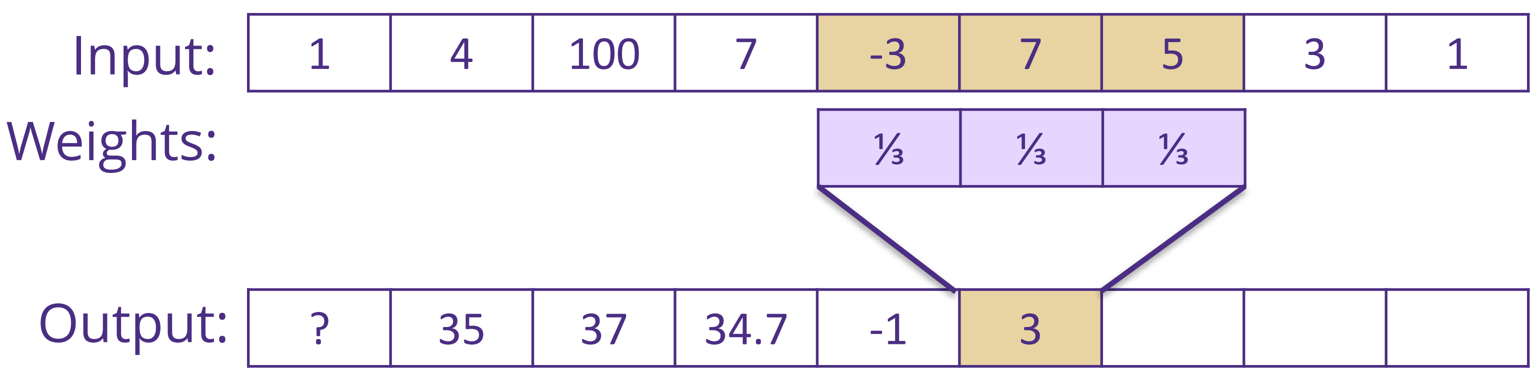
- Slide a window over an array
- Multiply each windowed value by a weight
- Sum and put the result in an output array
- Each output value is a weighted sum of nearby inputs.
- Many ways to deal with edges

Input:	1	4	100	7	-3	7	5	3	1
Weights:				⅓	⅓	⅓			
Output:	?	35	37	34.7	-1				

$$7 \cdot \frac{1}{3} + (-3) \cdot \frac{1}{3} + 7 \cdot \frac{1}{3} = -1$$

Convolutions

- Slide a window over an array
- Multiply each windowed value by a weight
- Sum and put the result in an output array
- Each output value is a weighted sum of nearby inputs.
- Many ways to deal with edges



$$(-3) \cdot \frac{1}{3} + 7 \cdot \frac{1}{3} + 5 \cdot \frac{1}{3} = 3$$

Convolutions

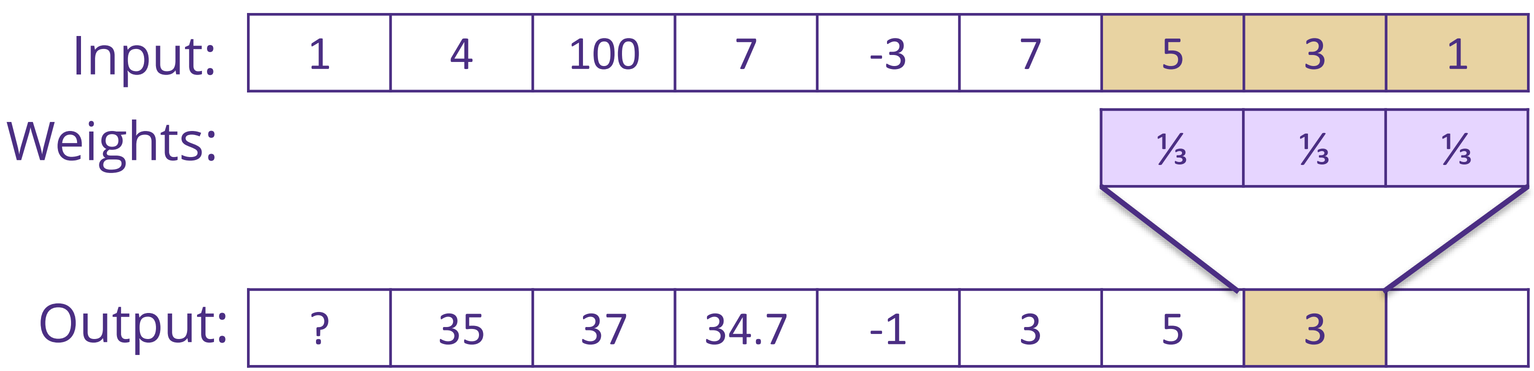
- Slide a window over an array
- Multiply each windowed value by a weight
- Sum and put the result in an output array
- Each output value is a weighted sum of nearby inputs.
- Many ways to deal with edges

Input:	1	4	100	7	-3	7	5	3	1
Weights:						⅓	⅓	⅓	
Output:	?	35	37	34.7	-1	3	5		

$$7 \cdot \frac{1}{3} + 5 \cdot \frac{1}{3} + 3 \cdot \frac{1}{3} = 5$$

Convolutions

- Slide a window over an array
- Multiply each windowed value by a weight
- Sum and put the result in an output array
- Each output value is a weighted sum of nearby inputs.
- Many ways to deal with edges



$$5 \cdot \frac{1}{3} + 3 \cdot \frac{1}{3} + 1 \cdot \frac{1}{3} = 3$$

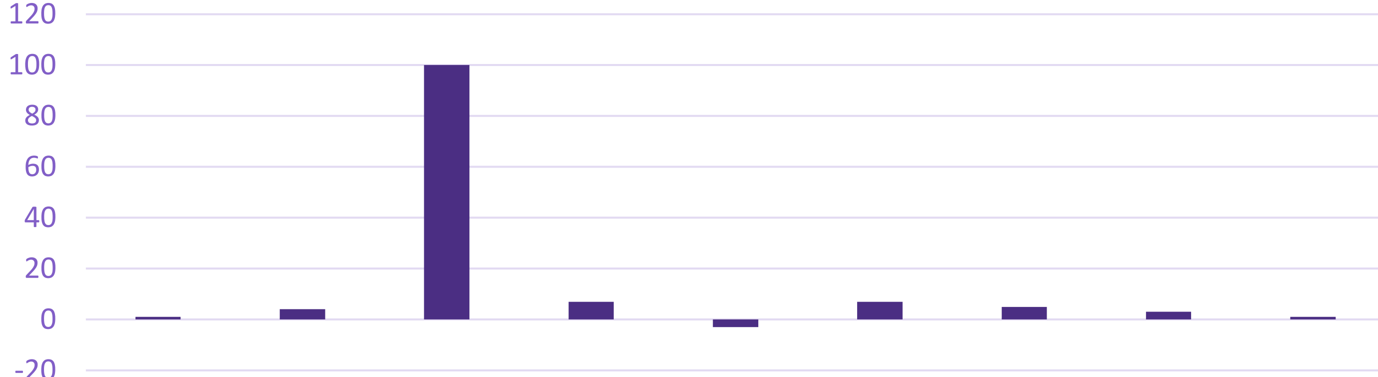
Convolutions

- Slide a window over an array
- Multiply each windowed value by a weight
- Sum and put the result in an output array
- Each output value is a weighted sum of nearby inputs.
- Many ways to deal with edges

Input:	1	4	100	7	-3	7	5	3	1
Weights:	⅓	⅓	⅓				⅓	⅓	⅓
Output:	35	35	37	34.7	-1	3	5	3	3

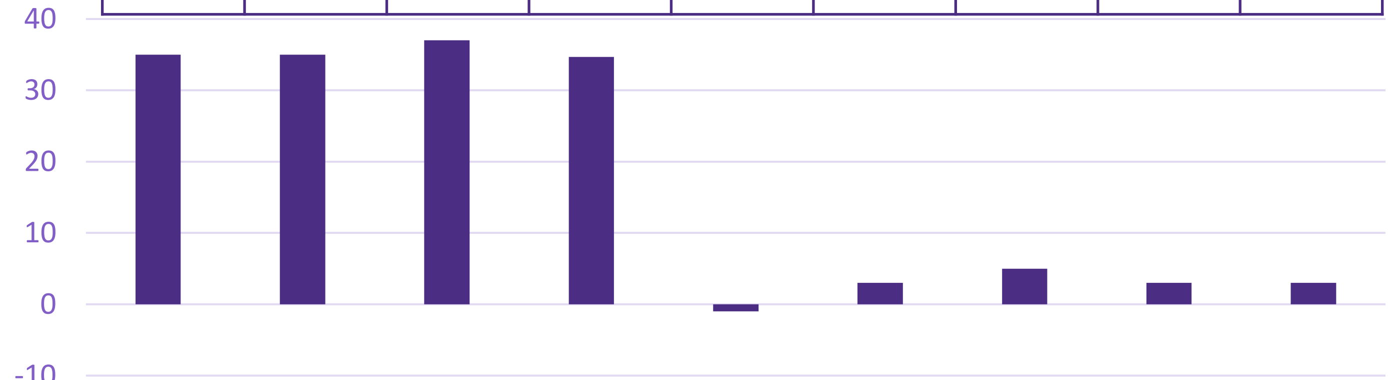
One way to handle edges

Smoothed!



Before:

1	4	100	7	-3	7	5	3	1



After:

35	35	37	34.7	-1	3	5	3	3

Code (Initial Attempt)

```java
public static void smooth(float[] in, float[] out) {
    int n = in.length;

    // Left boundary
    out[0] = (in[0] + in[1] + in[2]) / 3f;

    // Main loop
    for (int i = 1; i < n - 1; i++) {
        out[i] = (in[i - 1] + in[i] + in[i + 1]) / 3f;
    }

    // Right boundary
    out[n - 1] = (in[n - 3] + in[n - 2] + in[n - 1]) / 3f;
}
```

DEMO

Fork-Join

```
    public static void smooth(float[] in, float[] out) {
        int n = in.length;

        // Left boundary
        out[0] = (in[0] + in[1] + in[2]) / 3f;

        // Launch ForkJoin task for [1 .. n-2]
        POOL.invoke(new SmoothTask(in, out, 1, n - 1));

        // Right boundary
        out[n - 1] = (in[n - 3] + in[n - 2] + in[n - 1]) / 3f;
    }
...
```

Fork-Join continued

```
...
private static class SmoothTask extends RecursiveAction {
...
    public SmoothTask(float[] in, float[] out, int start, int end) { ... }

    protected void compute() {
        if (end - start < THRESHOLD) {
            computeSequential();
        } else {
            int mid = start + (end - start) / 2;

            SmoothTask left  = new SmoothTask(in, out, start, mid);
            SmoothTask right = new SmoothTask(in, out, mid, end);

            left.fork();
            right.compute();
            left.join();
        }
    }

    private void computeSequential() {
        for (int i = start; i < end; i++) {
            out[i] = (in[i - 1] + in[i] + in[i + 1]) / 3f;
        }
    }
}
```

DEMO

Parallel Streams

- JDK-8 introduced Parallel Streams
- Easy syntax for map and reduce
- Still Fork-Join under the hood

```
public static void smooth(float[] in, float[] out) {
    int n = in.length;

    // Left boundary
    out[0] = (in[0] + in[1] + in[2]) / 3f;

    // Parallel map for the main body: indices 1 .. n−2
    IntStream.range(1, n − 1)
            .parallel()
            .forEach(i → out[i] = (in[i − 1] + in[i] + in[i + 1]) / 3f);

    // Right boundary
    out[n − 1] = (in[n − 3] + in[n − 2] + in[n − 1]) / 3f;
} // PLEASE DO NOT WRITE CODE LIKE THIS ON THE FINAL
```

DEMO

Too many threads

- Do we need so many threads?
  - Perhaps not
- Still calculating one value at a time, per thread
  - What if we could do multiple in one operation?

VECTOR!

~~Committing crimes~~ "A quantity represented by an arrow, with both direction and magnitude"

Could be thought of as "a couple of numbers"
[1,3,4,-1]

Single Instruction, Multiple Data (SIMD)

- Operate on vectors, not "scalars"
  - $[a_1, a_2, a_3, a_4] + [b_1, b_2, b_3, b_4] = [a_1 + b_1, a_2 + b_2, a_3 + b_3, a_4 + b_4]$
  - Works elementwise
  - Parallelism *without* threads
- Requires a fixed vector size
  - 128, 256, and 512 bit vectors are common today
  - Corresponds to 4, 8, and 16, 32 bit integers
  - Or perhaps 2, 4, or 8, 64 bit floats

double #1	double #2	double #3	double #4
vector256			

Single Instruction, Multiple Data (SIMD)

- Does not help asymptotically
  - Only divides number of operations by a constant factor
- Extremely common today
  - Extensions available in consumer CPUs since the ‘90s
  - SSE and AVX on x86; SVE and Neon on ARM
- Not all problems benefit
  - Can worsen performance in many circumstances

Single Instruction, Multiple Data (SIMD)

- Vectors are normally loaded from arrays
  - Must be contiguous memory
- For *n* elements, vector size *V*:
  - $\left\lfloor \frac{n}{V} \right\rfloor$ vector loops iterations
  - $n \% V$ scalar loop iterations

- Cannot "branch" on each element
  - Can branch based on entire vector

Vector Code

```java
private static final VectorSpecies<Float> S = FloatVector.SPECIES_PREFERRED;

public static void smooth(float[] in, float[] out) {
    int n = in.length;

    out[0] = (in[0] + in[1] + in[2]) / 3f;

    int i = 1;
    int upper = n - 1 - (S.length() - 1);  // leave space for right boundary

    for (; i < upper; i += S.length()) {
        FloatVector left  = FloatVector.fromArray(S, in, i - 1);
        FloatVector mid   = FloatVector.fromArray(S, in, i);
        FloatVector right = FloatVector.fromArray(S, in, i + 1);

        FloatVector sum = left.add(mid).add(right);
        FloatVector smoothed = sum.div(3f);

        smoothed.intoArray(out, i);
    }

    // Scalar loop for remaining elements
    for (; i < n - 1; i++)
        out[i] = (in[i - 1] + in[i] + in[i + 1]) / 3f;

    out[n - 1] = (in[n - 3] + in[n - 2] + in[n - 1]) / 3f;
} // PLEASE DO NOT WRITE CODE LIKE THIS ON THE FINAL
```

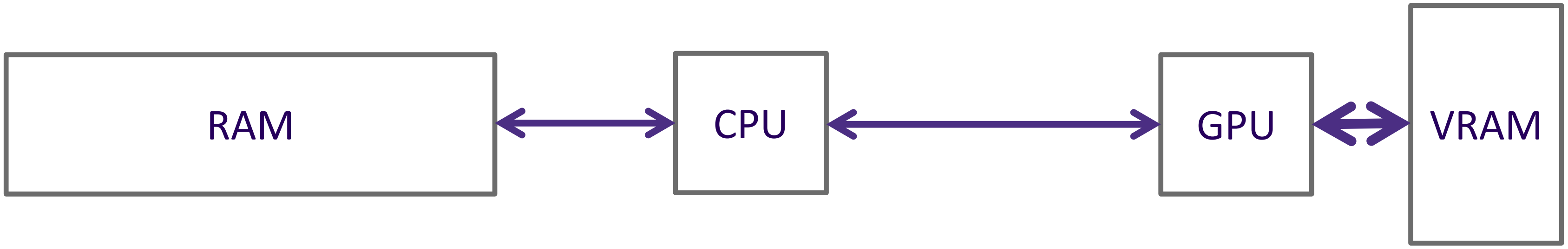
DEMO

Practical limits

- 500,000,000 computations
  - Load 3 floats
  - Sum
  - Write 1 float
  - 4 floats each computation
  - 4 bytes per float
  - 8 GB processed
- ~57 GB/second RAM throughput
  - Demo RAM speed: 2 channels @ 3600 MT/s
- 8 GB / 57 GB/second= 140 milliseconds
  - Hard lower bound

Beyond CPU

- GPUs perform math in parallel
  - A LOT of math
  - Thousands of threads at once
- Extremely fast memory
  - Hundreds of gigabytes per second
- Expensive to move data from RAM to VRAM



Writing Code for the GPU

> CPU threads can spawn thousands of GPU threads
  - GPU threads CANNOT fork more threads
  - Single Instruction Multiple Threads (SIMT)
> Each thread executes the same instruction
  - Few operations are available
  - Branching is costly
> Few options for languages
  - CUDA (NVIDIA)
  - ROCm/HIP (AMD)

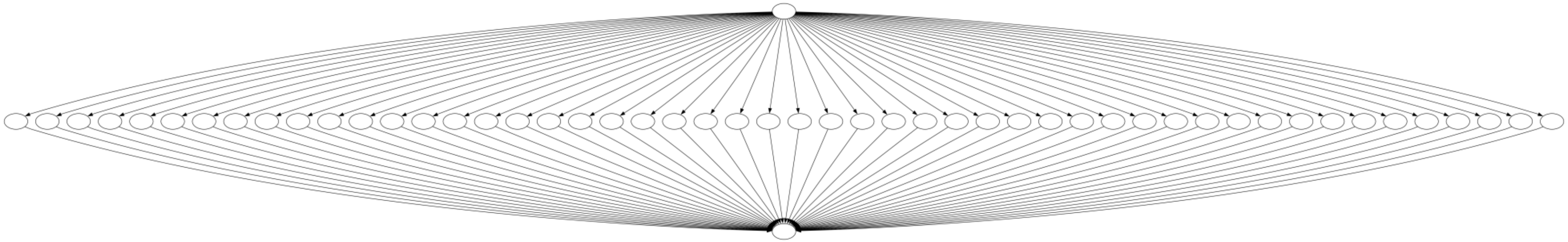
GPU Code (CUDA)

```
static double smooth(float* host_in, float* host_out, int n) {
    float *device_in, *device_out;
    cudaMalloc(&device_in, SIZE * sizeof(float));
    cudaMalloc(&device_out, SIZE * sizeof(float));
    // Copy input to GPU
    cudaMemcpy(device_in, host_in, SIZE * sizeof(float), cudaMemcpyHostToDevice);
    cudaDeviceSynchronize();

    // “Fork”
    int blockSize = 256;
    int gridSize = ((n - 2) + blockSize - 1) / blockSize;

    smoothKernel<<<gridSize, blockSize>>>(device_in, device_out, n);

    // “Join”
    cudaDeviceSynchronize();

    // Copy GPU result back to main memory
    cudaMemcpy(host_out, device_out, SIZE * sizeof(float), cudaMemcpyDeviceToHost);

    // Handle boundaries on CPU
    host_out[0] = (host_in[0] + host_in[1] + host_in[2]) / 3.0f;
    host_out[SIZE - 1] = (host_in[SIZE - 3] + host_in[SIZE - 2] + host_in[SIZE - 1]) / 3.0f;

    // Cleanup GPU memory allocations
    cudaFree(device_in);
    cudaFree(device_out);
    return kernel_elapsedMS;
}
```

GPU Code (CUDA)

```
__global__ void smoothKernel(const float* __restrict__ in,
                             float* __restrict__ out,
                             int n) {
    // The thread knows this implicitly.
    int i = blockIdx.x * blockDim.x + threadIdx.x + 1;

    // Main case
    out[i] = (in[i - 1] + in[i] + in[i + 1]) / 3.0f;
}
```

Analyzing GPU Parallelism



DEMO

When to use?

- **Fork-Join**
  - **CSE 332 Exams**
  - **Advanced parallel algorithms**
- **Parallel Streams**
  - **Easy map/reduce parallelization**
- **SIMD** (CPU Vector)
  - **Accelerating core library functions**
- **GPU**
  - **Training/Running Machine Learning models**
  - **3d rendering**