DATA LEVEL PARALLELISM

Mahdi Nazm Bojnordi

Assistant Professor

School of Computing

University of Utah

Overview

- Announcement
  - Homework 5: due on Nov. 20th
- This lecture
  - Data level parallelism

Overview

- ILP: instruction level parallelism
  - Out of order execution (all in hardware)
  - IPC hardly achieves more than 2
- Other forms of parallelism
  - DLP: data level parallelism
    - Vector processors, SIMD, and GPUs
  - TLP: thread level parallelism
    - Multiprocessors, and hardware multithreading
  - RLP: request level parallelism
    - Datacenters

Data Level Parallelism (DLP)

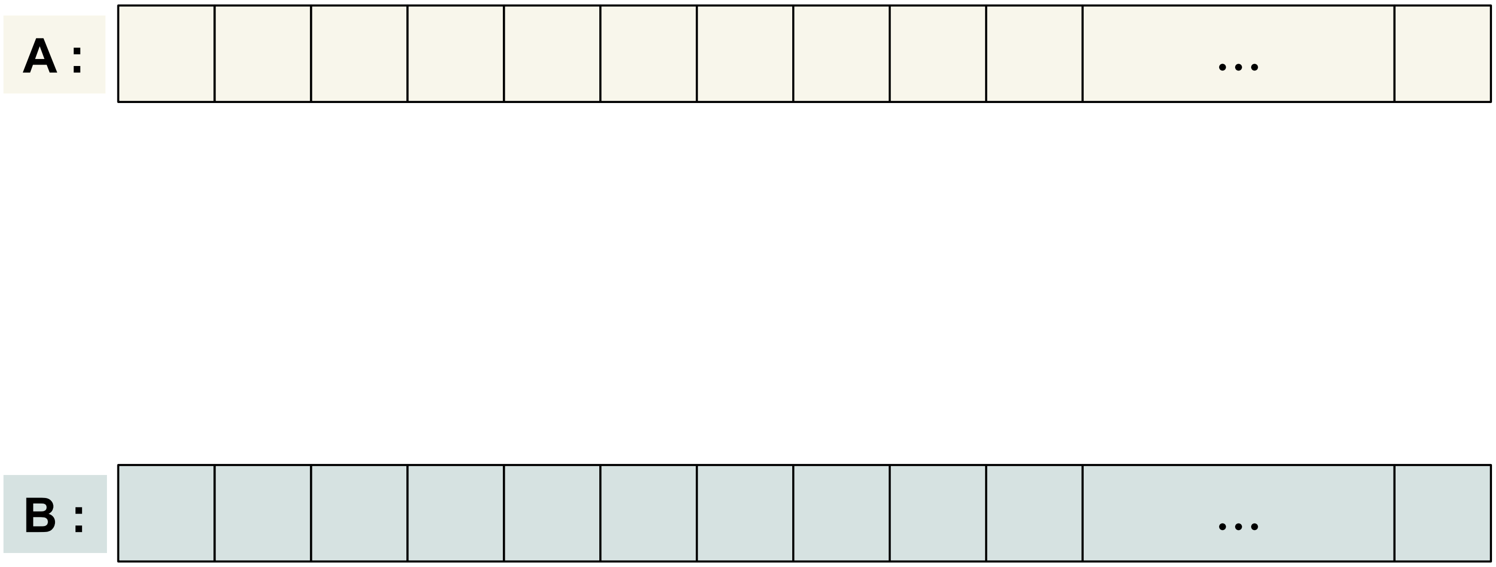
Data Level Parallelism

- Due to executing the same code on a large number of objects
  - Common in scientific computing
- DLP architectures
  - Vector processors—e.g., Cray machines
  - SIMD extensions—e.g., Intel MMX
  - Graphics processing unit—e.g., NVIDIA
- Improve throughput rather than latency
  - Not good for non-parallel workloads

Vector Processing

- Scalar vs. vector processor

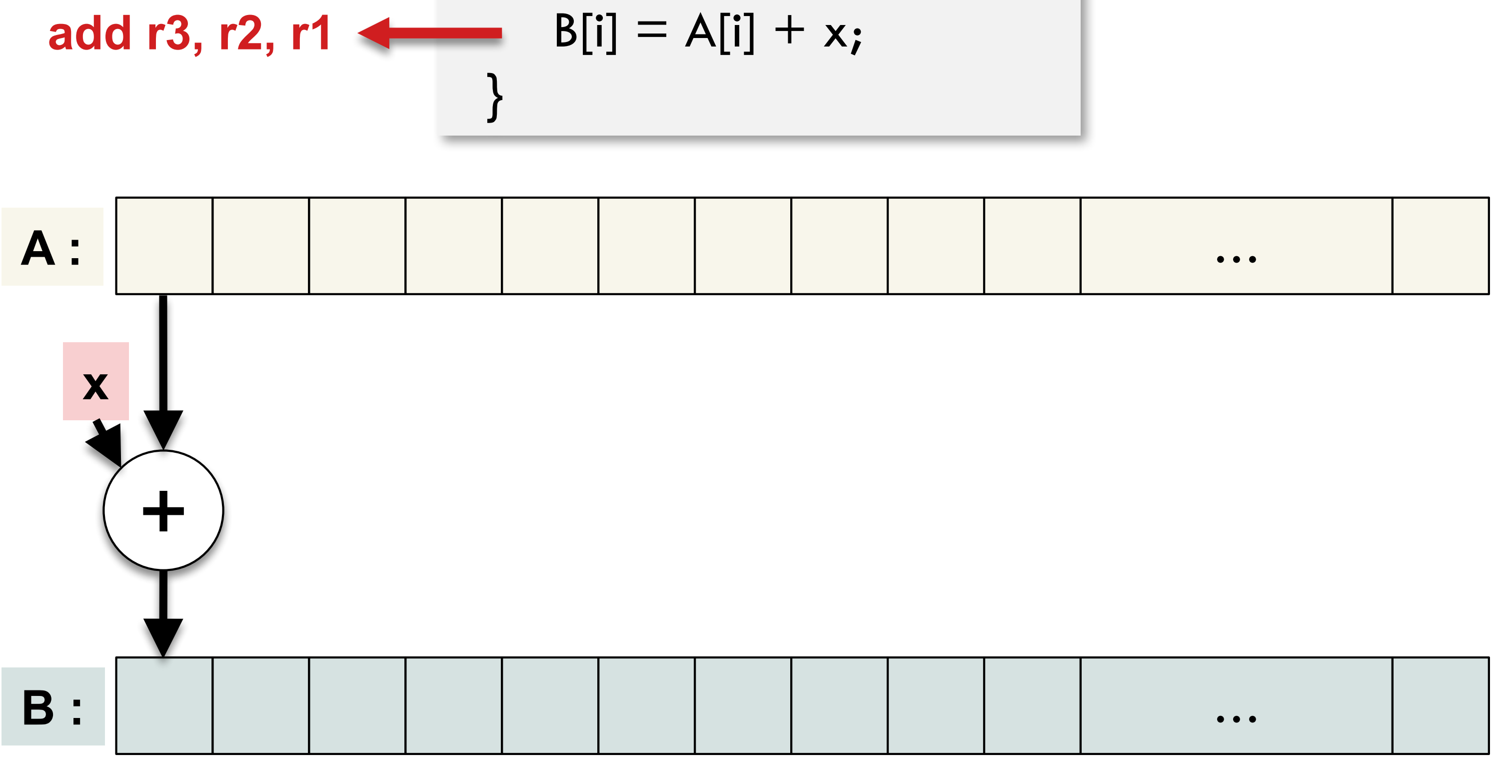
```
for(i=0; i<1000; ++i) {
    B[i] = A[i] + x;
}
```



Vector Processing

- Scalar vs. vector processor

```
for(i=0; i<1000; ++i) {
    B[i] = A[i] + x;
}
```

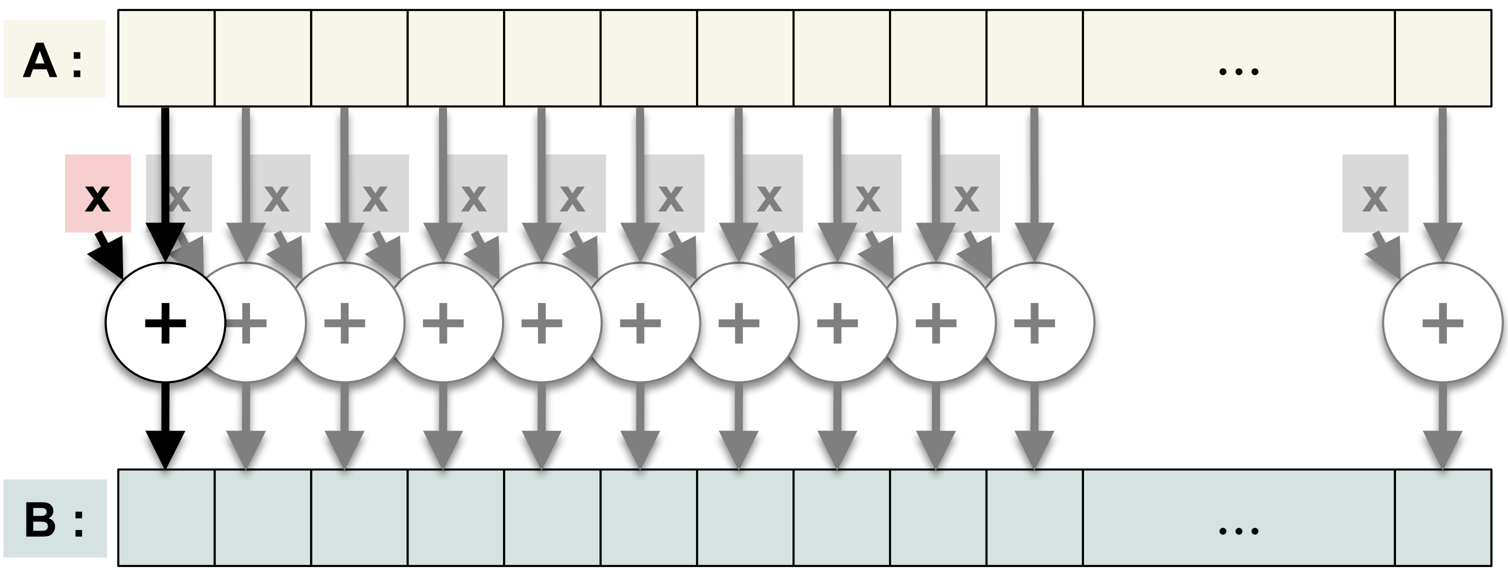


Vector Processing

- Scalar vs. vector processor

```
for(i=0; i<1000; ++i) {
    B[i] = A[i] + x;
}
```

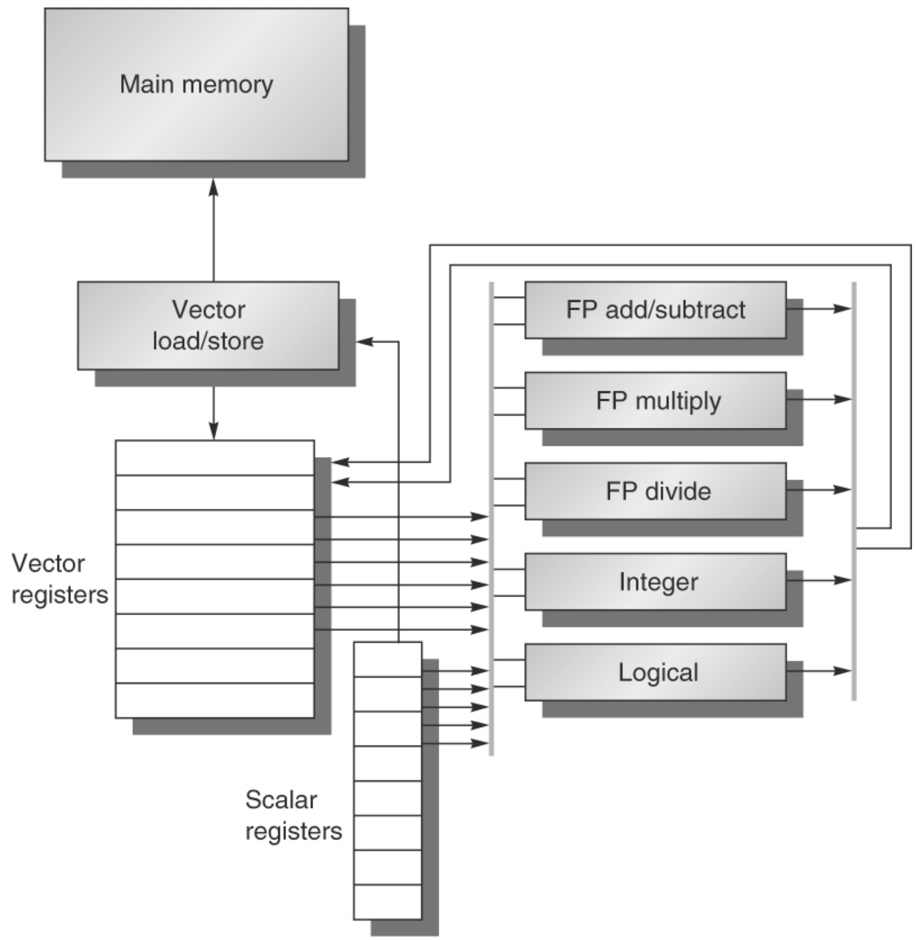
→ **vadd v3, v2, v1**



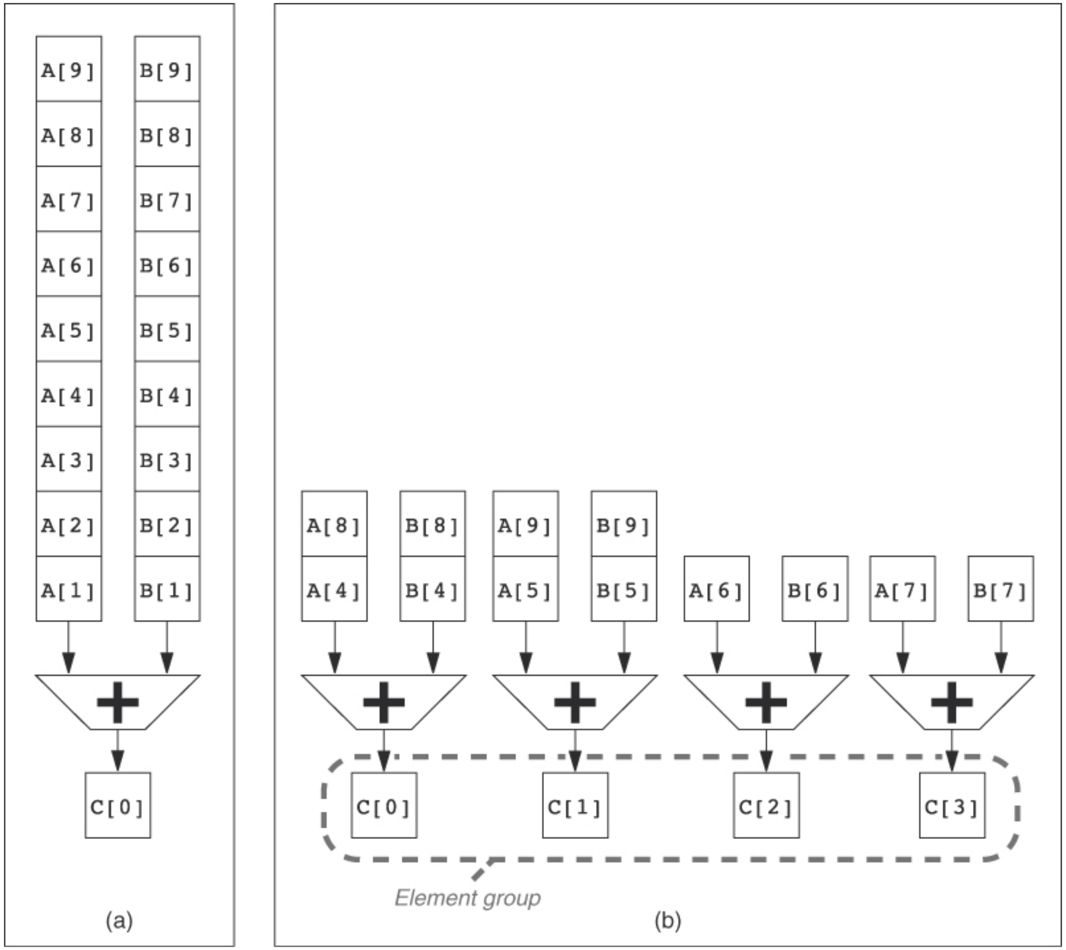
Vector Processor

- A scalar processor—e.g., MIPS
  - Scalar register file
  - Scalar functional units
- Vector register file
  - 2D register array
  - Each register is an array of registers
  - The number of elements per register determines the max vector length
- Vector functional units
  - Single opcode activates multiple units
  - Integer, floating point, load and stores

Basic Vector Processor Architecture



Parallel vs. Pipeline Units



Vector Instruction Set Architecture

- Single instruction defines multiple operations
  - Lower instruction fetch/decode/issue cost
- Operations are executed in parallel
  - Naturally no dependency among data elements
  - Simple hardware
- Predictable memory access pattern
  - Improve performance via prefetching
  - Simple memory scheduling policy
  - Multi banking may be used for improving bandwidth

Vector Operation Length

- Fixed in hardware
  - Common in narrow SIMD
  - Not efficient for wide SIMD

- Variable length
  - Determined by a vector length register (VLR)
  - MVL is the maximum VL
  - How to process vectors wider than MVL?

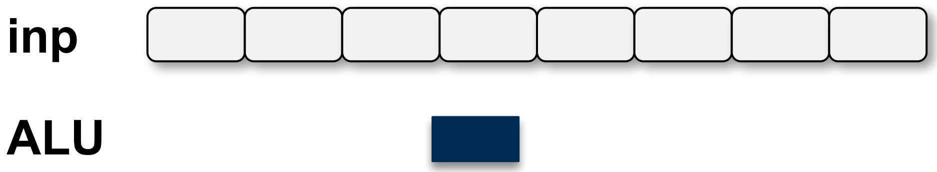
Conditional Execution

- Question: how to handle branches?
- Solution: by predication
  - Use masks, flag vectors with single-bit elements
  - Determine the flag values based on vector compare
  - Use flag registers as control mask for the next vector operations

```
for(i=0; i<1000; ++i) {
    if(A[i] !=B[i])
        A[i]  -=  B[i];
}
```

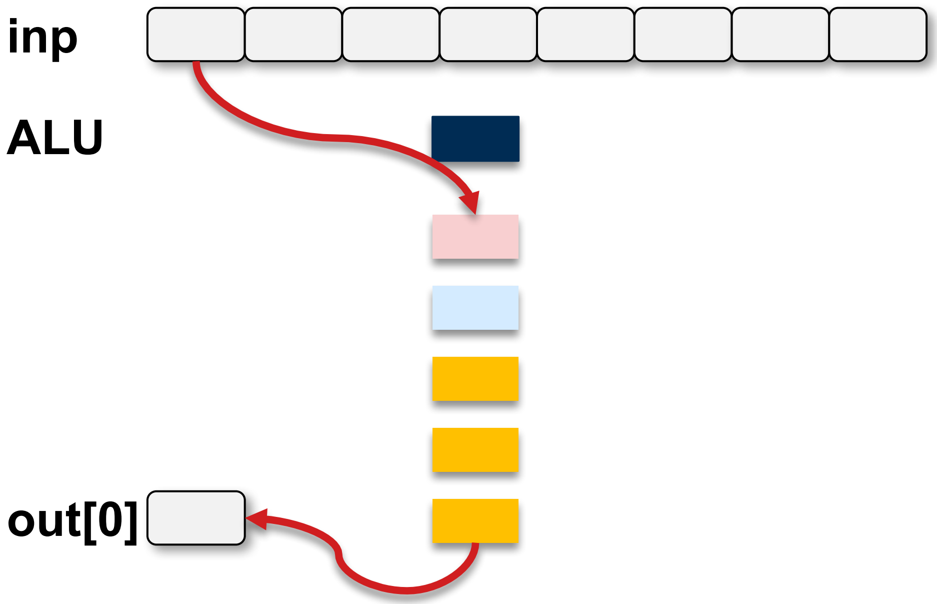
```
vld V1, Ra
vld V2, Rb
vcmp.neq.vv M0, V1, V2
vsub.vv V3, V2, V1, M0
vst V3, Ra
```

Branches in Scalar Processors



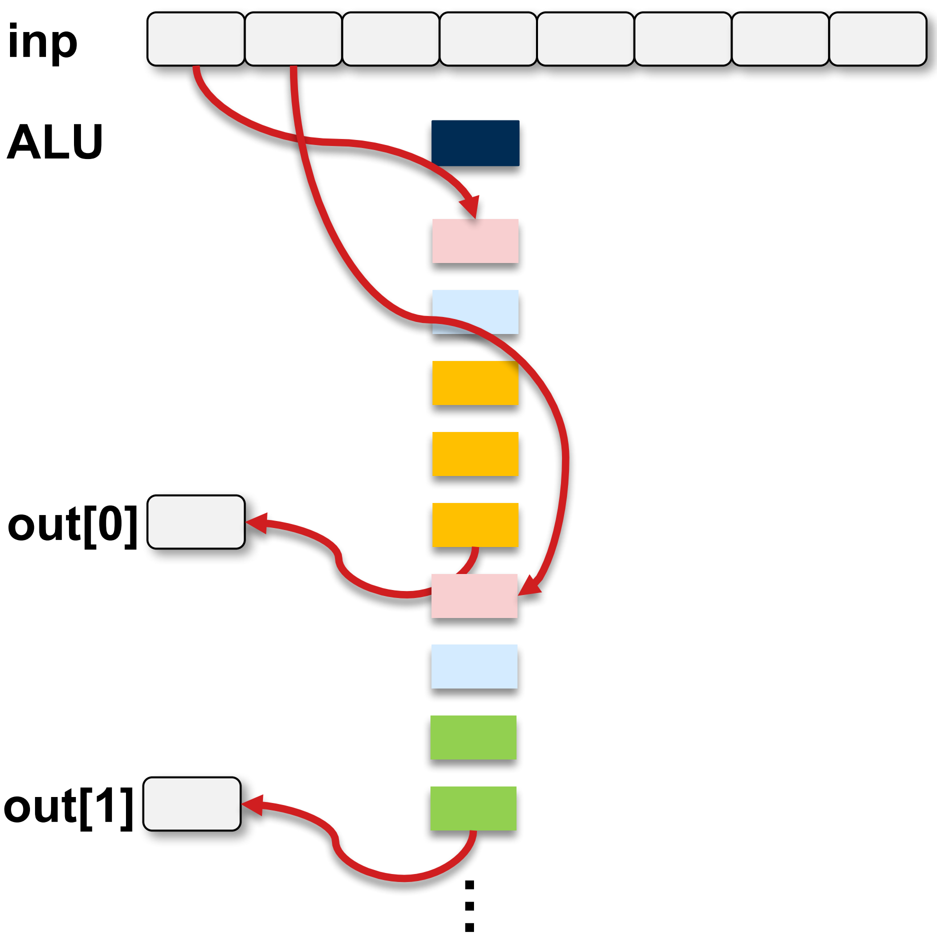

```
for (i =0; i < 8; ++i) {
    if (inp[i] > 0) {
        y = inp[i] * inp[i];
        y = y + 2 * inp[i];
        out[i] = y + 3;
    } else {
        y = 4 * inp[i];
        out[i] = y + 1;
    }
}
```

Branches in Scalar Processors



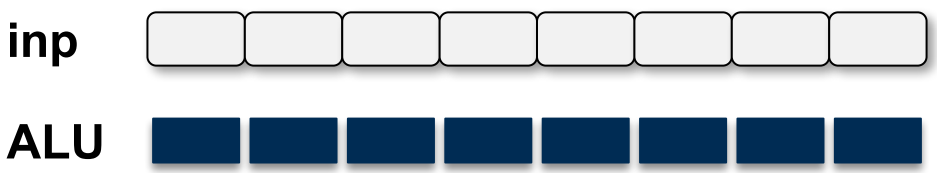

```
for (i =0; i < 8; ++i) {
    if (inp[i] > 0) {
        y = inp[i] * inp[i];
        y = y + 2 * inp[i];
        out[i] = y + 3;
    } else {
        y = 4 * inp[i];
        out[i] = y + 1;
    }
}
```

Branches in Scalar Processors



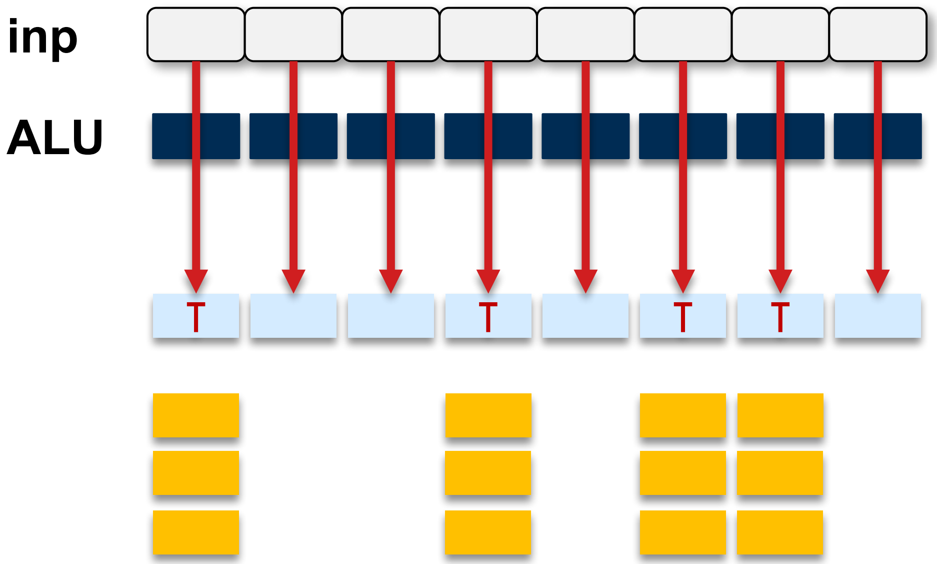

```
for (i =0; i < 8; ++i) {
    if (inp[i] > 0) {
        y = inp[i] * inp[i];
        y = y + 2 * inp[i];
        out[i] = y + 3;
    } else {
        y = 4 * inp[i];
        out[i] = y + 1;
    }
}
```

Branches in Vector Processors



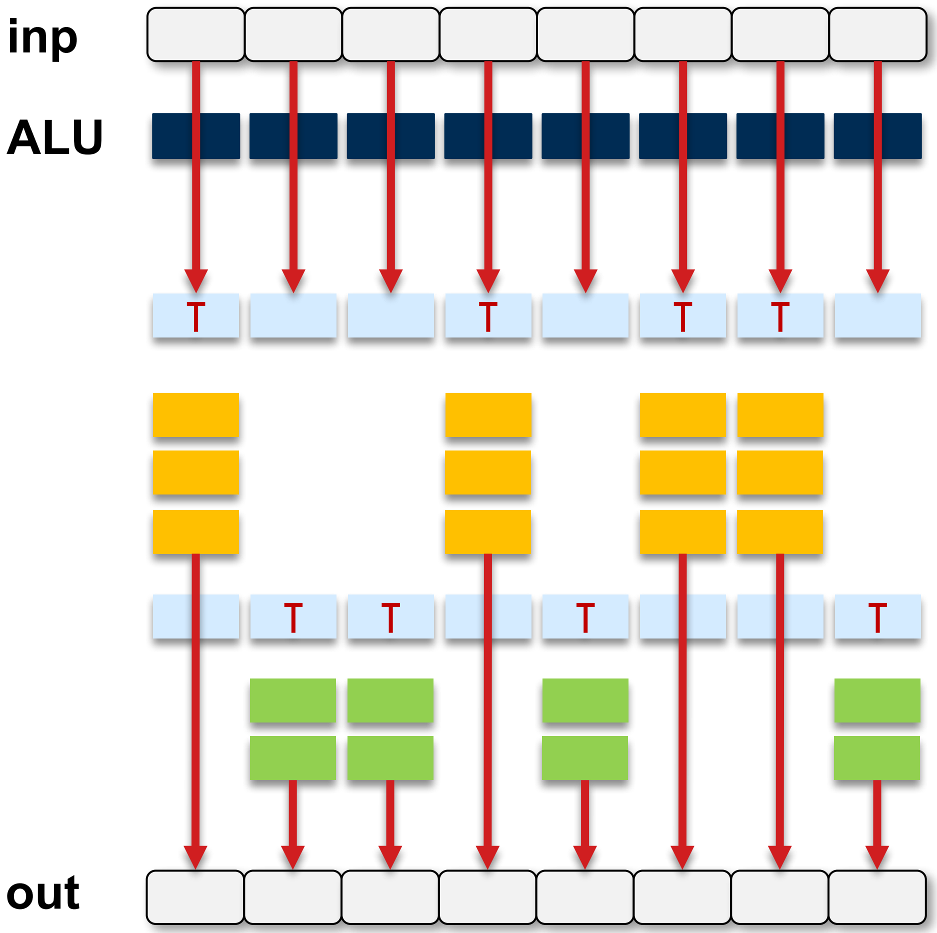

```
if (inp[i] > 0) {
    y = inp[i] * inp[i];
    y = y + 2 * inp[i];
    out[i] = y + 3;
} else {
    y = 4 * inp[i];
    out[i] = y + 1;
}
```

Branches in Vector Processors




```
if (inp[i] > 0) {
    y = inp[i] * inp[i];
    y = y + 2 * inp[i];
    out[i] = y + 3;
} else {
    y = 4 * inp[i];
    out[i] = y + 1;
}
```

Branches in Vector Processors

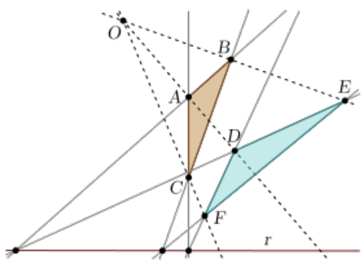



```
if (inp[i] > 0) {
    y = inp[i] * inp[i];
    y = y + 2 * inp[i];
    out[i] = y + 3;
} else {
    y = 4 * inp[i];
    out[i] = y + 1;
}
```

Graphics Processing Unit (GPU)

Graphics Processing Unit

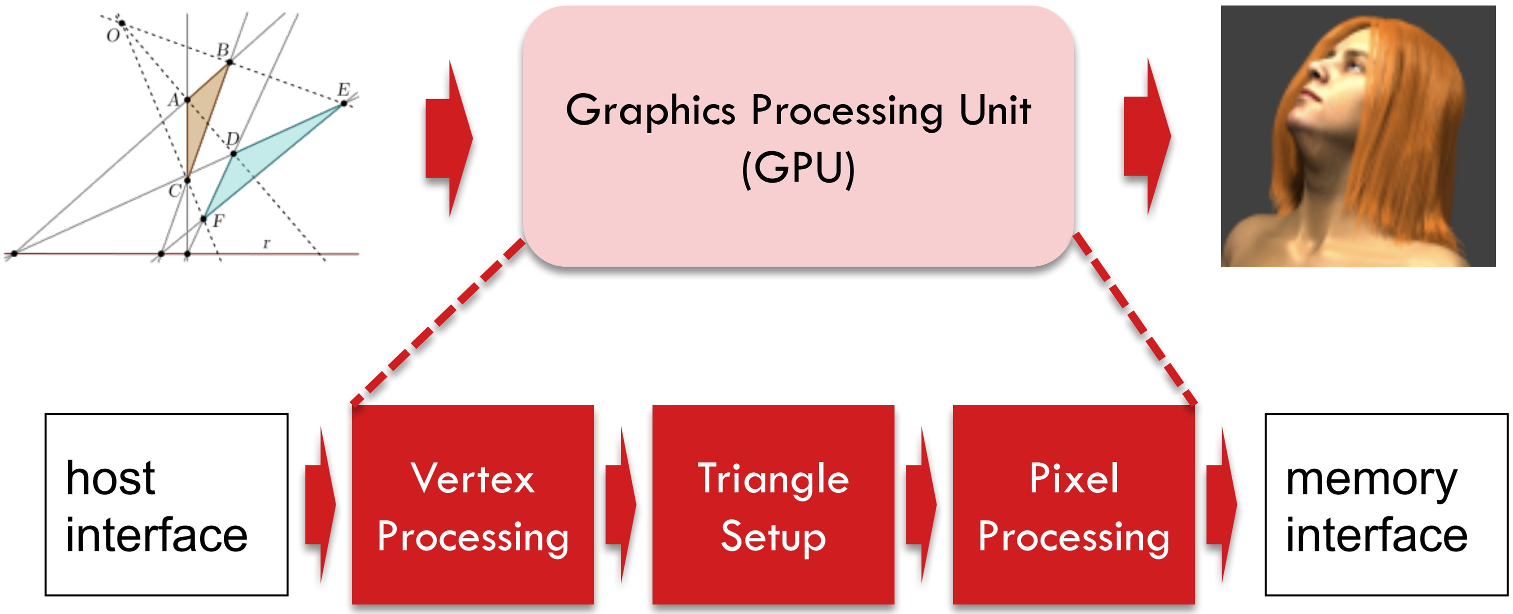
- Initially developed as graphics accelerator
  - It receives geometry information from the CPU as an input and provides a picture as an output



Graphics Processing Unit (GPU)

Graphics Processing Unit

- Initially developed as graphics accelerator
  - It receives geometry information from the CPU as an input and provides a picture as an output

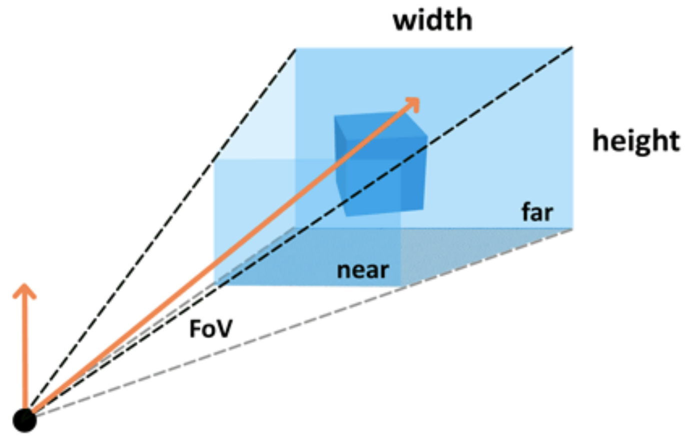


Host Interface

- The host interface is the communication bridge between the CPU and the GPU
- It receives commands from the CPU and also pulls geometry information from system memory
- It outputs a *stream* of vertices in object space with all their associated information

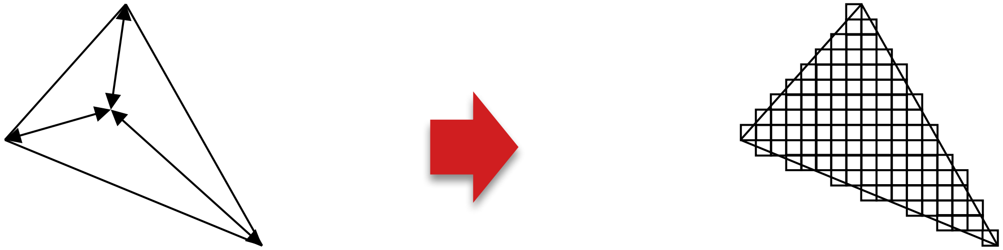
Vertex Processing

- The vertex processing stage receives vertices from the host interface in object space and outputs them in screen space
- This may be a simple linear transformation, or a complex operation involving morphing effects



Pixel Processing

- Rasterize triangles to pixels
- Each fragment provided by triangle setup is fed into fragment processing as a set of attributes (position, normal, texcoord etc), which are used to compute the final color for this pixel
- The computations taking place here include texture mapping and math operations

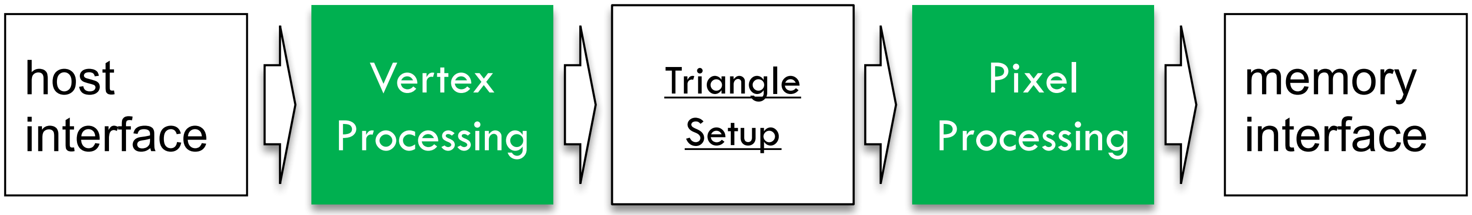


Programming GPUs

- The programmer can write programs that are executed for every vertex as well as for every fragment
- This allows fully customizable geometry and shading effects that go well beyond the generic look and feel of older 3D applications

Programming GPUs

- The programmer can write programs that are executed for every vertex as well as for every fragment
- This allows fully customizable geometry and shading effects that go well beyond the generic look and feel of older 3D applications

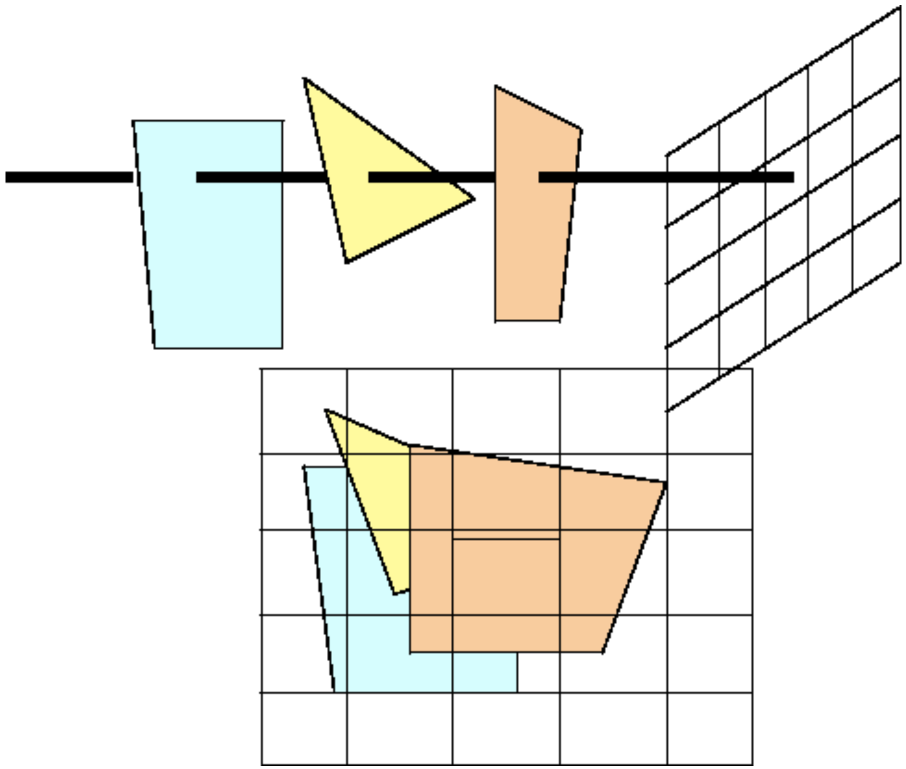


Memory Interface

- Fragment colors provided by the previous stage are written to the framebuffer
- Used to be the biggest bottleneck before fragment processing took over
- Before the final write occurs, some fragments are rejected by the zbuffer, stencil and alpha tests
- On modern GPUs, z and color are compressed to reduce framebuffer bandwidth (but not size)

Z-Buffer

- Example of 3 objects

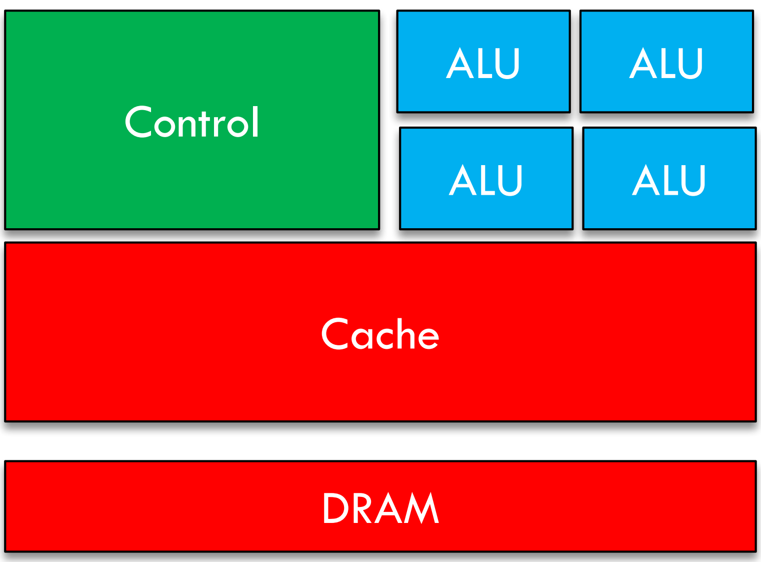


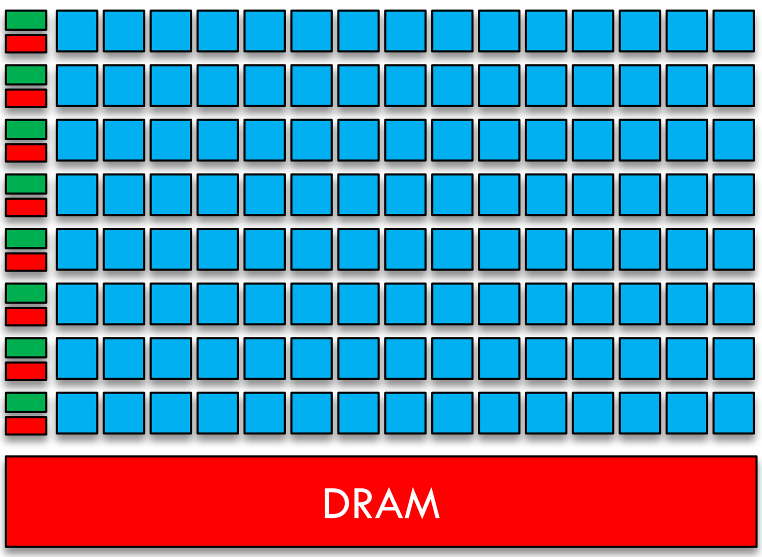
Graphics Processing Unit

- Initially developed as graphics accelerators
  - one of the densest compute engines available now
- Many efforts to run non-graphics workloads on GPUs
  - general-purpose GPUs (GPGPUs)
- C/C++ based programming platforms
  - CUDA from NVidia and OpenCL from an industry consortium
- A heterogeneous system
  - a regular host CPU
  - a GPU that handles CUDA (may be on the same CPU chip)

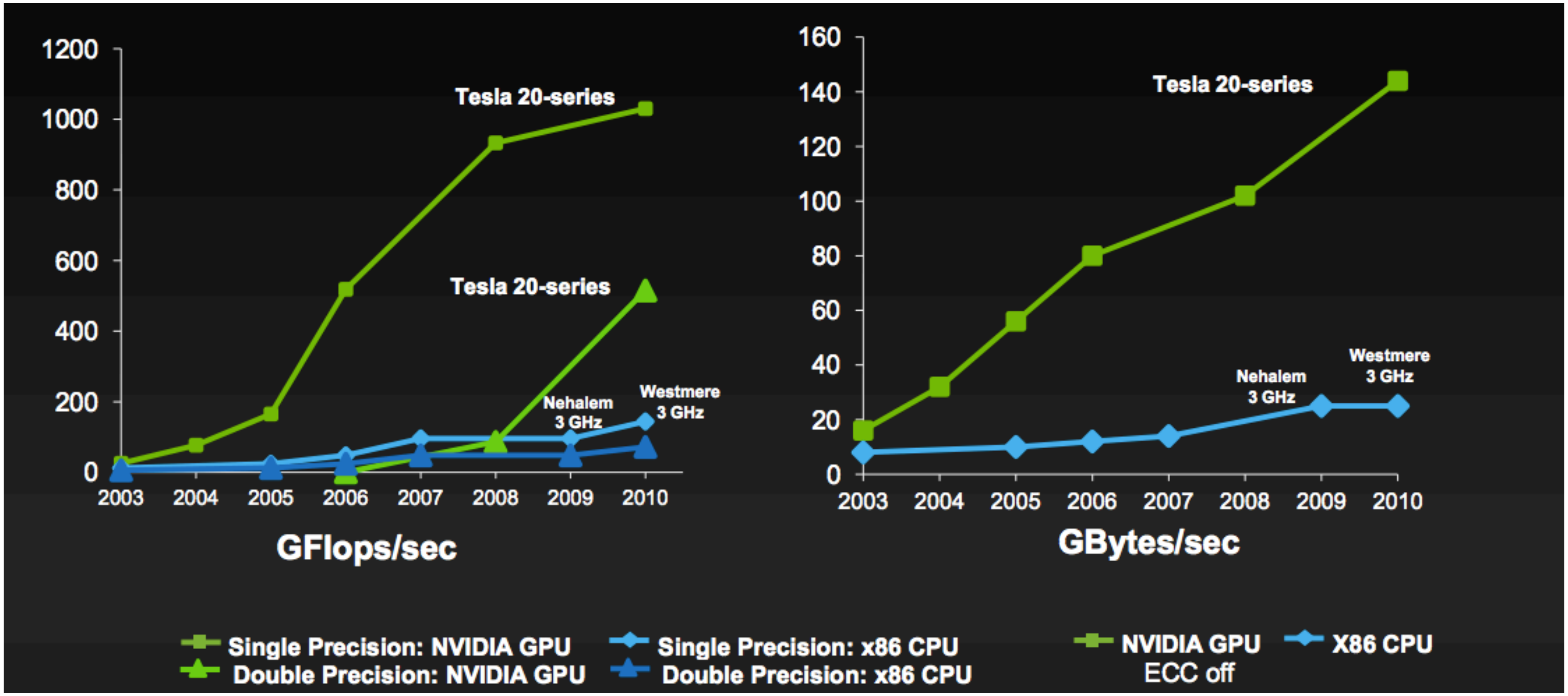
Graphics Processing Unit

- Simple in-order pipelines that rely on thread-level parallelism to hide long latencies
- Many registers (~1K) per in-order pipeline (lane) to support many active warps





Why GPU Computing?



Source: NVIDIA

The GPU Architecture

- SIMT – single instruction, multiple threads
  - GPU has many SIMT cores
- Application → many thread blocks (1 per SIMT core)
- Thread block → many warps (1 warp per SIMT core)
- Warp → many in-order pipelines (SIMD lanes)