

# Parallel Architectures

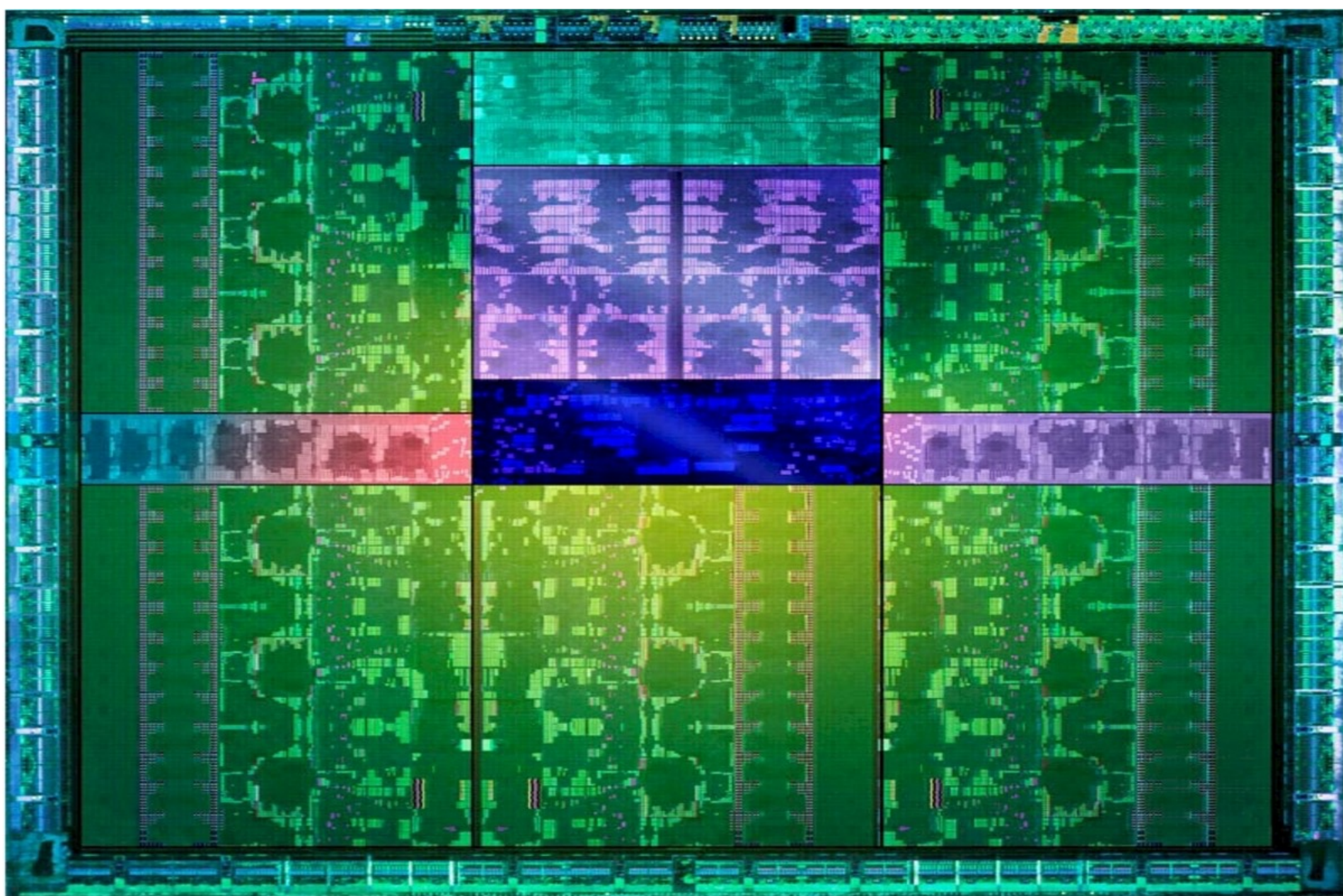
Chris Rossbach

# Outline for Today

- Questions?
- Administrivia
  - Exam soon
- Agenda
  - Parallel Architectures (GPU background)

# Faux Quiz questions

- What is hardware multi-threading; what problem does it solve?
- What is the difference between a vector processor and a scalar?
- Implement a parallel scan or reduction
- How are GPU workloads different from GPGPU workloads?
- How does SIMD differ from SIMT?
- List and describe some pros and cons of vector/SIMD architectures.
- GPUs historically have elided cache coherence. Why? What impact does it have on the the programmer?
- List some ways that GPUs use concurrency but not necessarily parallelism.



# A modern GPU



Also:  
CU or ACE

Roughly: all  
of pfxsum  
1,000s X/sec

- 80 SMs
  - Streaming Multiprocessor
  - 64 cores/SM
  - 5210 threads!
  - 15.7 TFLOPS
- 640 Tensor cores
- HBM2 memory
  - 4096-bit bus
  - No cache coherence!
- 16 GB memory
  - PCIe-attached



*How do you program a machine like this? pthread\_create()?*

# GPUs: Outline

- Background from many areas
  - Architecture
    - Vector processors
    - Hardware multi-threading
  - Graphics
    - Graphics pipeline
    - Graphics programming models
  - Algorithms
    - parallel architectures → parallel algorithms
- Programming GPUs
  - CUDA
  - Basics: getting something working
  - Advanced: making it perform

# Architecture Review: Pipelines

Processor algorithm:

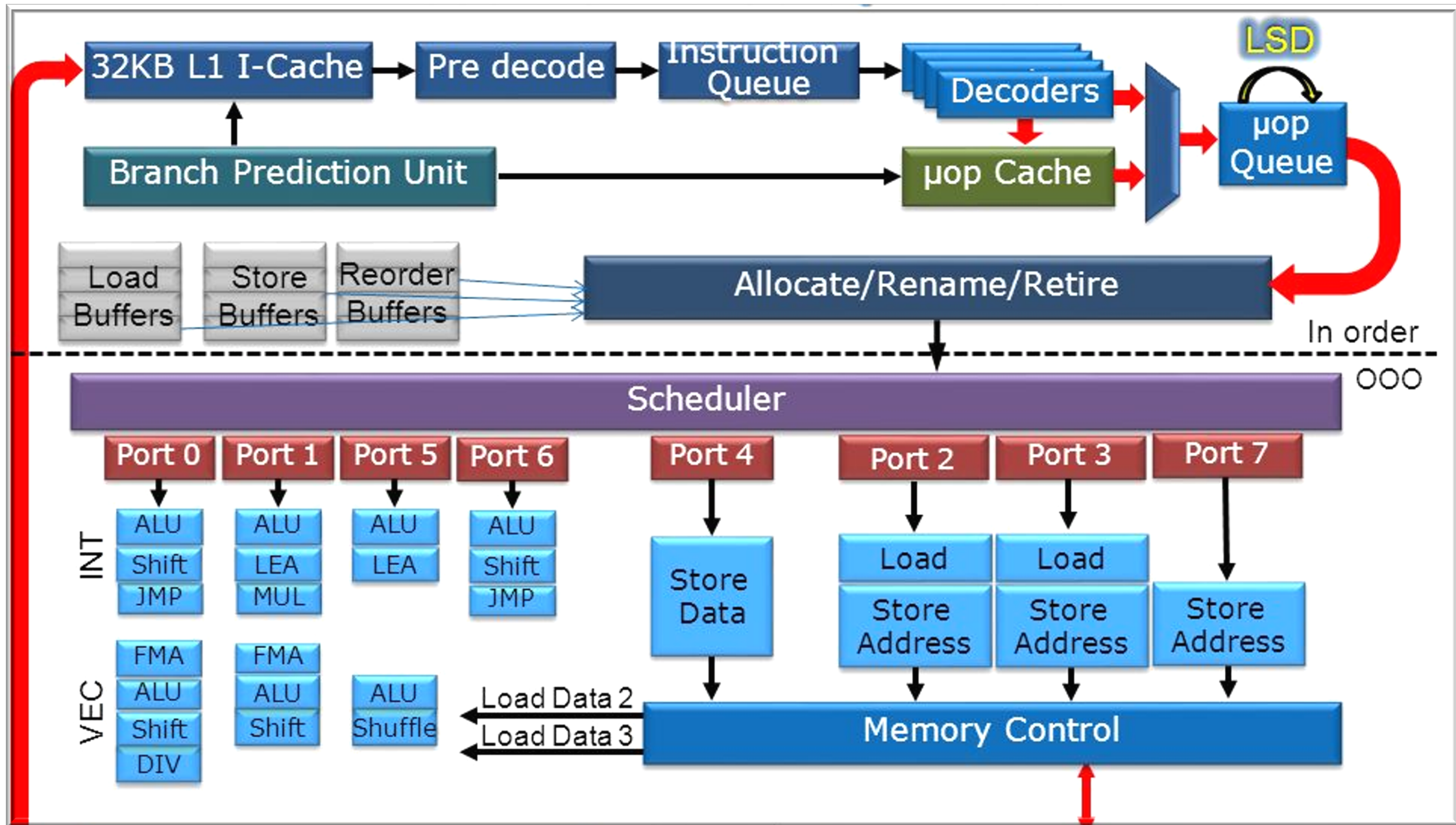
```
main() {  
    v  
    main() {  
        pthread_create(do_instructions);  
    }  
    pthread_create(do_decode);  
    pthread_create(do_execute);  
    ...  
    pthread_join(...);  
    ...  
}  
access_memory(ops, regs);  
write_back(regs);  
}
```

```
do_instructions() {  
    while(true) {  
        instruction = fetch();  
        enqueue(DECODE, instruction);  
    }  
}
```

```
do_decode() {  
    while(true) {  
        instruction = dequeue();  
        ops, regs = decode(instruction);  
        enqueue(EX, instruction);  
    }  
}
```

```
do_execute() {  
    while(true) {  
        instruction = dequeue();  
        execute_calc_addr(ops, regs);  
        enqueue(MEM, instruction);  
    }  
}
```

....

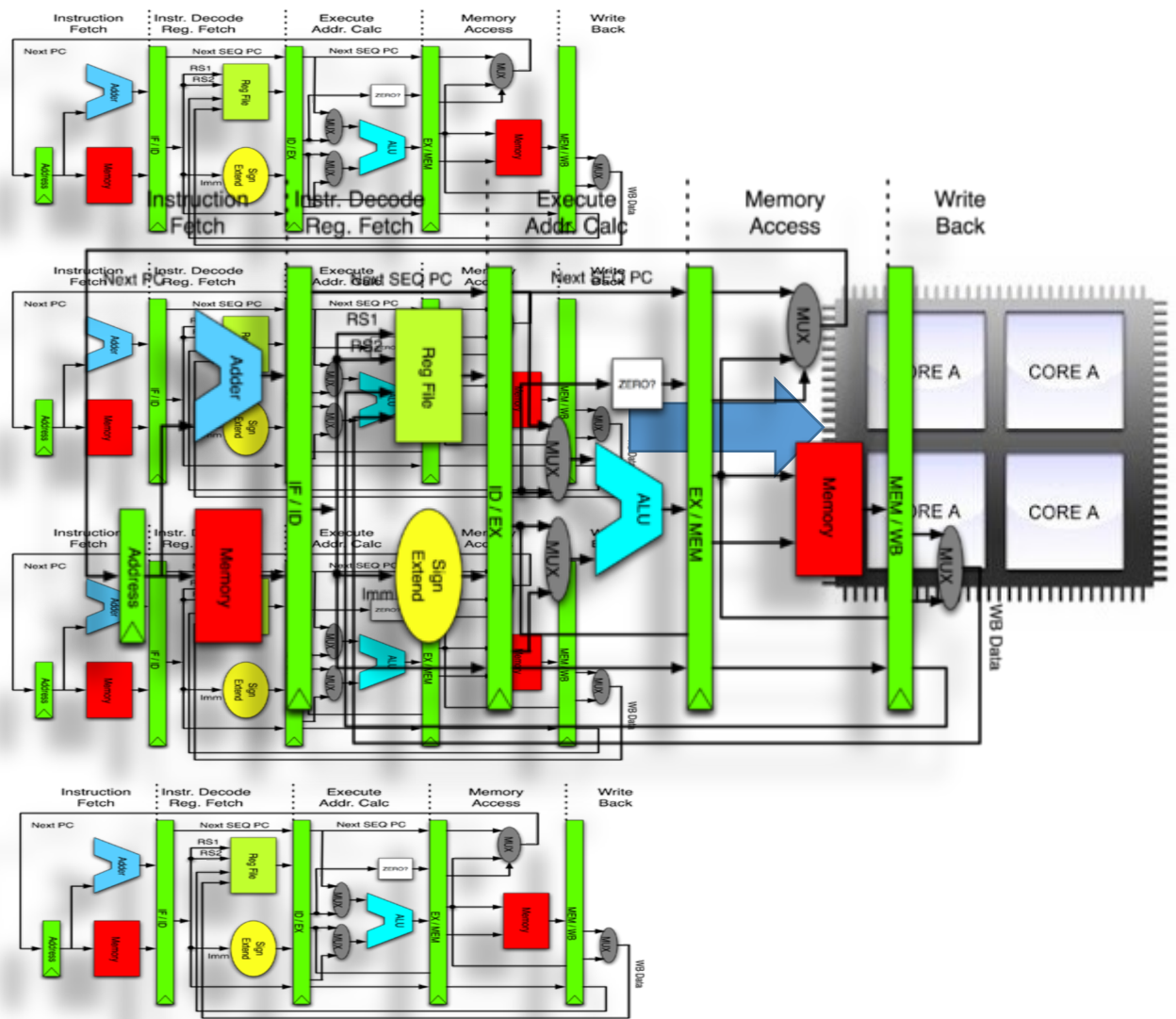


m?  
stalls?  
elism?



- *Pros: Simple*
- *Cons: programmer has to find the parallelism!*

# Multi-core/SMPs

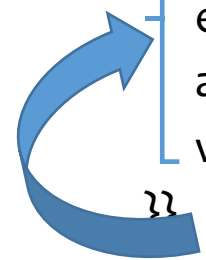


```

main() {
    for(i=0; i<CORES; i++) {
        pthread_create(
            do_instructions());
    }
}

do_instructions() {
    while(true) {
        instruction = fetch();
        ops, regs = decode(instruction);
        execute_calc_addrs(ops, regs);
        access_memory(ops, regs);
        write_back(regs);
    }
}

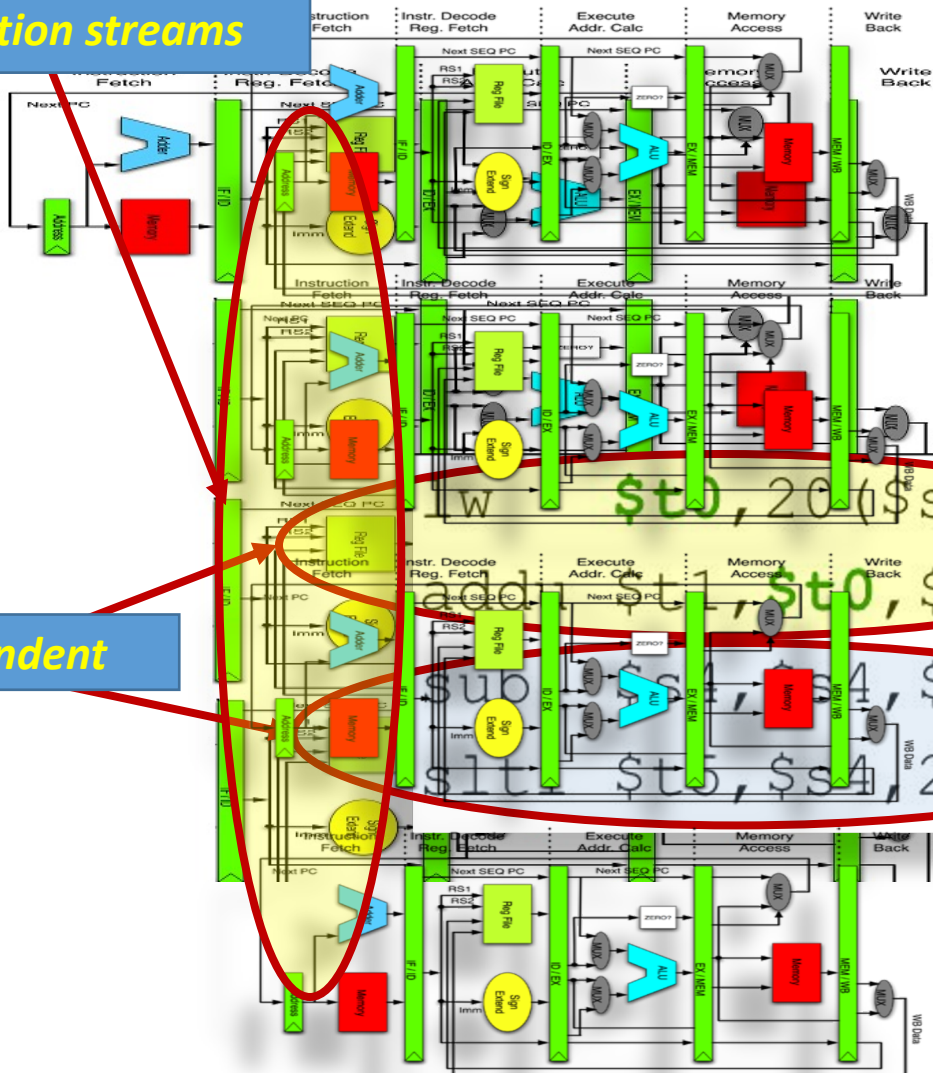
```



*Other techniques extract parallelism here, try to let the machine find parallelism*

# Superscalar processors

Remove extra instruction streams



independent

```
main() {  
  for(i=0; i<CORES; i++)  
    pthread_create(&decode_exec);  
  while(true) {  
    instruction = fetch();  
    enqueue(instruction);  
  }  
}
```

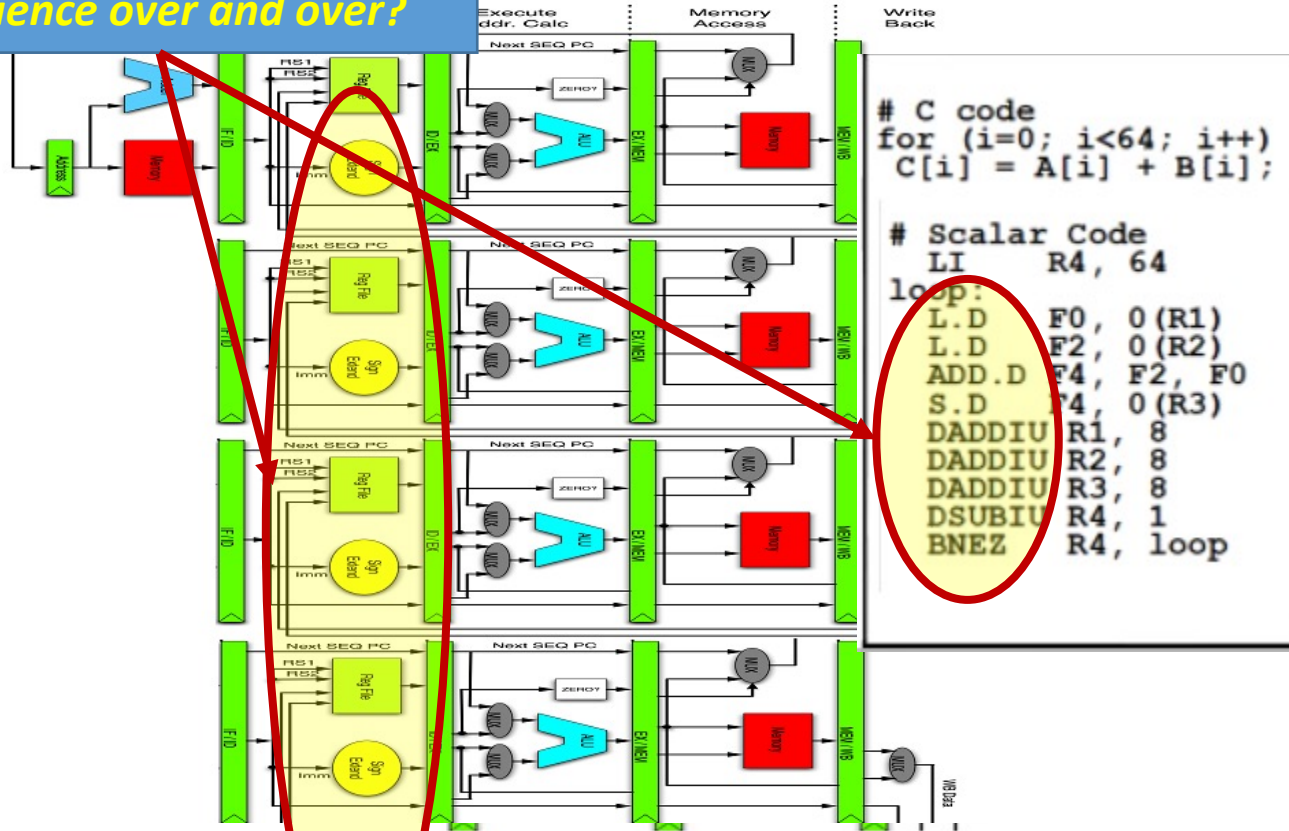
```
decode_exec() {  
  instruction = dequeue();  
  ops, regs = decode(instruction);  
  execute_calc_addrs(ops, regs);  
  access_memory(ops, regs);  
  write_back(regs);  
}
```

Doesn't look that different does it? Why do it?

Enables independent instruction parallelism.

# Vector/SIMD processors

Why decode same instruction sequence over and over?



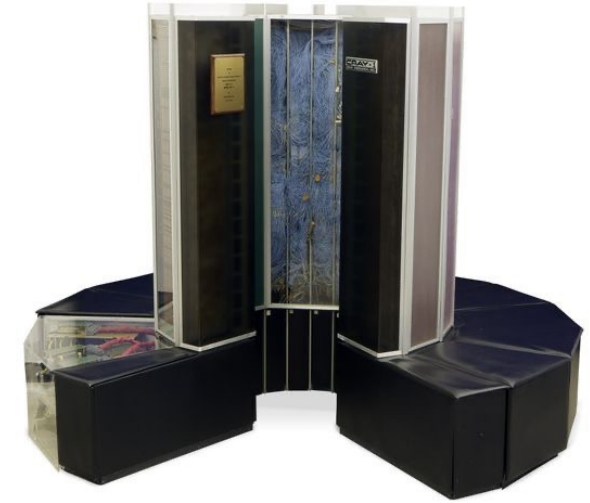
```
main() {
  for(i=0; i<CORES; i++)
    pthread_create(exec);
  while(true) {
    ops, regs = fetch_decode();
    enqueue(ops, regs);
  }
}

exec() {
  ops, regs = dequeue();
  execute_calc_addr(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Single instruction stream, multiple computations  
But now all my instructions need multiple operands!

# Vector Processors

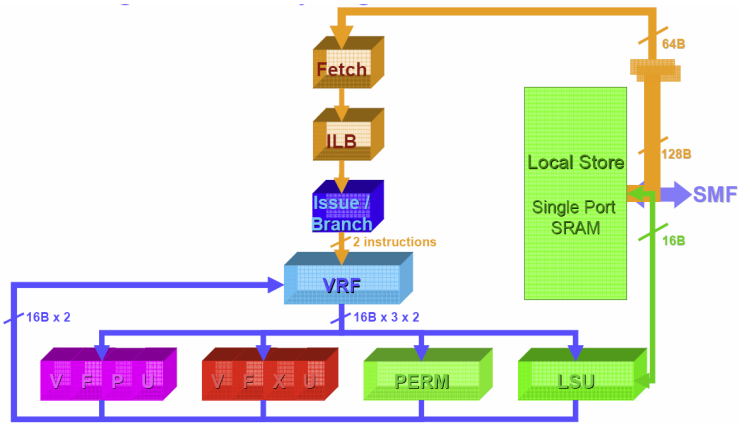
- Process multiple data elements simultaneously.
- Common in supercomputers of the 1970's 80's and 90's.
- Modern CPUs support some vector processing instructions
  - Usually called SIMD
- Can operate on a few vectors elements per clock cycle in a pipeline or,
  - SIMD operate on all per clock cycle
- 1962 University of Illinois Illiac IV - completed 1972 → 64 ALUs 100-150 MFlops
- (1973) TI's Advance Scientific Computer (ASC) 20-80 MFlops
- (1975) Cray-1 first to have vector registers instead of keeping data in memory



*Single instruction stream, multiple data →  
Programming model has to change*

# Vector Processors

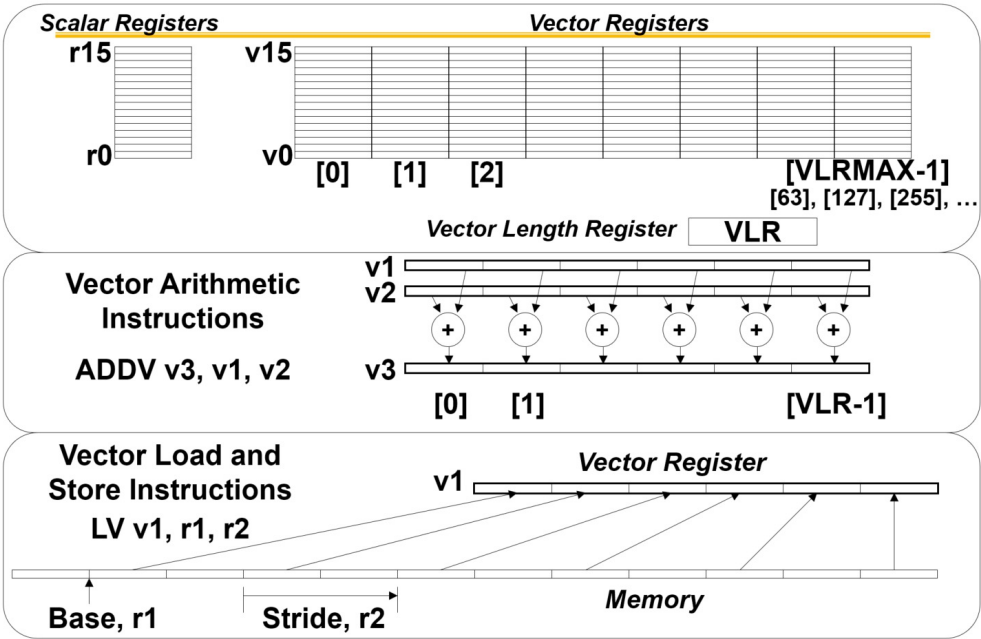
*GPUs: same basic idea*



Source: Gschwind et al., Hot Chips 17, 2005

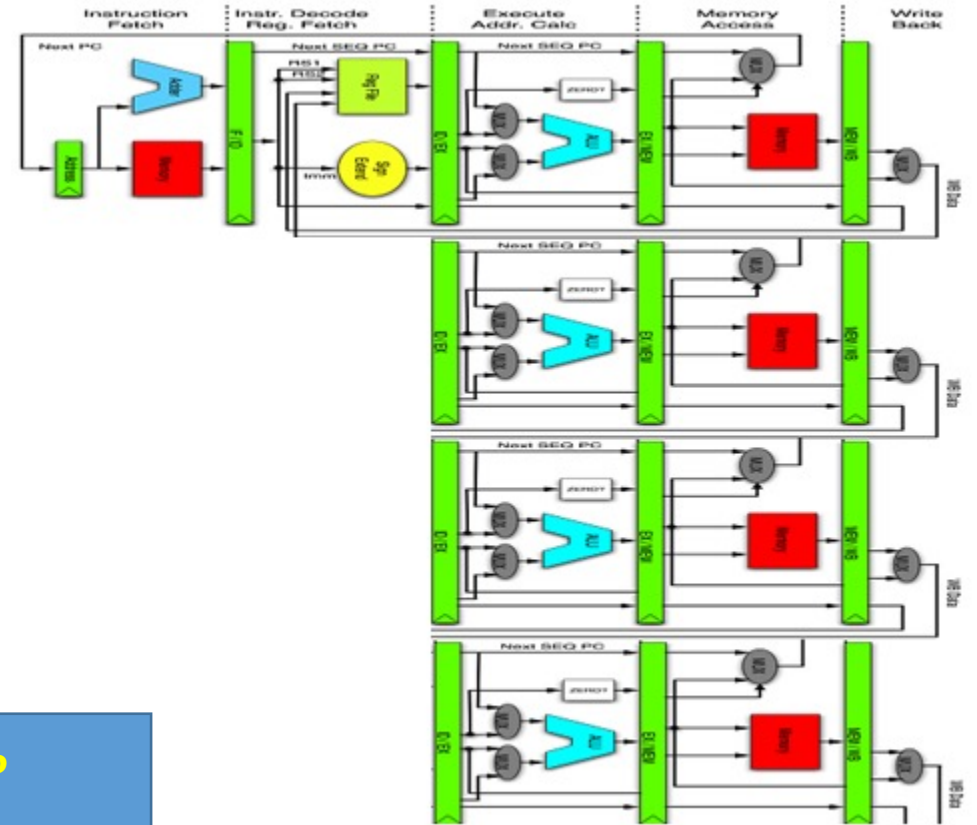
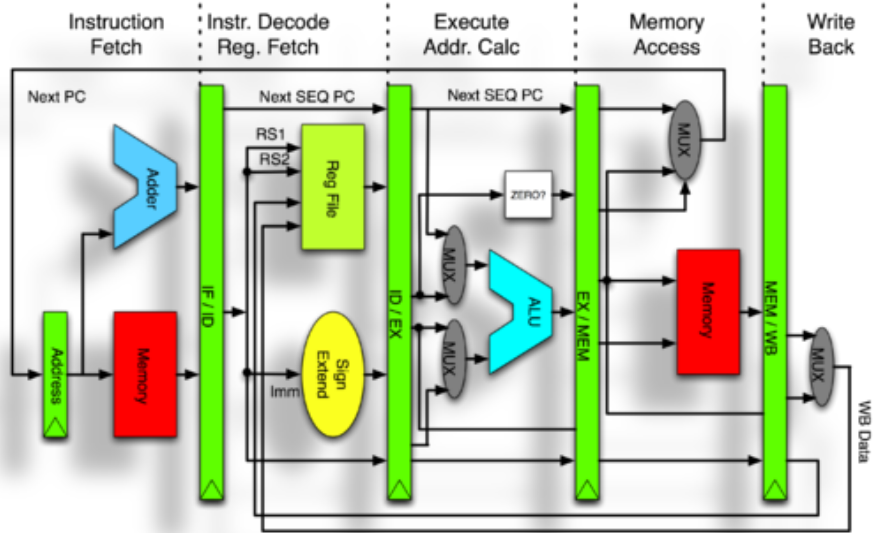
## Implementation:

- Instruction fetch control logic shared
- Same instruction stream executed on
- Multiple pipelines
- Multiple different operands in parallel



<pre># C code for (i=0; i&lt;64; i++)   C[i] = A[i] + B[i];</pre>	<pre># Scalar Code LI      R4, 64 loop: L.D     F0, 0(R1) L.D     F2, 0(R2) ADD.D   F4, F2, F0 S.D     F4, 0(R3) DADDIU  R1, 8 DADDIU  R2, 8 DADDIU  R3, 8 DSUBIU  R4, 1 BNEZ   R4, loop</pre>	<pre># Vector Code LI      VLR, 64 LV      V1, R1 LV      V2, R2 ADDV.D  V3, V1, V2 SV      V3, R3</pre>
---	--	--

# When does vector processing help?

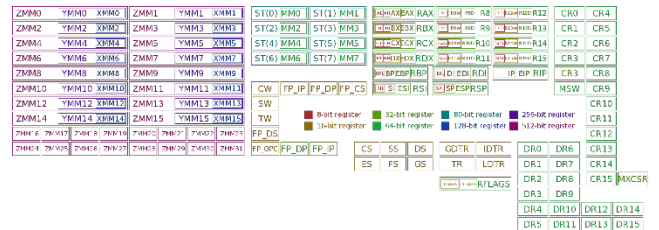
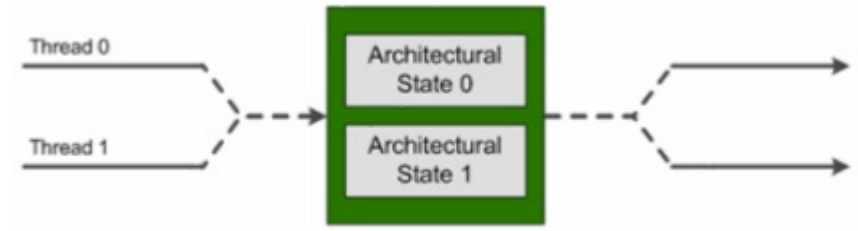
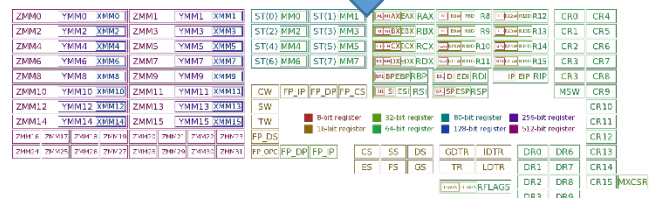
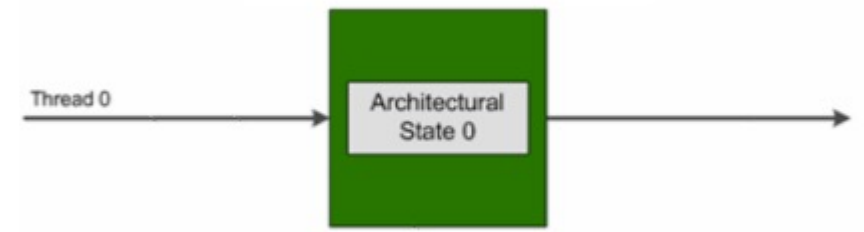
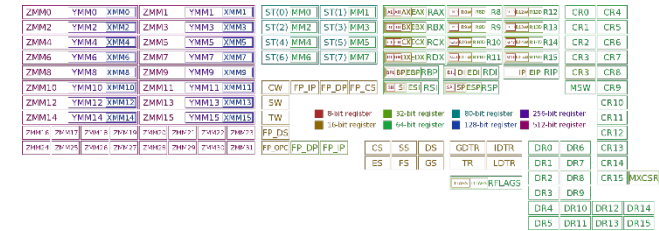


*What are the potential bottlenecks here?  
When can it improve throughput?*

*Only helps if memory can keep the pipeline busy!*

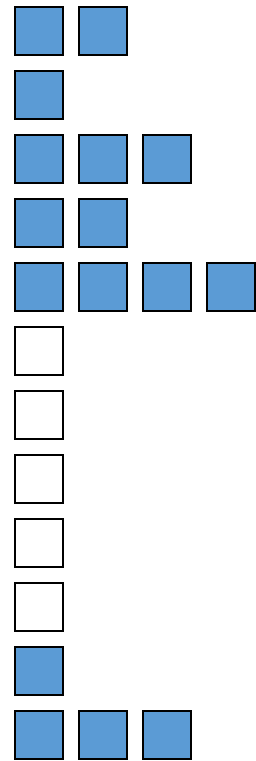
# Hardware multi-threading

- Address memory bottleneck
- Share exec unit across
  - Instruction streams
  - Switch on stalls
- Looks like multiple cores to the OS
- Three variants:
  - Coarse
  - Fine-grain
  - Simultaneous

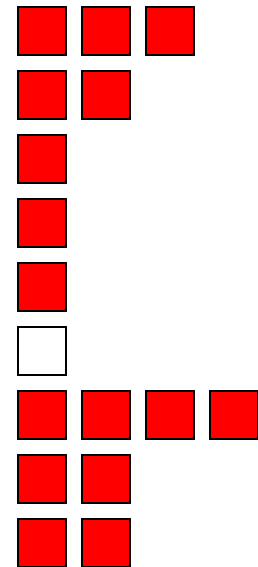


# Running example

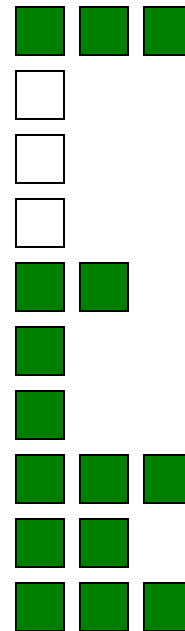
Thread A



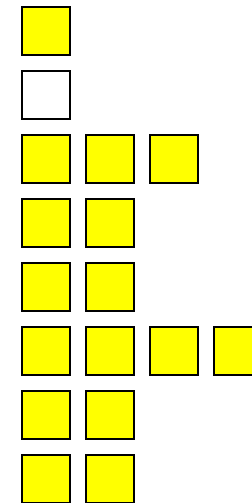
Thread B



Thread C



Thread D

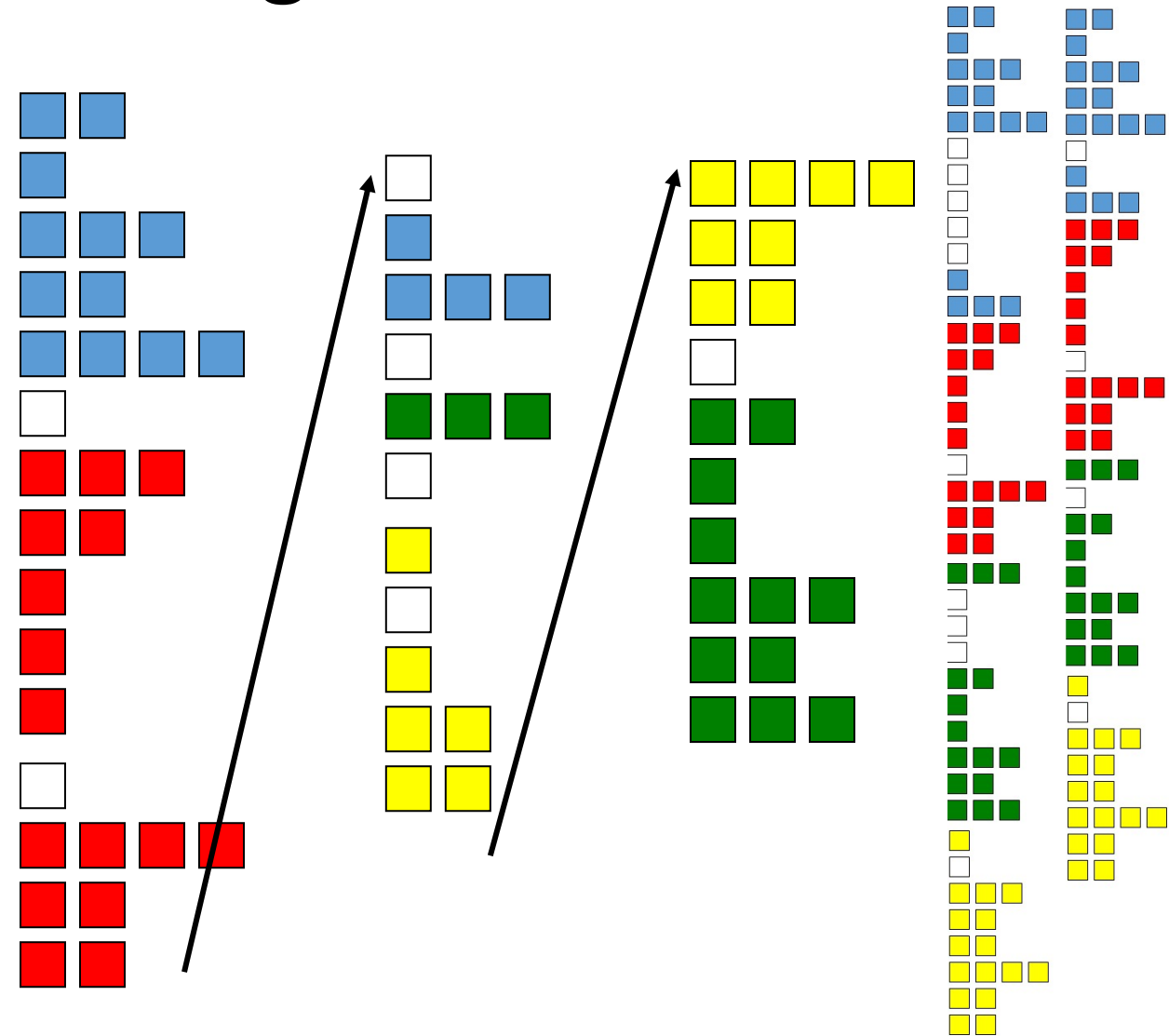


- Colors → pipeline full
- White → stall



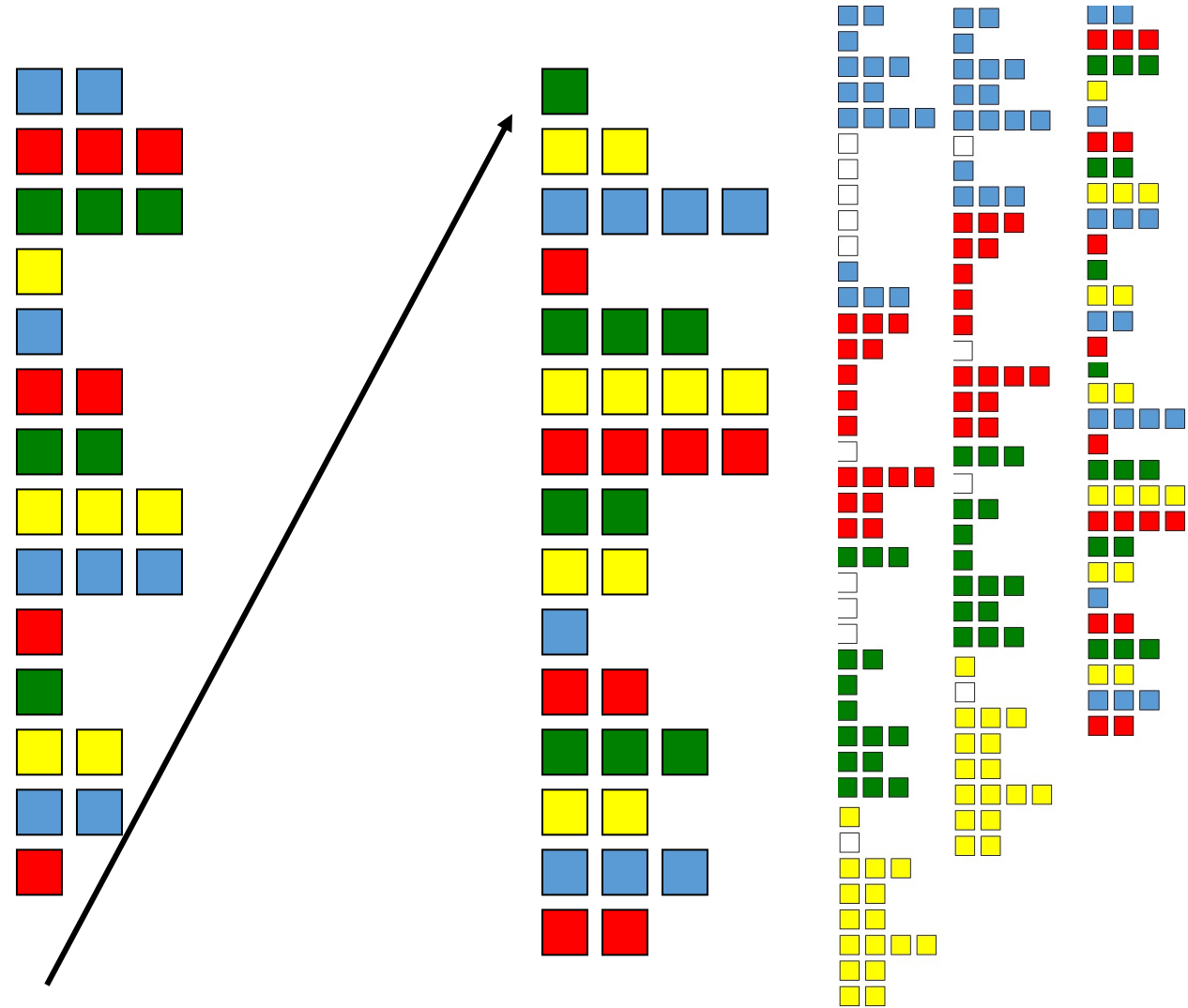
# Coarse-grained multithreading

- Single thread runs until a costly stall
  - E.g. 2nd level cache miss
- Another thread starts during stall
  - Pipeline fill time requires several cycles!
- Does not cover short stalls
- Hardware support required
  - PC and register file for each thread
  - little other hardware
  - Looks like another physical CPU to OS/software



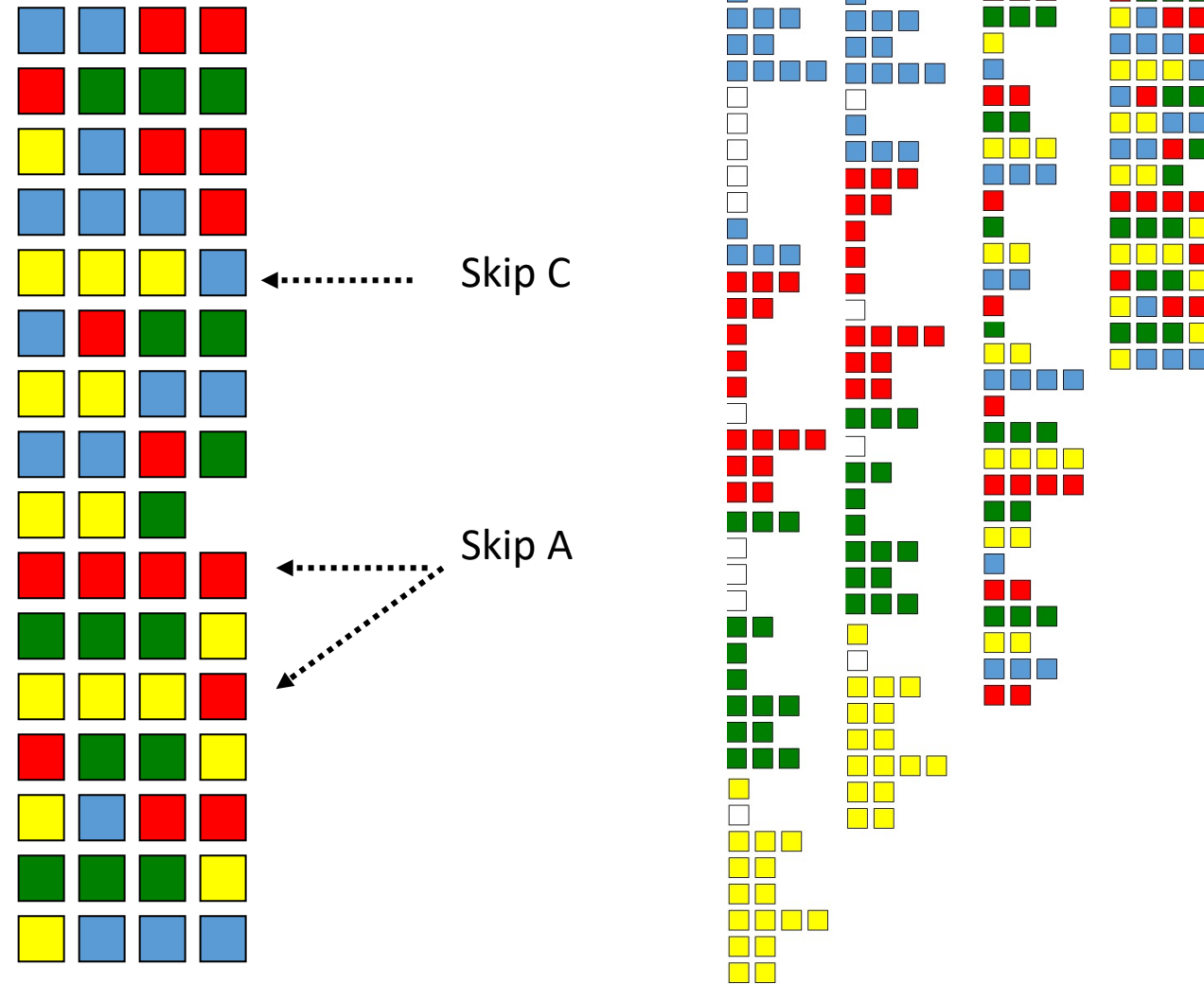
# Fine-grained multithreading

- Threads interleave instructions
  - Round-robin
  - Skip stalled threads
- Hardware support required
  - Separate PC and register file per thread
  - Hardware to control alternating pattern
- Naturally hides delays
  - Data hazards, Cache misses
  - Pipeline runs with rare stalls
- Doesn't make full use of multi-issue



# Simultaneous Multithreading (SMT)

- Instructions from multiple threads issued on same cycle
  - Uses register renaming
  - dynamic scheduling facility of multi-issue architecture
- Hardware support:
  - Register files, PCs per thread
  - Temporary result registers pre commit
  - Support to sort out which threads get results from which instructions
- Maximal util. of execution units



# Why Vector and Multithreading Background?

GPU:

- A very wide vector machine
- Massively multi-threaded to hide memory latency
- Originally designed for graphics pipelines...

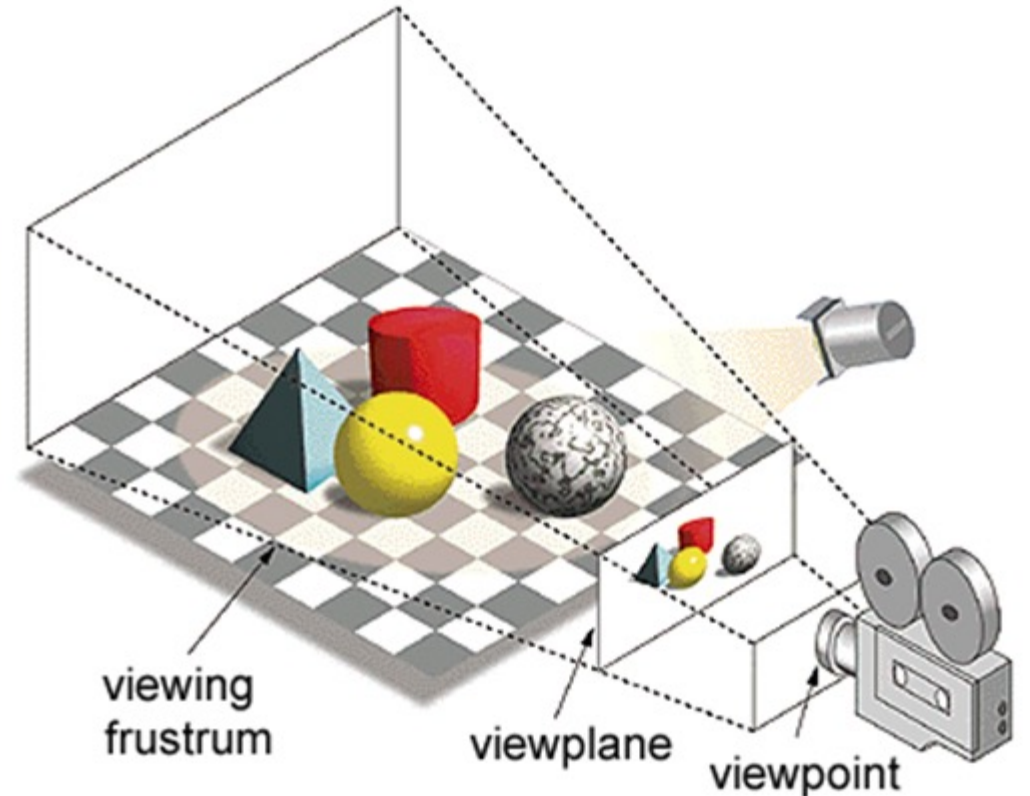
# Graphics $\approx$ Rendering

## Inputs

- 3D world model(objects, materials)
  - Geometry modeled w triangle meshes, surface normals
  - GPUs subdivide triangles into “fragments” (rasterizat
  - Materials modeled with “textures”
  - Texture coordinates, sampling “map” textures  $\rightarrow$  geometry
- Light locations and properties
  - Attempt to model surface/light interactions with modeled objects/materials
- View point

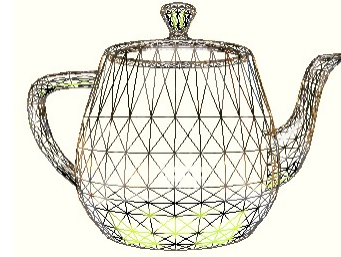
## Output

- 2D projection seen from the view-point



# Grossly over-simplified rendering algorithm

```
foreach(vertex v in model)
    map  $v_{\text{model}} \rightarrow v_{\text{view}}$ 
fragment[] frags = {};
foreach triangle t ( $v_0, v_1, v_2$ )
    frags.add(rasterize(t));
foreach fragment f in frags
    choose_color(f);
display(visible_fragments(frags));
```



# Algorithm $\rightarrow$ Graphics Pipeline

```
foreach(vertex v in model)
```

```
  map  $v_{\text{model}} \rightarrow v_{\text{view}}$ 
```

```
  fragment[] frags = {};
```

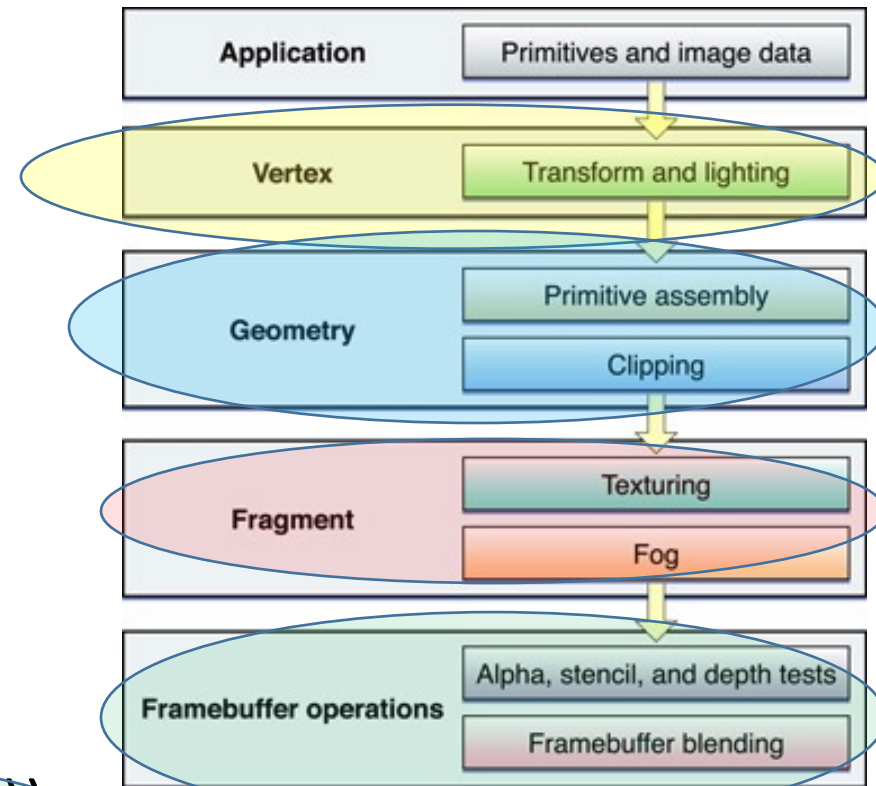
```
  foreach triangle t ( $v_0, v_1, v_2$ )
```

```
    frags.add(rasterize(t));
```

```
  foreach fragment f in frags
```

```
    choose_color(f);
```

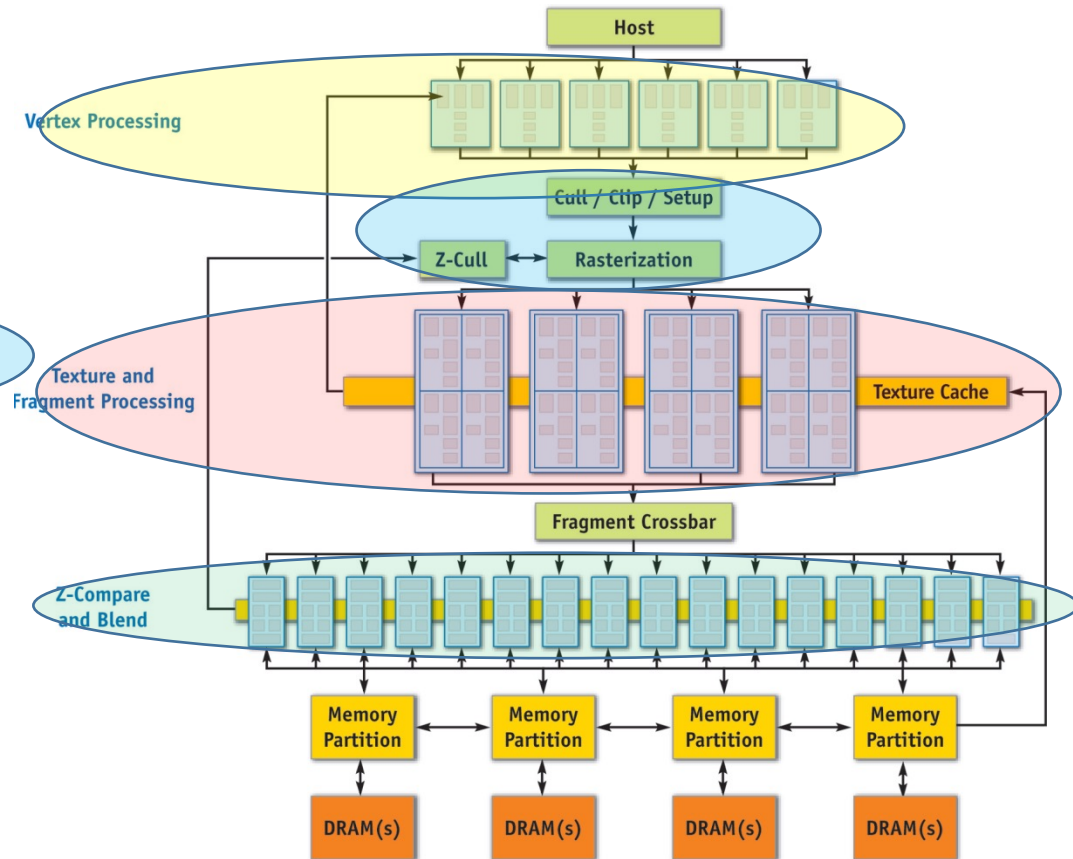
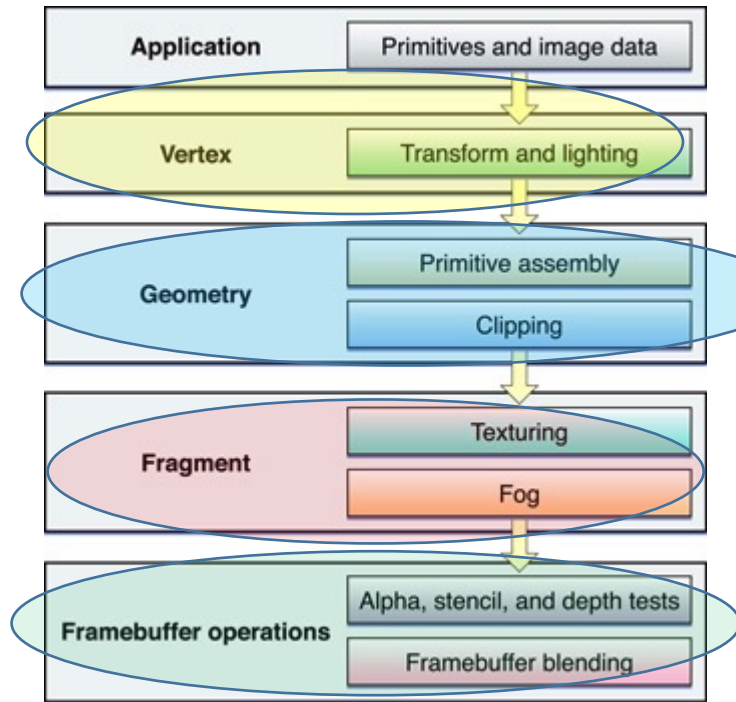
```
  display(visible_fragments(frags));
```



OpenGL pipeline

To first order, DirectX looks the same!

# Graphics pipeline → GPU architecture

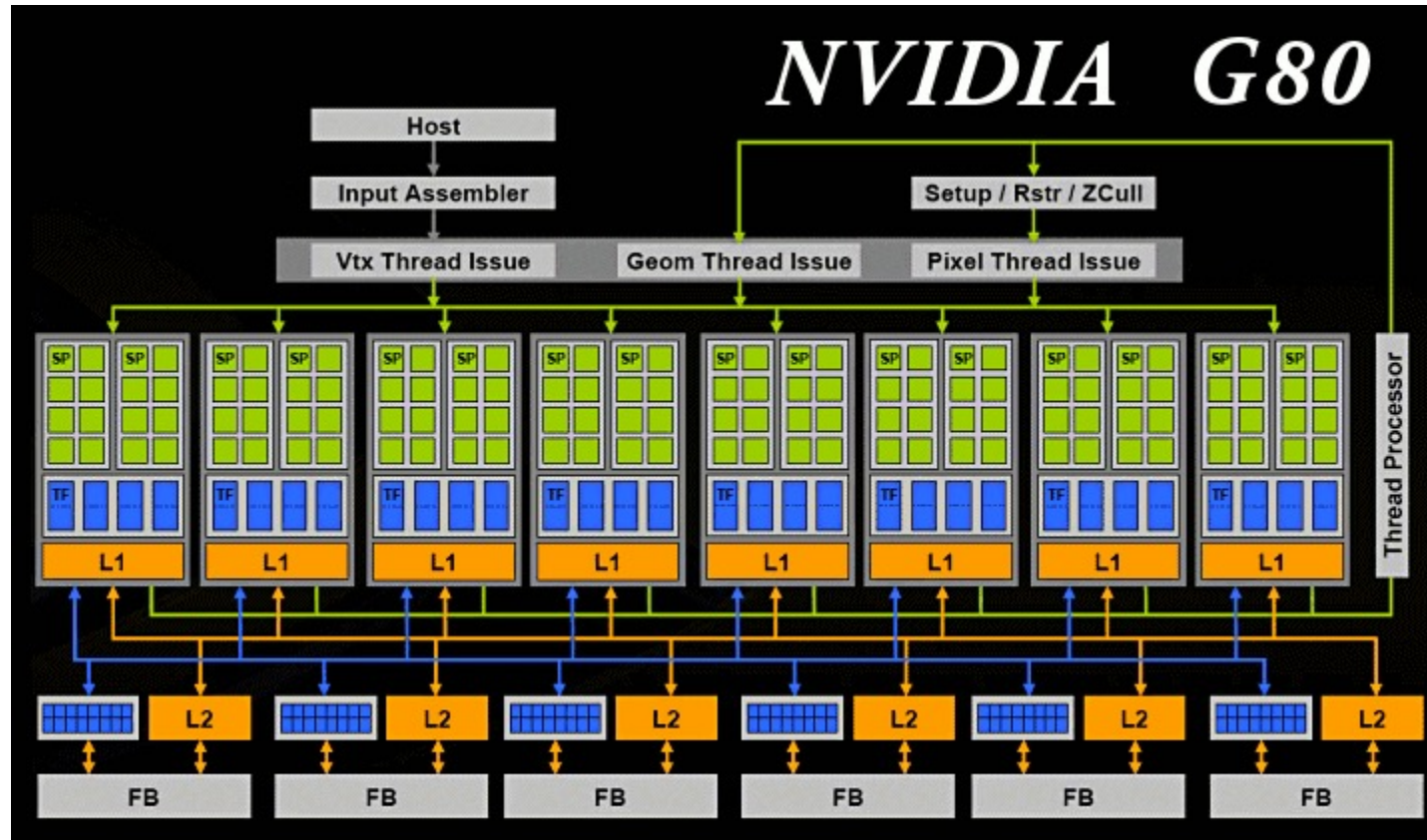


GeForce 6 series

Limited “programmability” of shaders:  
Minimal/no control flow  
Maximum instruction count

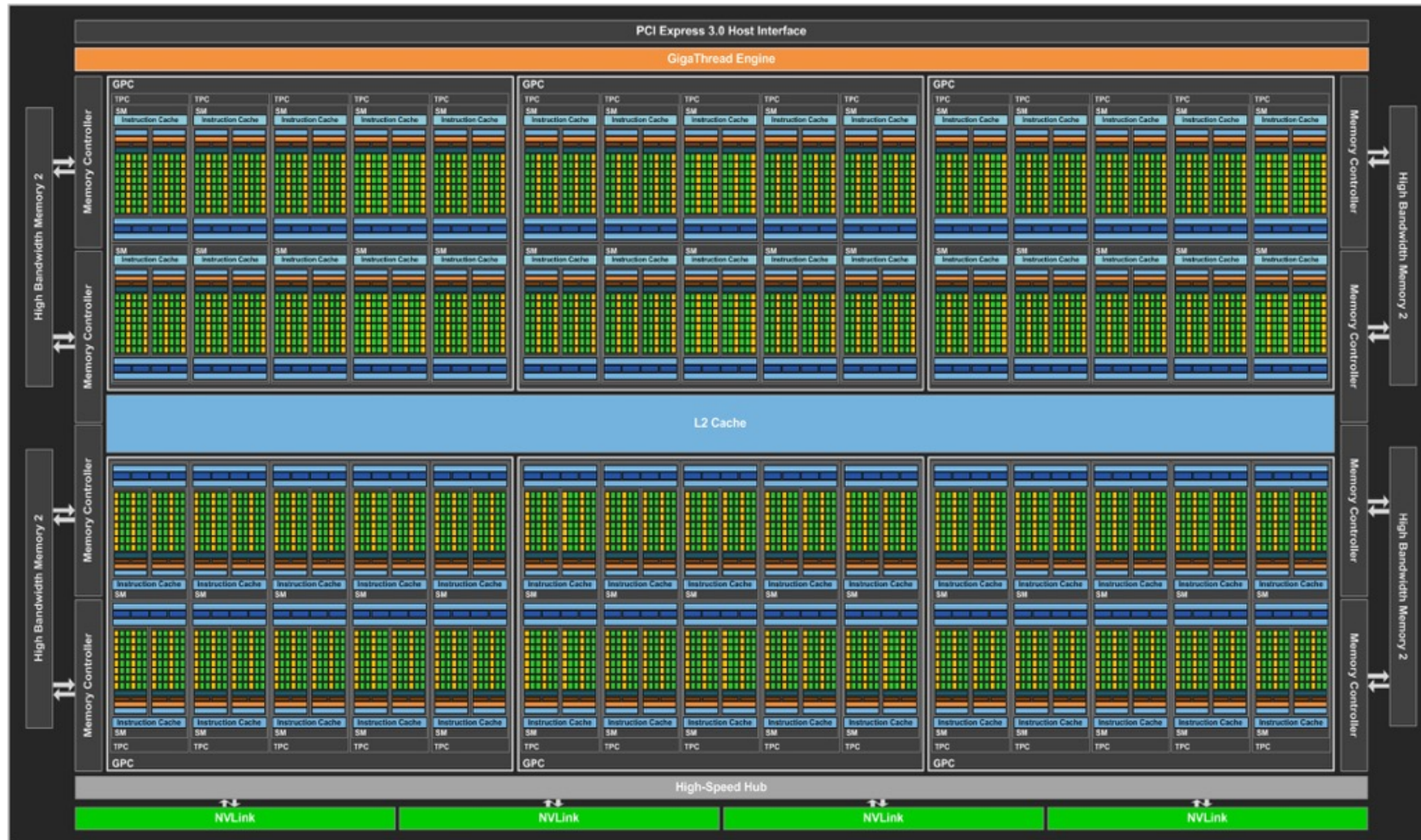


# Late Modernity: unified shaders

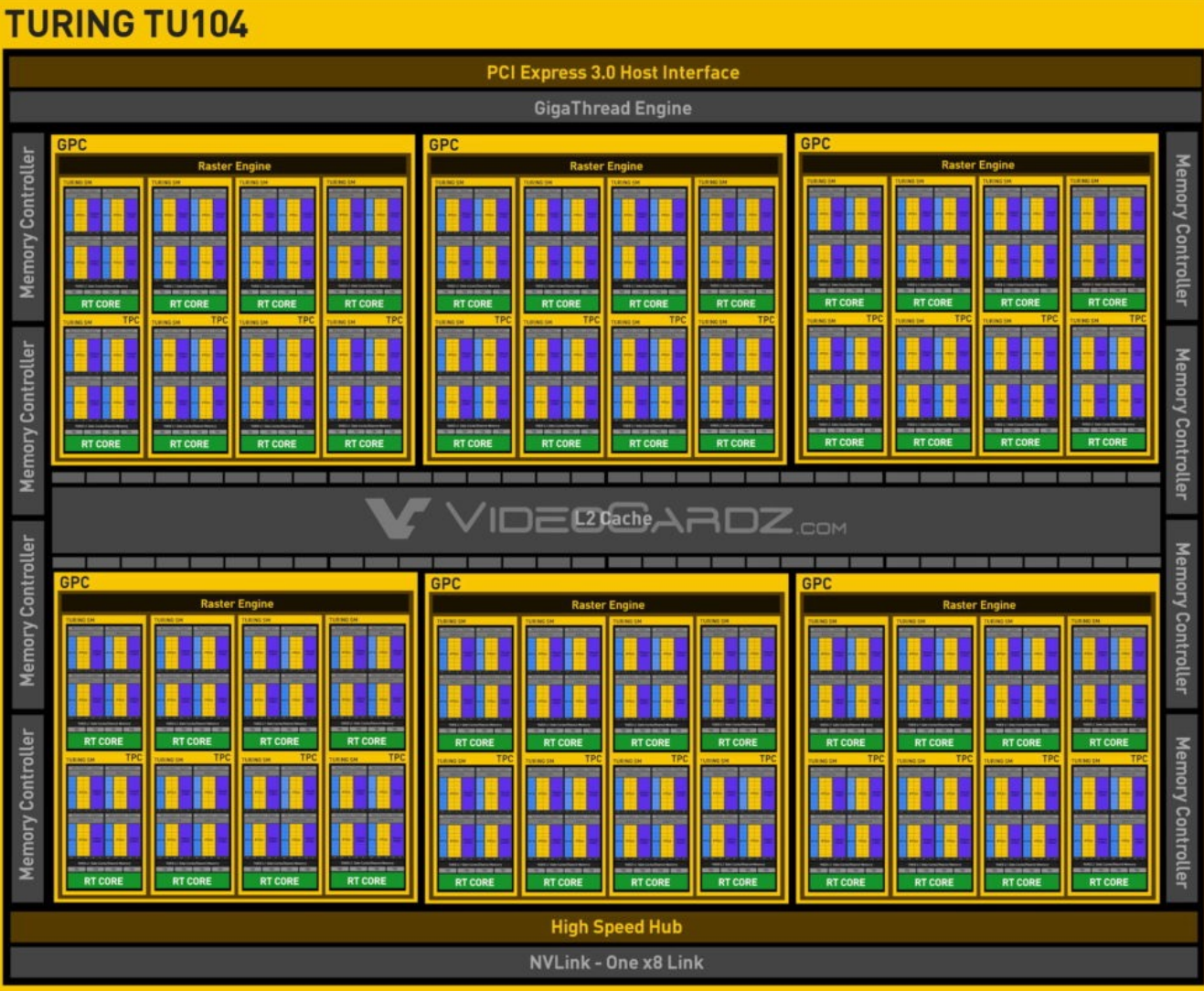
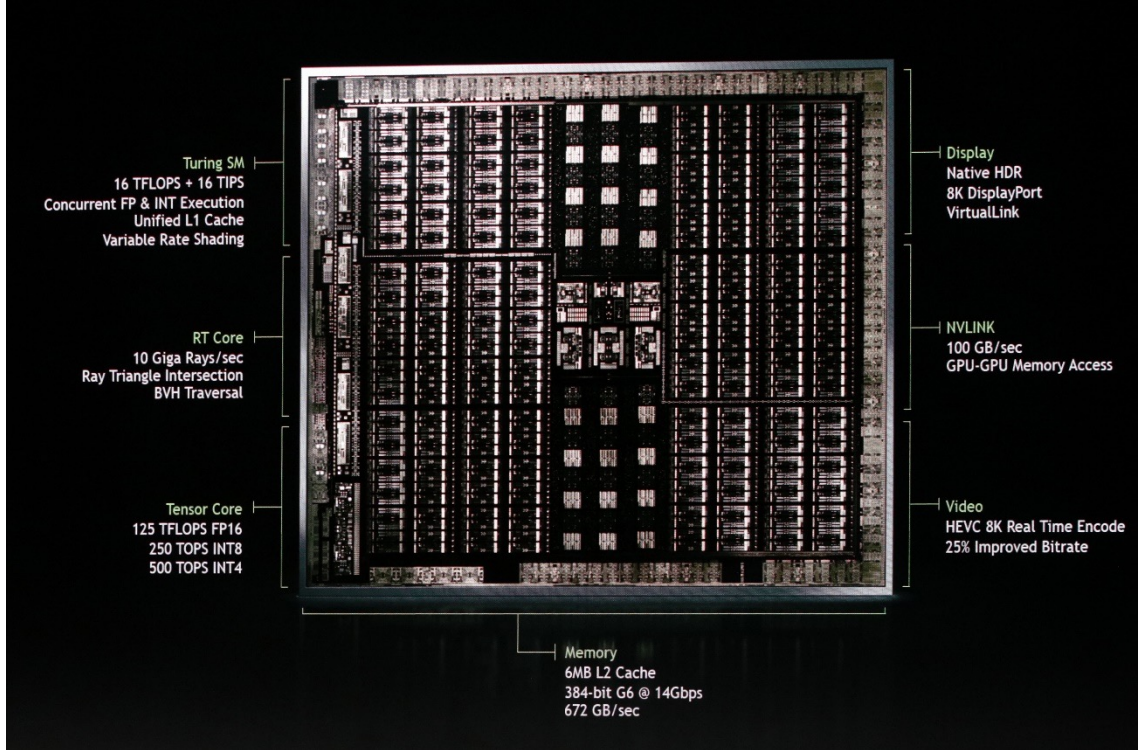


Mapping to Graphics pipeline no longer apparent  
Processing elements no longer specialized to a particular role  
Model supports *real* control flow, larger instr count

# Mostly Modern: Pascal



# Definitely Modern: Turing



# Modern Enough: Pascal SM



# Cross-generational observations

GPUs designed for parallelism in graphics pipeline:

- Data
  - Per-vertex
  - Per-fragment
  - Per-pixel
- Task
  - Vertex processing
  - Fragment processing
  - Rasterization
  - Hidden-surface elimination
- MLP
  - HW multi-threading for hiding memory latency

Even as GPU architectures become more general, certain assumptions persist:

1. Data parallelism is *trivially* exposed
2. All problems look like painting a box with colored dots

*But what if my problem isn't painting a box?!?!?*

# The big ideas still present in GPUs

- Simple cores
- Single instruction stream
  - Vector instructions (SIMD) OR
  - Implicit HW-managed sharing (SIMT)
- Hide memory latency with HW multi-threading

# Programming Model

- ***GPUs are I/O devices, managed by user-code***
- “kernels” == “shader programs”
- 1000s of HW-scheduled threads per kernel
- Threads grouped into independent blocks.
  - Threads in a block can synchronize (barrier)
  - This is the *\*only\** synchronization
- “Grid” == “launch” == “invocation” of a kernel
  - a group of blocks (or warps)

# Parallel Algorithms

- Sequential algorithms often do not permit easy parallelization
  - Does not mean there work has no parallelism
  - A different approach can yield parallelism
  - but often changes the algorithm
  - Parallelizing != just adding locks to a sequential algorithm
- Parallel Patterns
  - Map
  - Scatter, Gather
  - Reduction
  - Scan
  - Search, Sort

If you can express your algorithm using these patterns, an apparently fundamentally sequential algorithm can be made parallel



# Map

- Inputs
  - Array A
  - Function  $f(x)$
- $\text{map}(A, f) \rightarrow$  apply  $f(x)$  on all elements in A
- Parallelism trivially exposed
  - $f(x)$  can be applied in parallel to all elements, in principle

```
for(i=0; i<numPoints; i++) {  
    labels[i] = findNearestCenter(points[i]);  
}
```



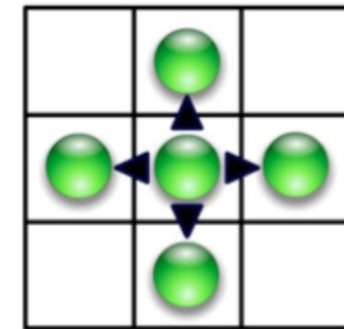
```
map(points, findNearestCenter)
```

# Scatter and Gather

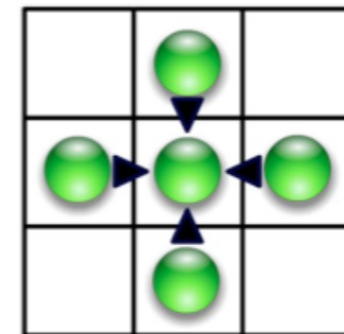
- Gather:
  - Read multiple items to single location
- Scatter:
  - Write single data item to multiple locations

```
for (i=0; i<N; ++i)  
x[i] = y[idx[i]];      gather(x, y, idx)
```

```
for (i=0; i<N; ++i)  
y[idx[i]] = x[i];      scatter(x, y, idx)
```



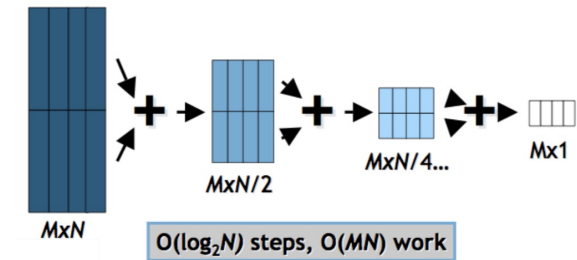
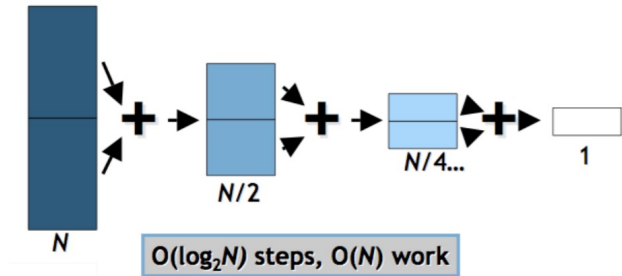
Scatter



Gather

# Reduce

- Input
  - Associative operator **op**
  - Ordered set  $s = [a, b, c, \dots z]$
- $\text{Reduce}(\text{op}, s)$  returns  $a \text{ op } b \text{ op } c \dots \text{ op } z$



```
for(i=0; i<N; ++i) {  
    accum += (map(sqr, point[i]))  
}
```

$\text{accum} = \text{reduce}(+, \text{map}(\text{sqr}, \text{point}))$

Why must op be associative?

# Scan (prefix sum)

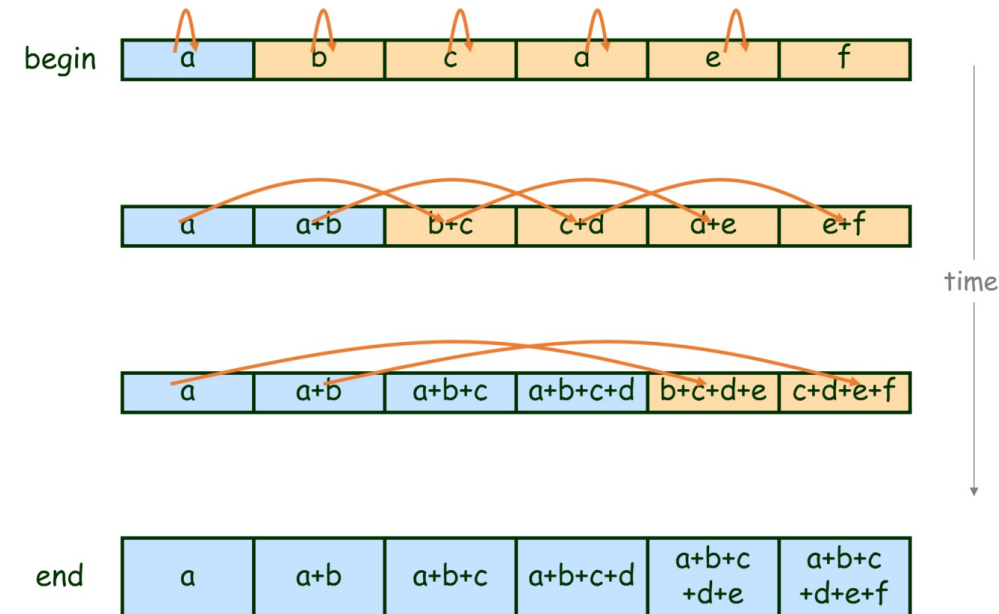
- Input

- Associative operator **op**
- Ordered set  $s = [a, b, c, \dots z]$
- Identity  $I$

- $\text{scan}(\text{op}, s) = [I, a, (a \text{ op } b), (a \text{ op } b \text{ op } c) \dots]$

- Scan is the workhorse of parallel algorithms:

- Sort, histograms, sparse matrix, string compare, ...



# Summary

- Re-expressing apparently sequential algorithms as combinations of parallel patterns is a common technique when targeting GPUs