Parallel Architectures
Parallel Algorithms
CUDA

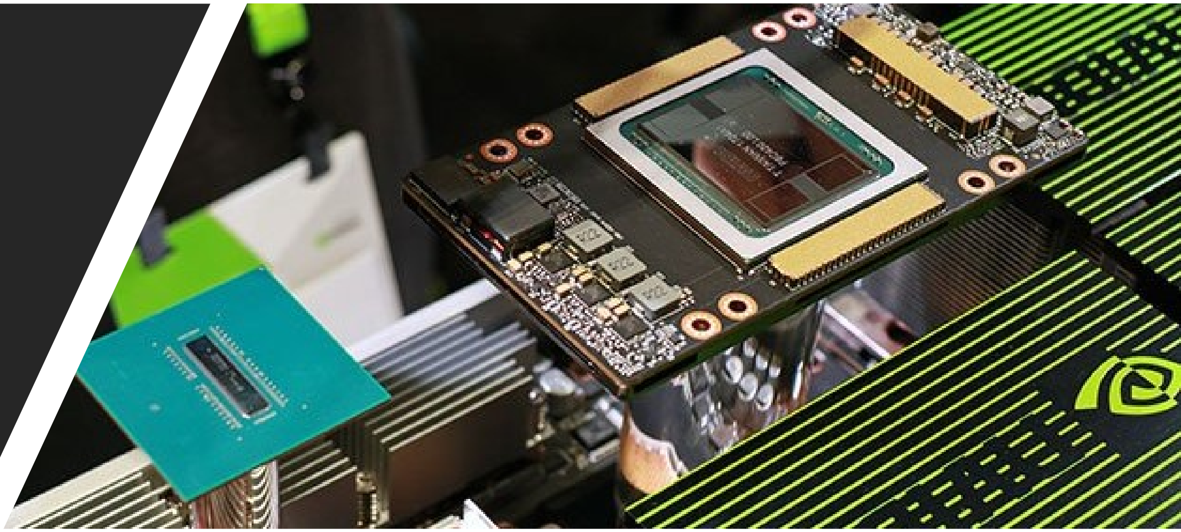
Chris Rossbach

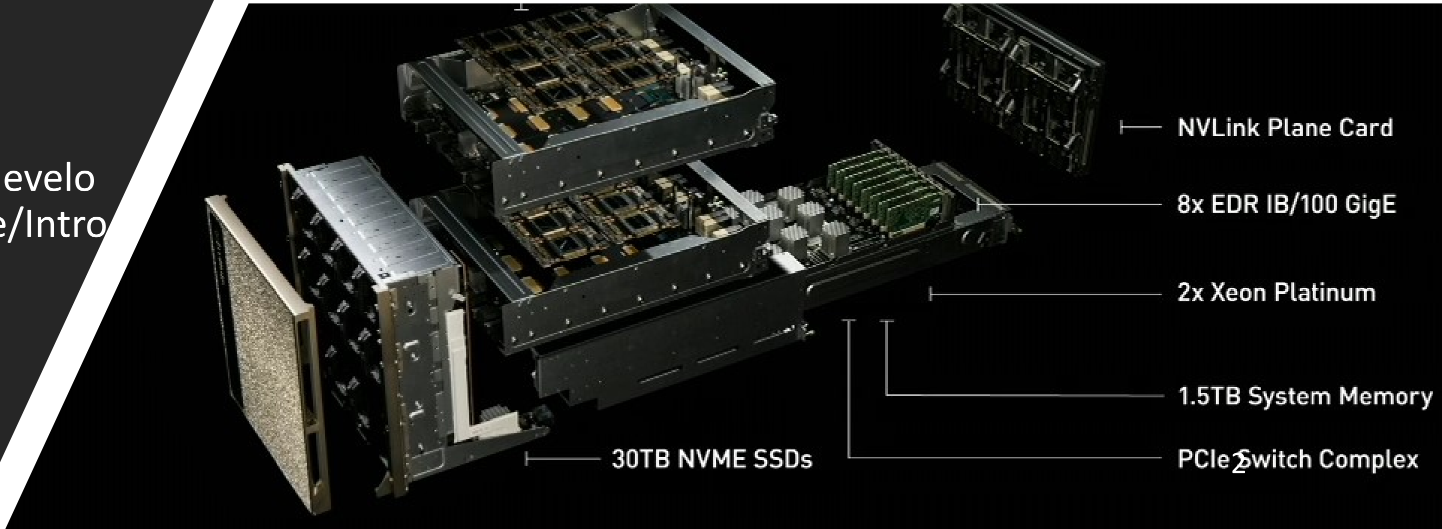
cs378h

Outline for Today

- Questions?
- Administrivia
  - pedagogical-* machines should be available
- Agenda
  - Parallel Algorithms
  - CUDA

- Acknowledgements: http://developer.download.nvidia.com/compute/developertrainingmaterials/presentations/cuda_language/Introduction_to_CUDA_C.pptx

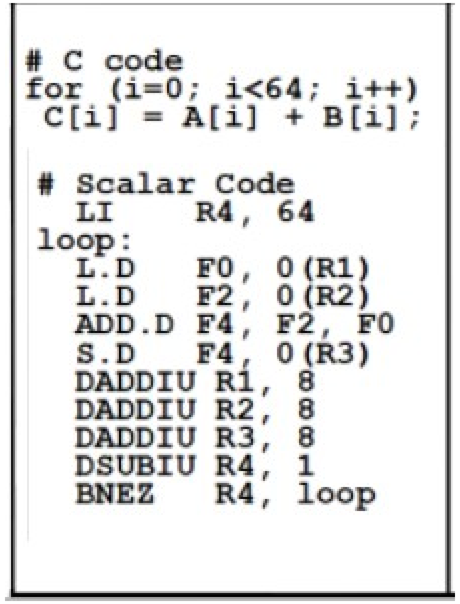




Faux Quiz Questions

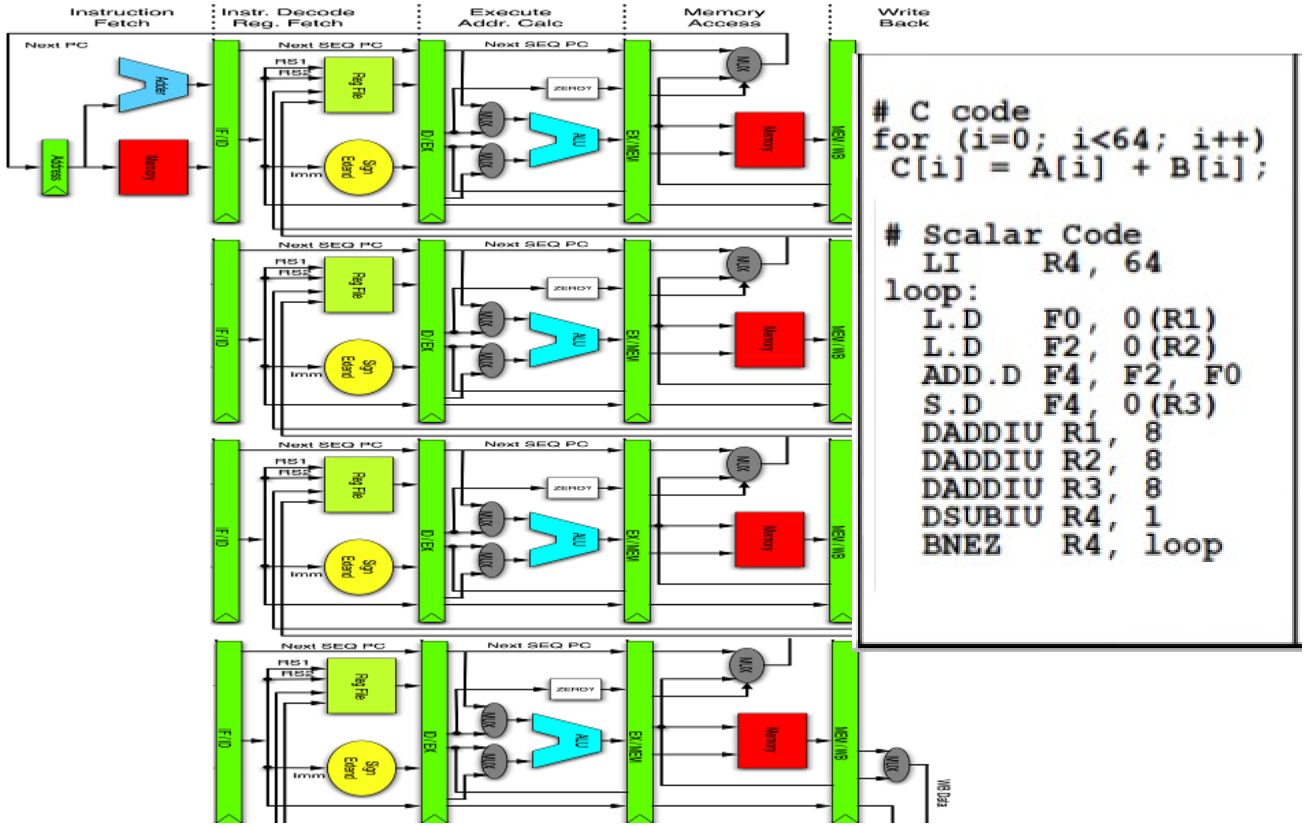
- What is a reduction? A prefix sum? Why are they hard to parallelize and what basic techniques can be used to parallelize them?
- Define flow dependence, output dependence, and anti-dependence: give an example of each. Why/how do compilers use them to detect loop-independent vs loop-carried dependences?
- What is the difference between a thread-block and a warp?
- How/Why must programmers copy data back and forth to a GPU?
- What is “shared memory” in CUDA? Describe a setting in which it might be useful.
- CUDA kernels have implicit barrier synchronization. Why is __syncthreads() necessary in light of this fact?
- How might one implement locks on a GPU?
- What ordering guarantees does a GPU provide across different hardware threads’ access to a single memory location? To two disjoint locations?
- When is it safe for one GPU thread to wait (e.g. by spinning) for another?

Review: what is a vector processor?



```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];

 # 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
```

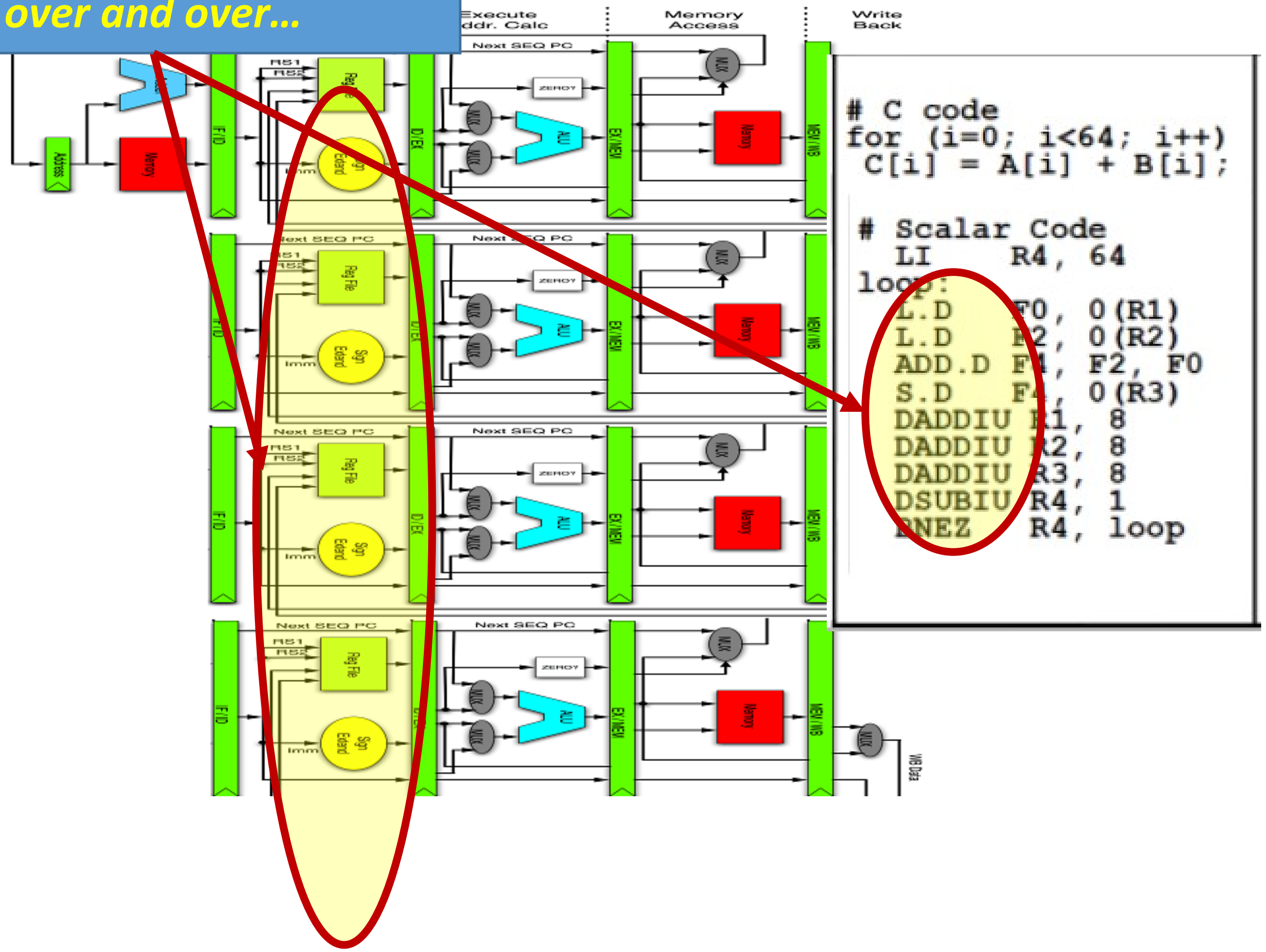
Review: what is a vector processor?



```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];

 # 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
```

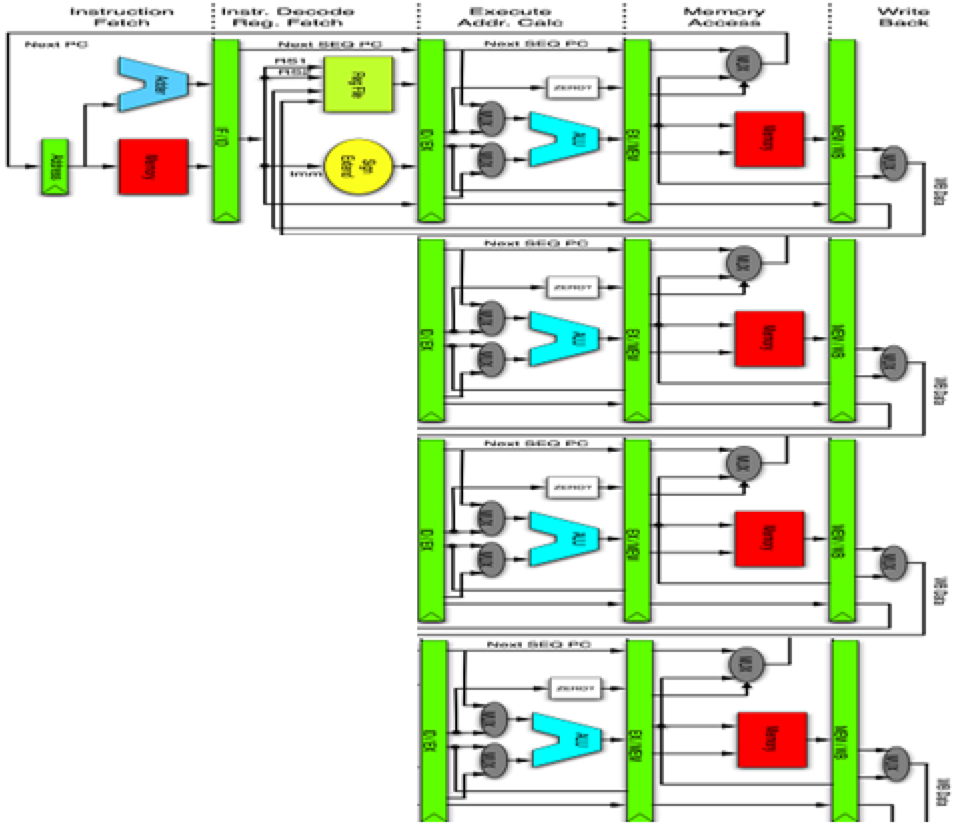
Review: what is a vector processor?

Dont decode same instruction over and over…

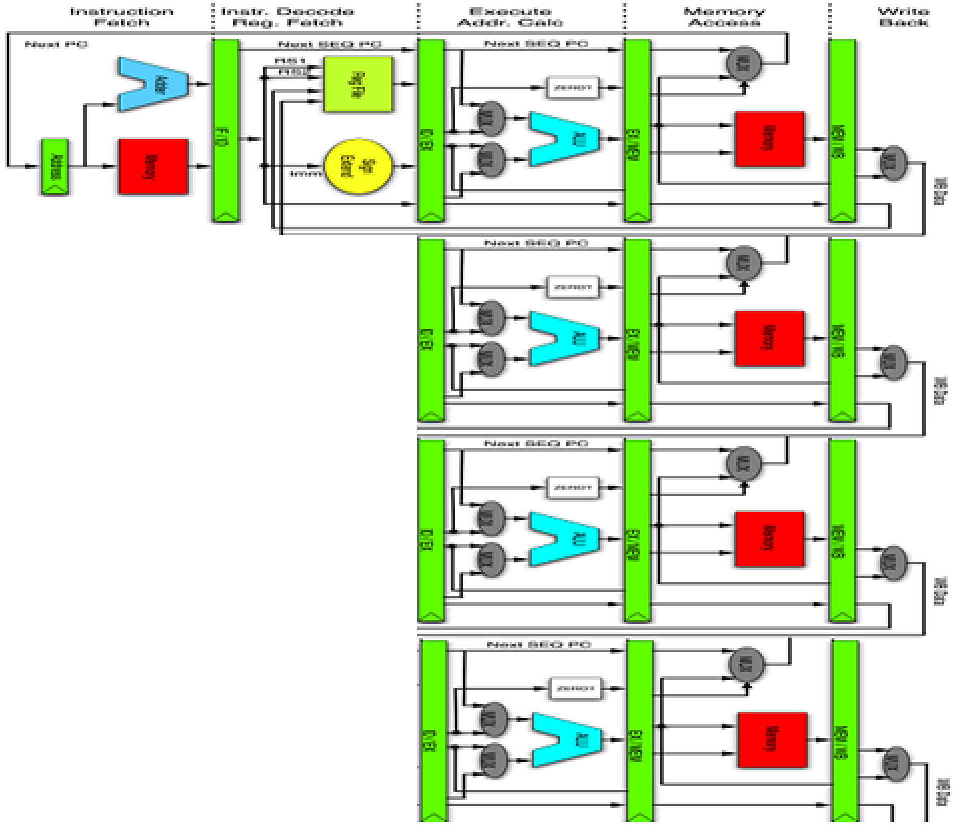


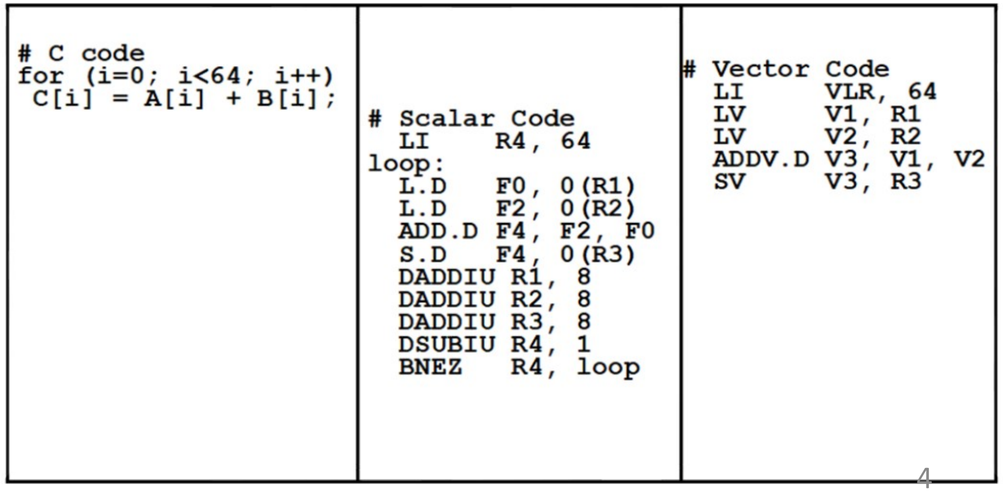
```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];

 # 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
```

Review: what is a vector processor?



Review: what is a vector processor?



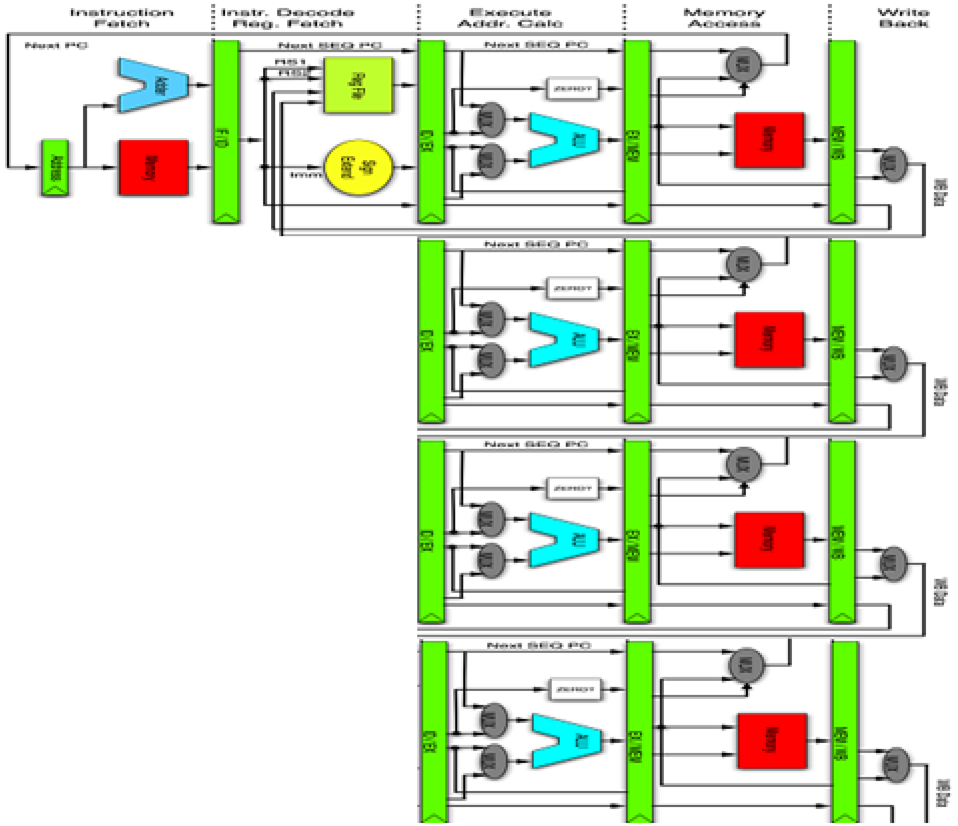


```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];
```

```
# 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
```

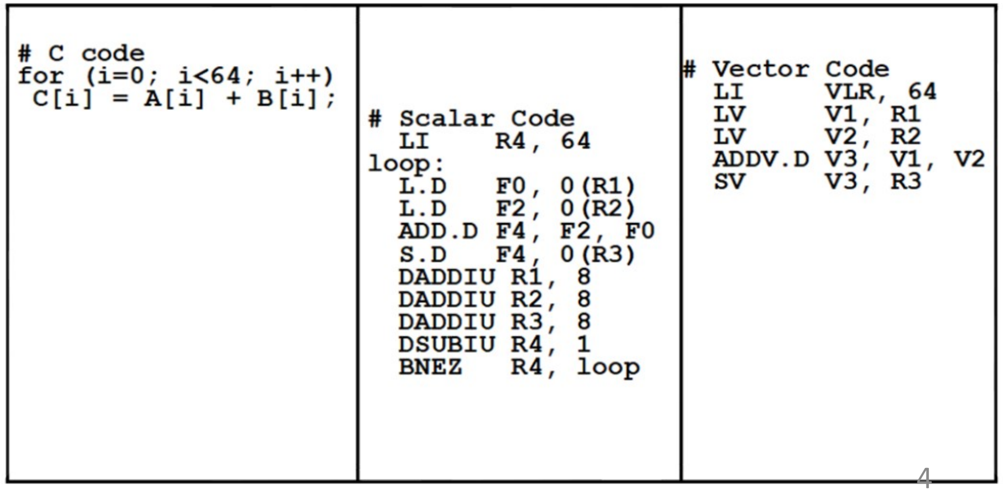
```
# Vector Code
  LI     VLR, 64
  LV     V1, R1
  LV     V2, R2
  ADDV.D V3, V1, V2
  SV     V3, R3
```

Review: what is a vector processor?



Implementation:

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

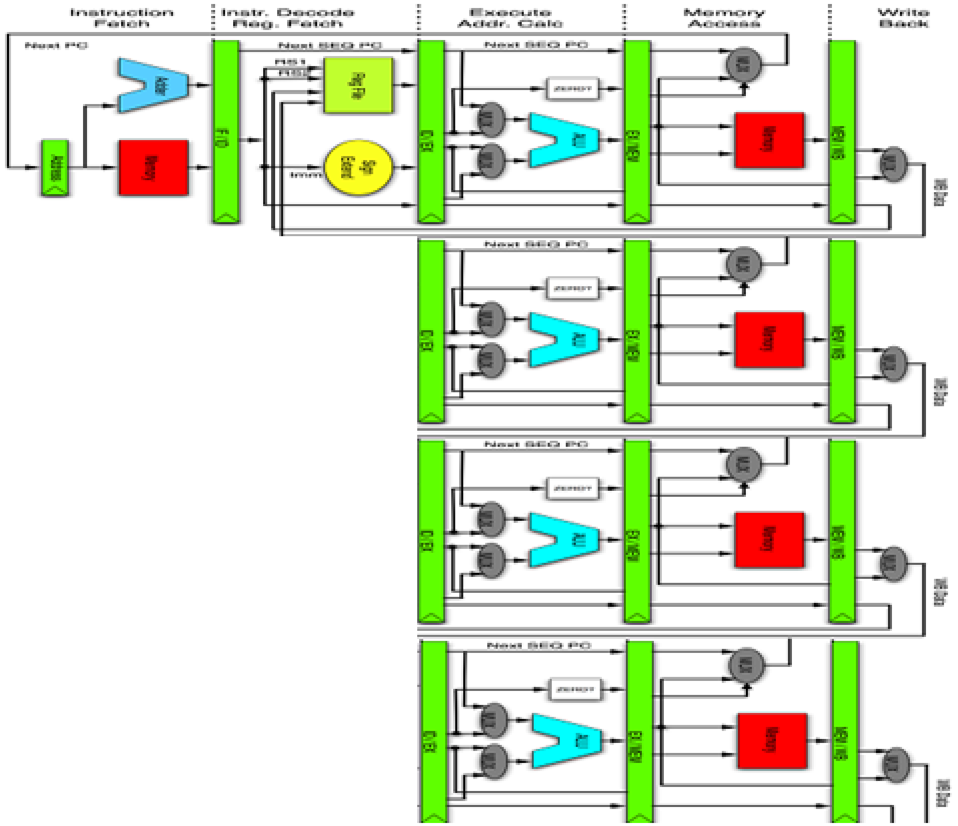


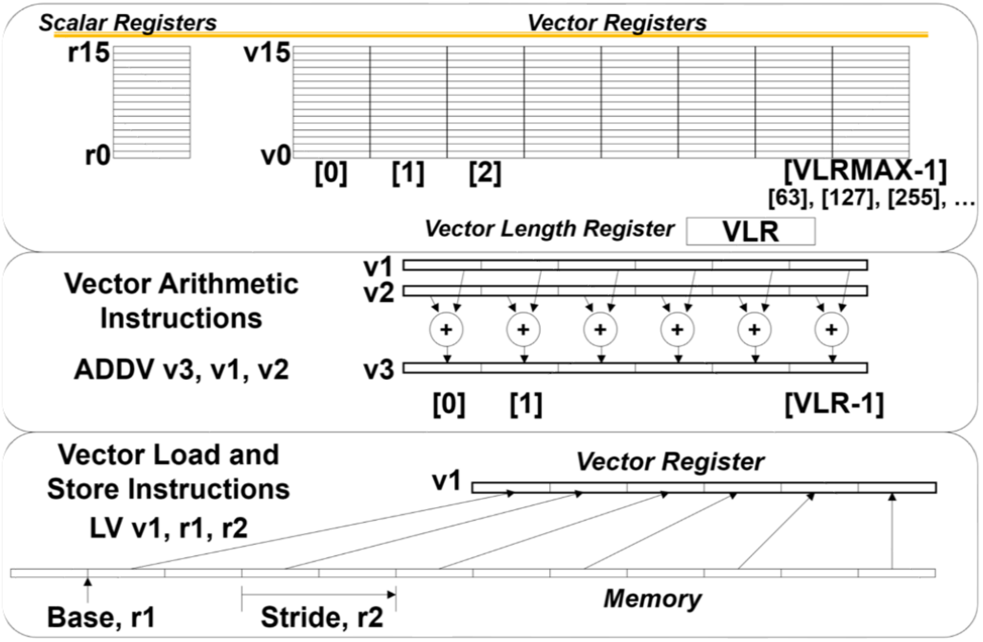
```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];
```

```
# 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
```

```
# Vector Code
  LI     VLR, 64
  LV     V1, R1
  LV     V2, R2
  ADDV.D V3, V1, V2
  SV     V3, R3
```

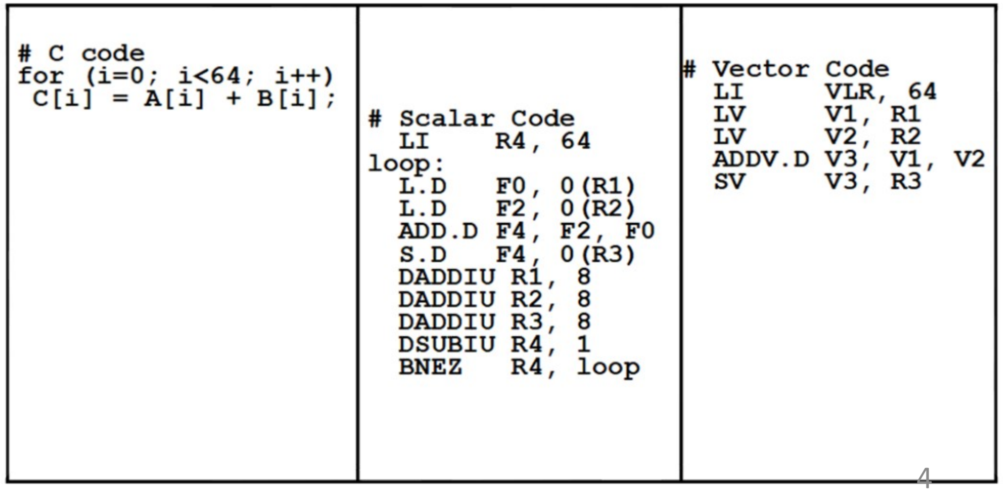
Review: what is a vector proc





Imp

- Ir
- Sa
- M
- Multiple different operands in parallel

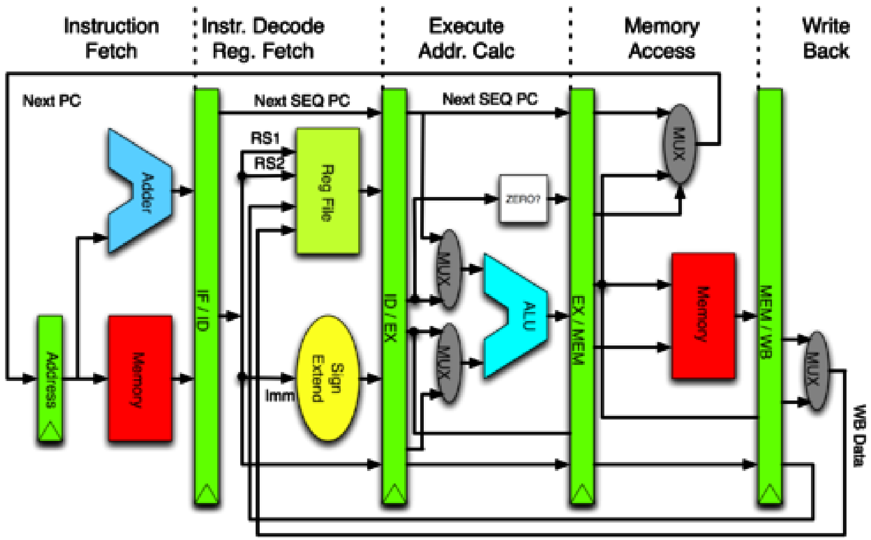


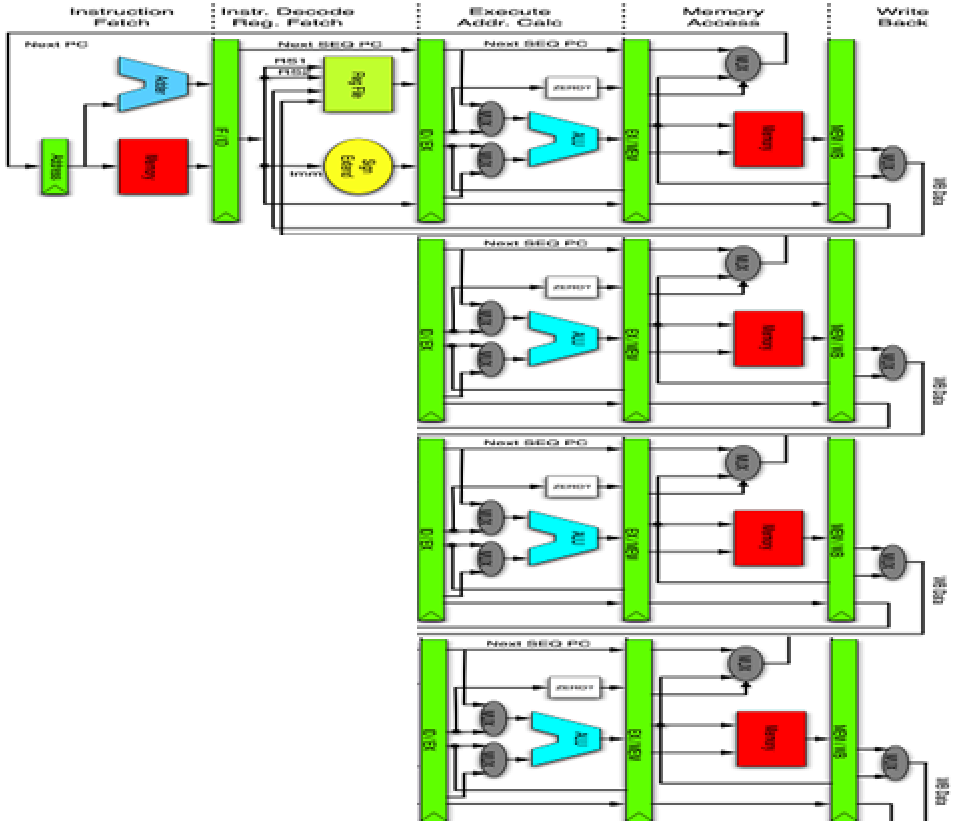
```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];
```

```
# 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
```

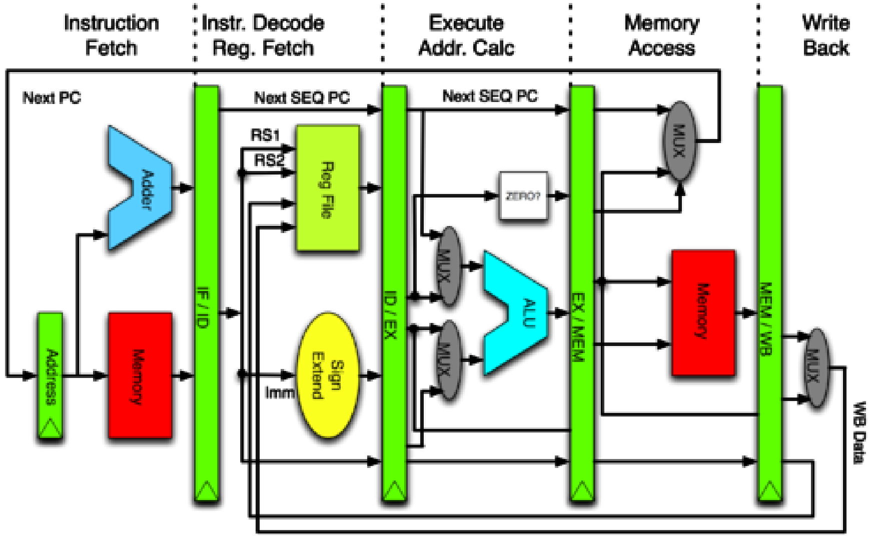
```
# Vector Code
  LI     VLR, 64
  LV     V1, R1
  LV     V2, R2
  ADDV.D V3, V1, V2
  SV     V3, R3
```

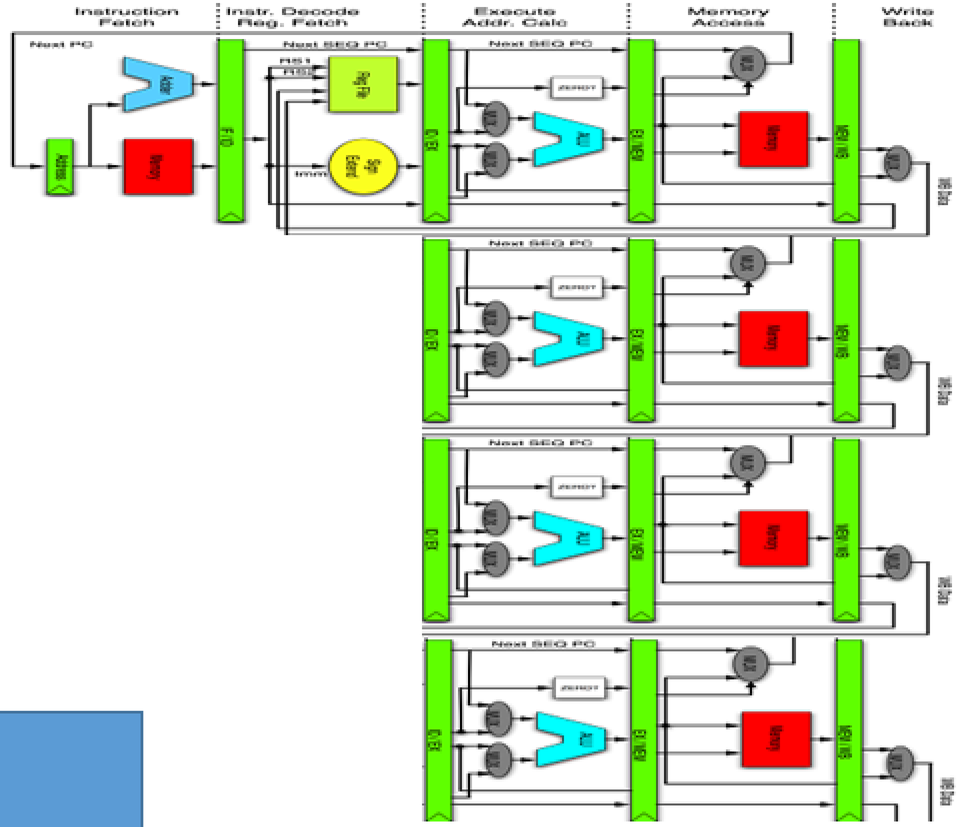
When does vector processing help?





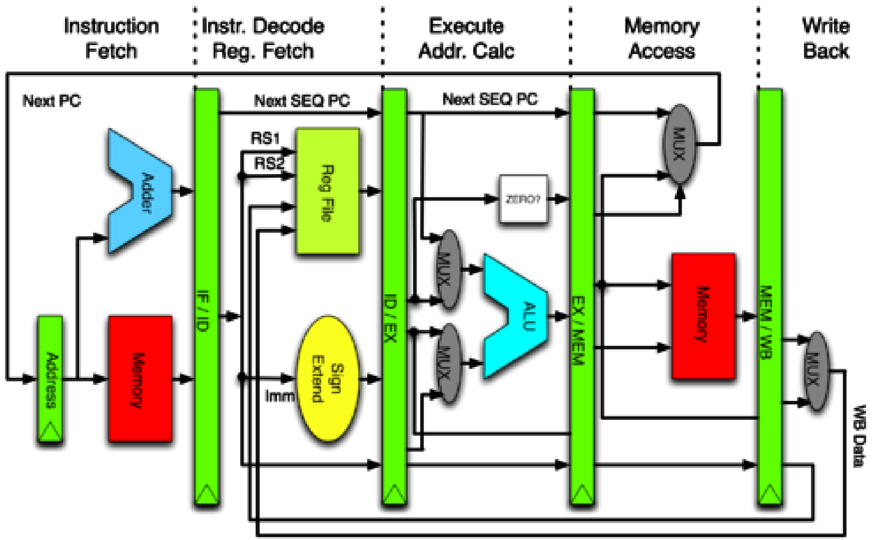
When does vector processing help?

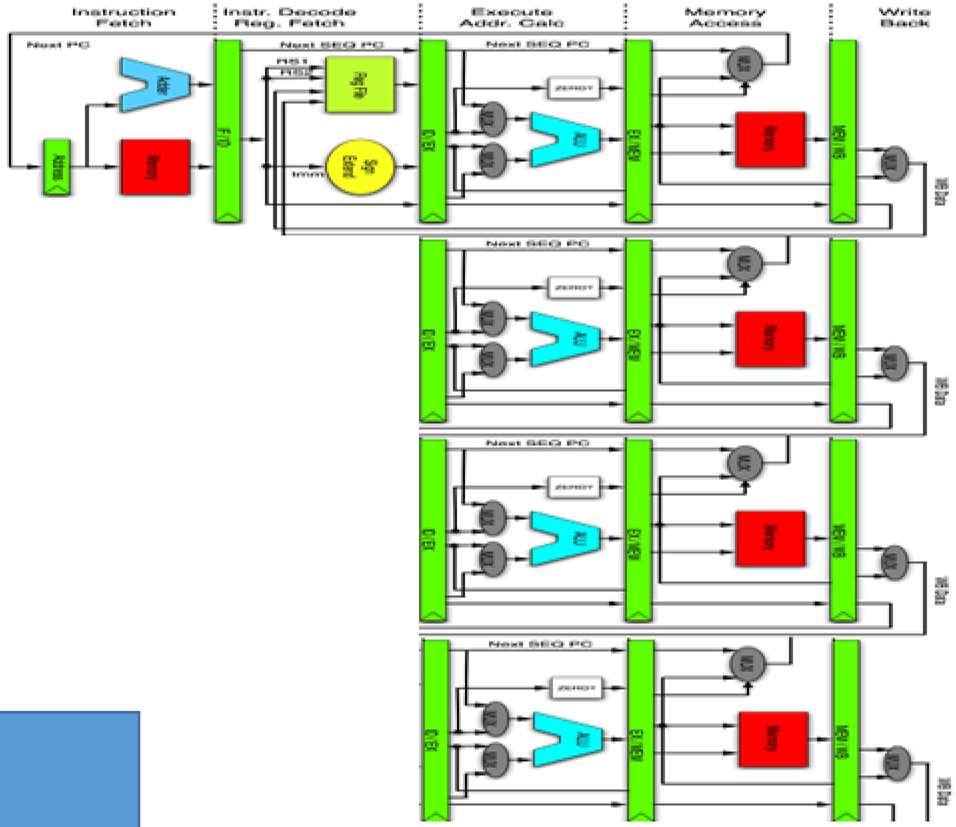




What are the potential bottlenecks here?
When can it improve throughput?

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

Hardware multi-threading

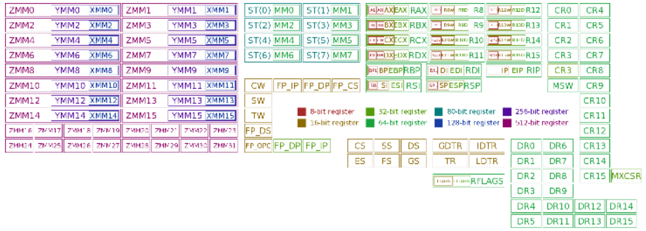
- Address memory bottleneck

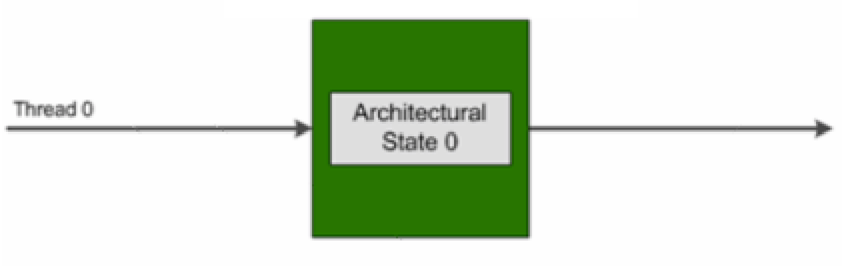
Hardware multi-threading

- Address memory bottleneck
- Share exec unit across
    - Instruction streams
    - Switch on stalls

Hardware multi-threading

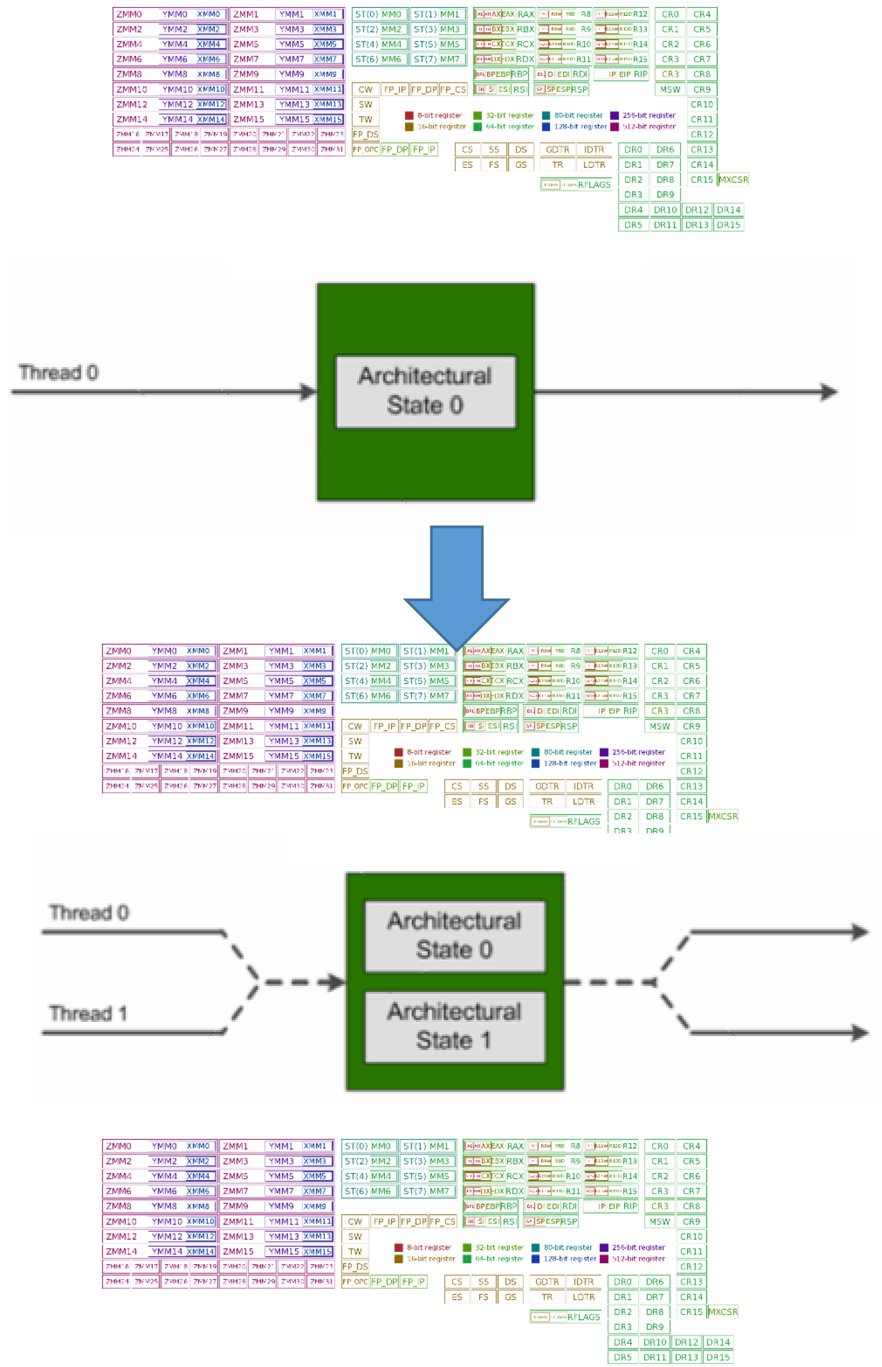
- Address memory bottleneck
- Share exec unit across
  - Instruction streams
  - Switch on stalls





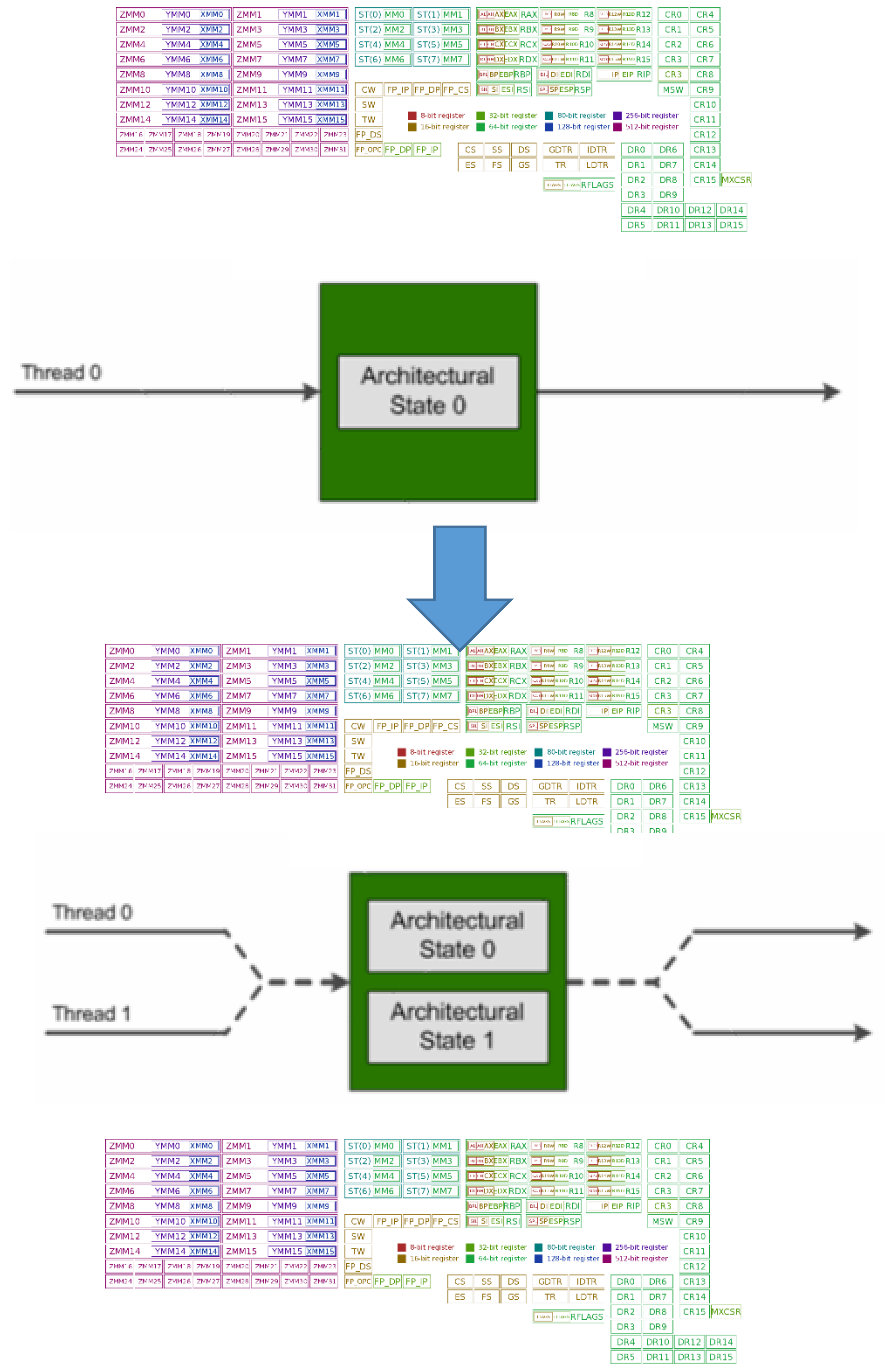
Hardware multi-threading

- Address memory bottleneck
- Share exec unit across
  - Instruction streams
  - Switch on stalls



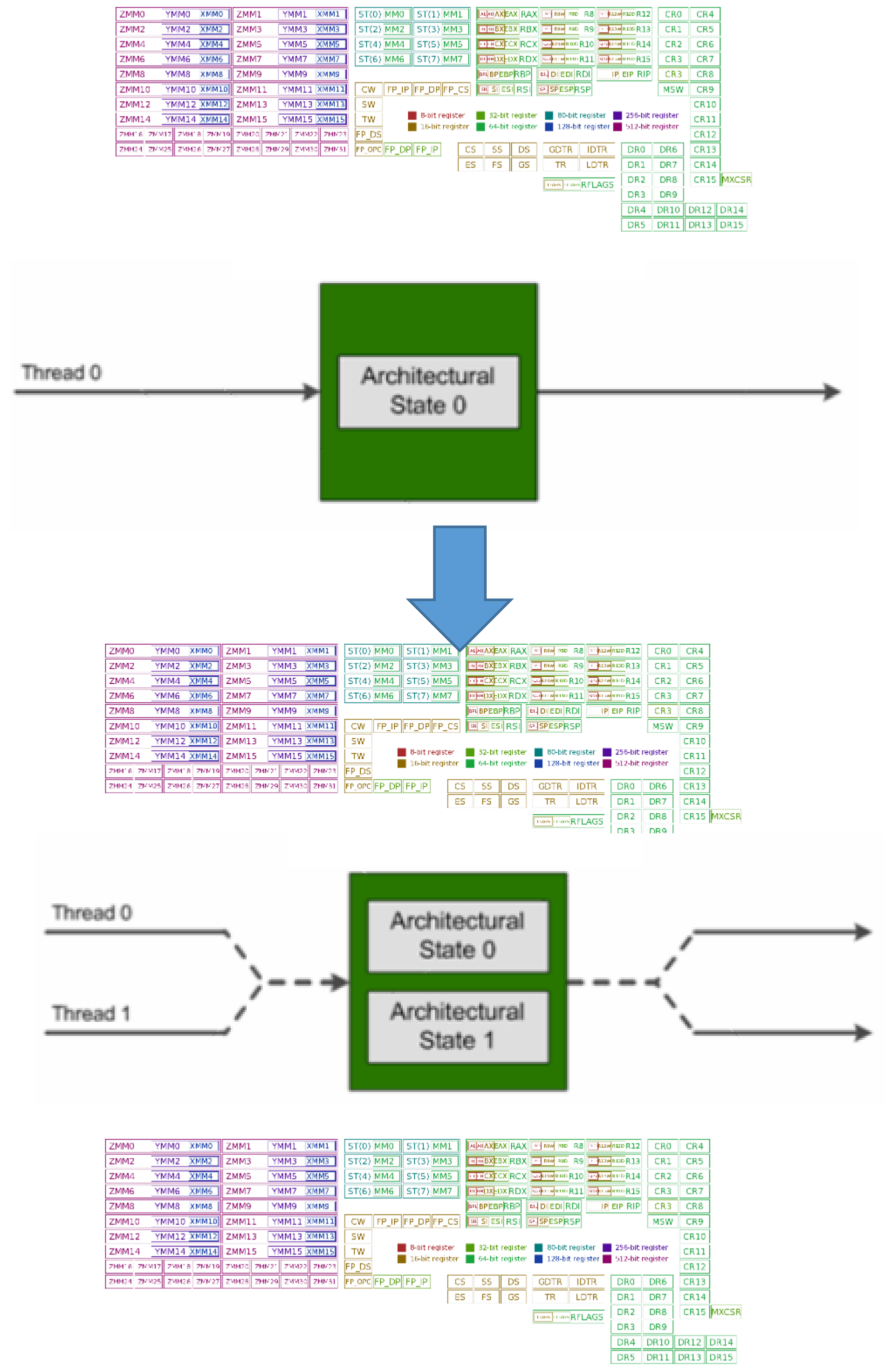
Hardware multi-threading

- Address memory bottleneck
- Share exec unit across
  - Instruction streams
  - Switch on stalls
- Looks like multiple cores to the OS

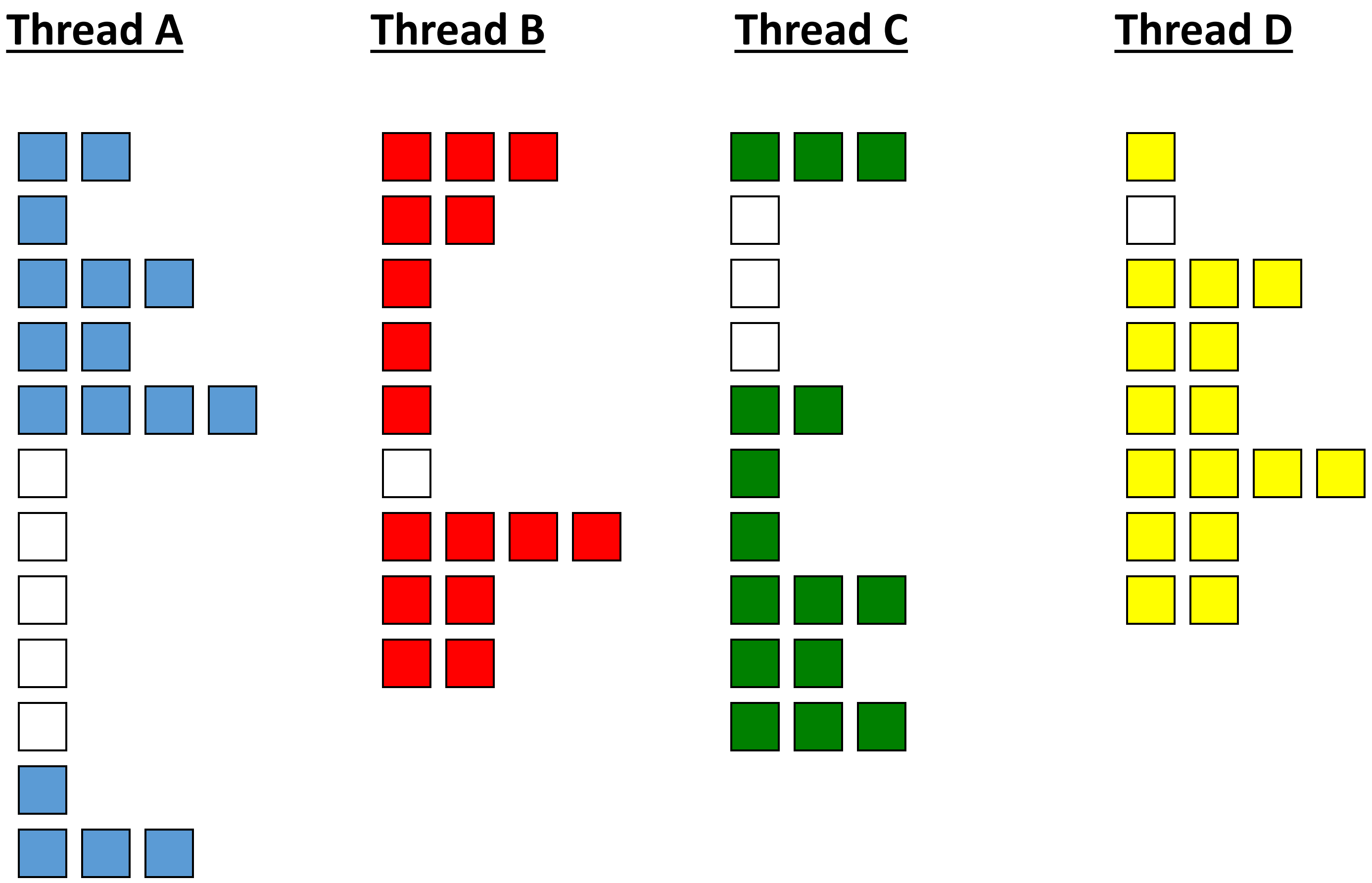


Hardware multi-threading

- 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



- Colors → pipeline full
- White → stall

Coarse- grained multithreading

Coarse- grained multithreading

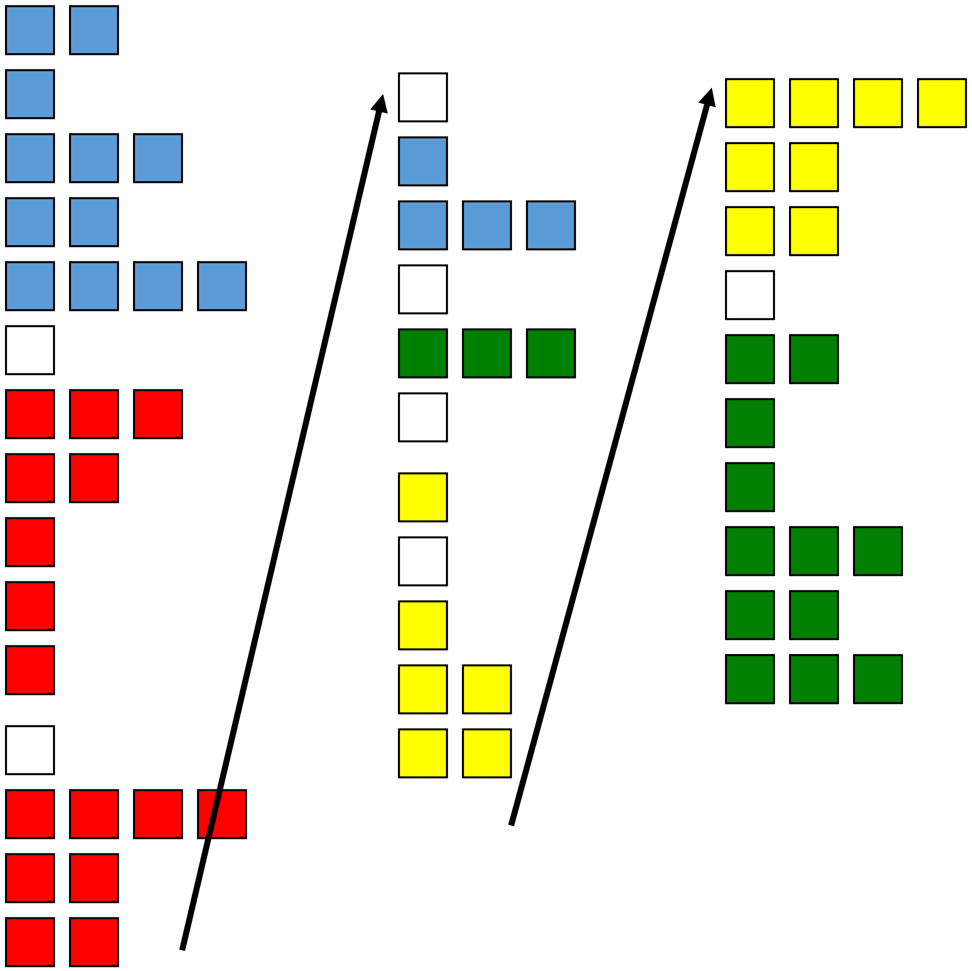
- Single thread runs until a costly stall
    - E.g. 2nd level cache miss

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!

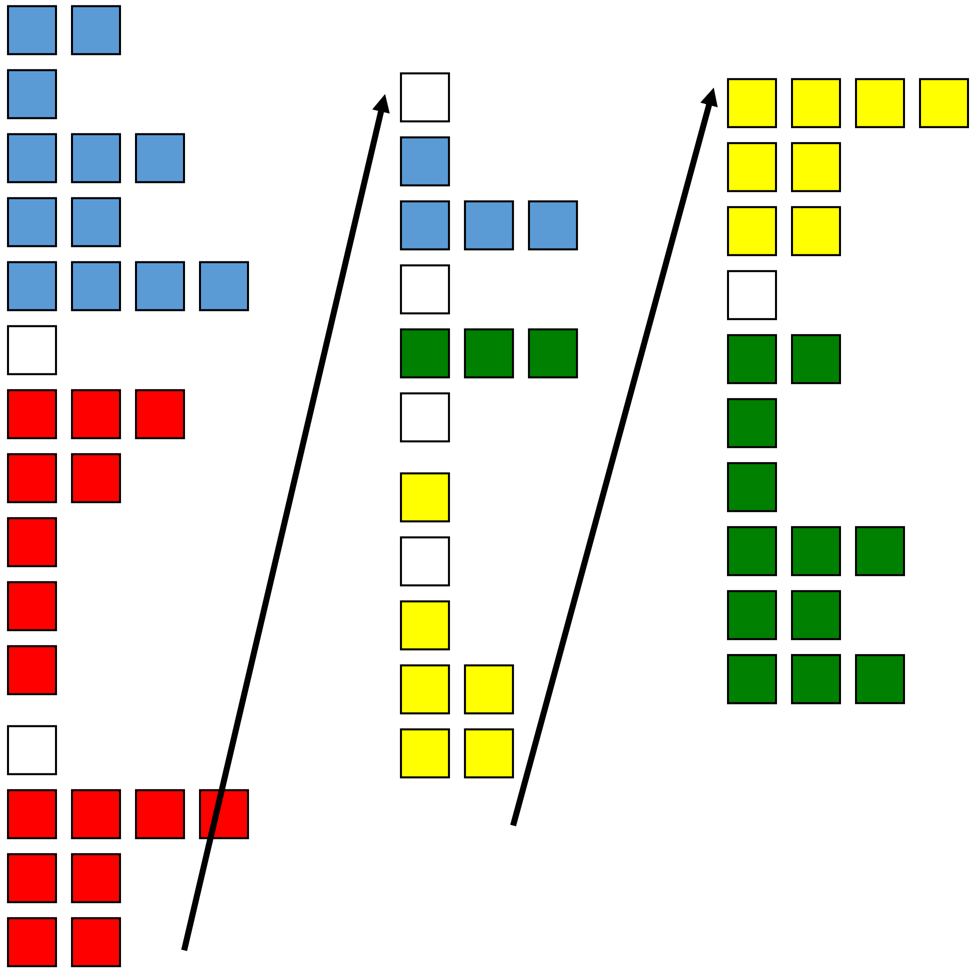
Coarse- grained multithreading

- Single thread runs until a costly stall
  - E.g. 2nd level cache miss
- Another thread starts during stall
  - Pipeline fill time requires several cycles!



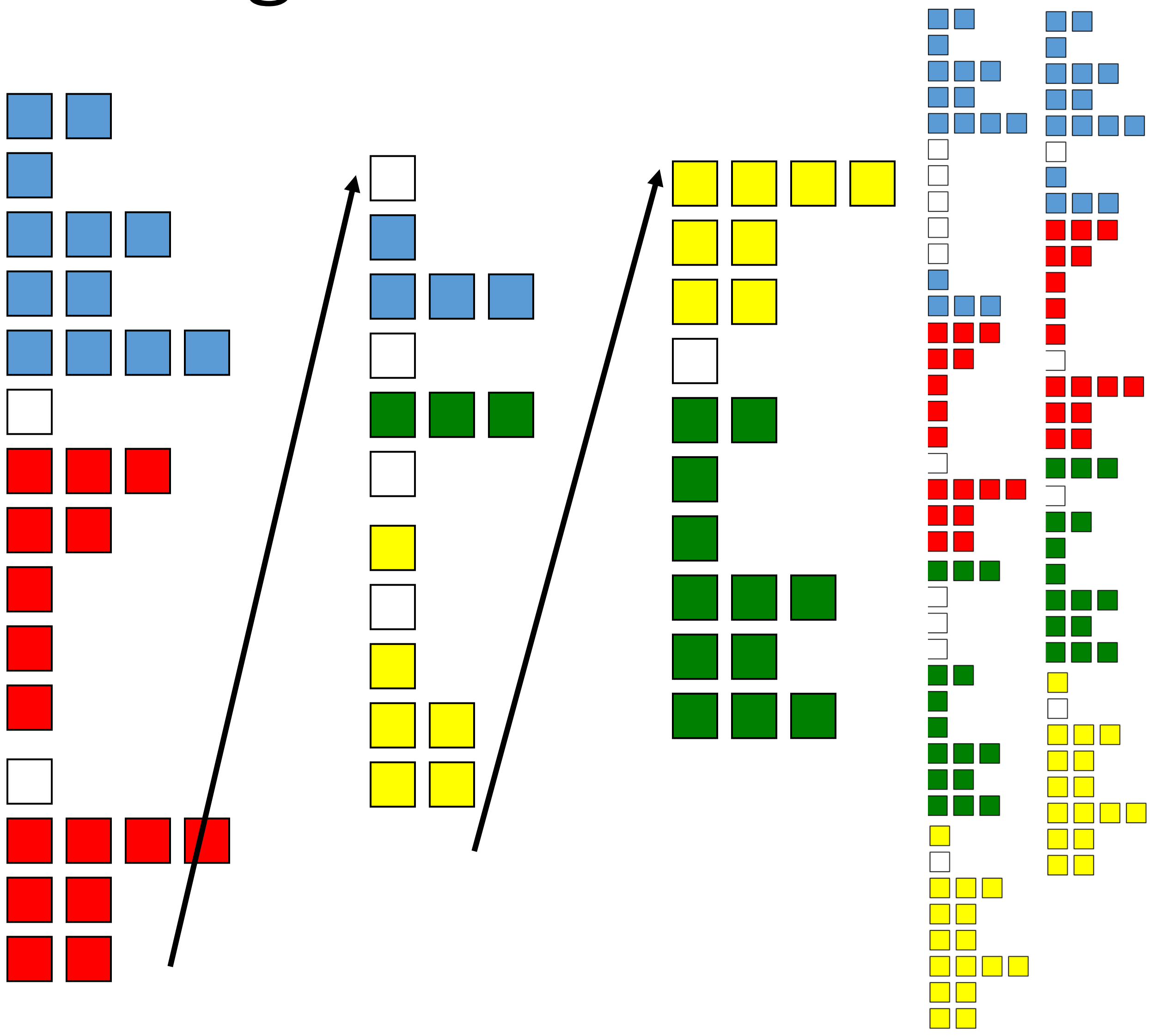
Coarse- grained multithreading

- Single thread runs until a costly stall
  - E.g. 2nd level cache miss
- Another thread starts during stall
  - Pipeline fill time requires several cycles!
- Hardware support required
  - PC and register file for each thread
  - Looks like another physical CPU to OS/software



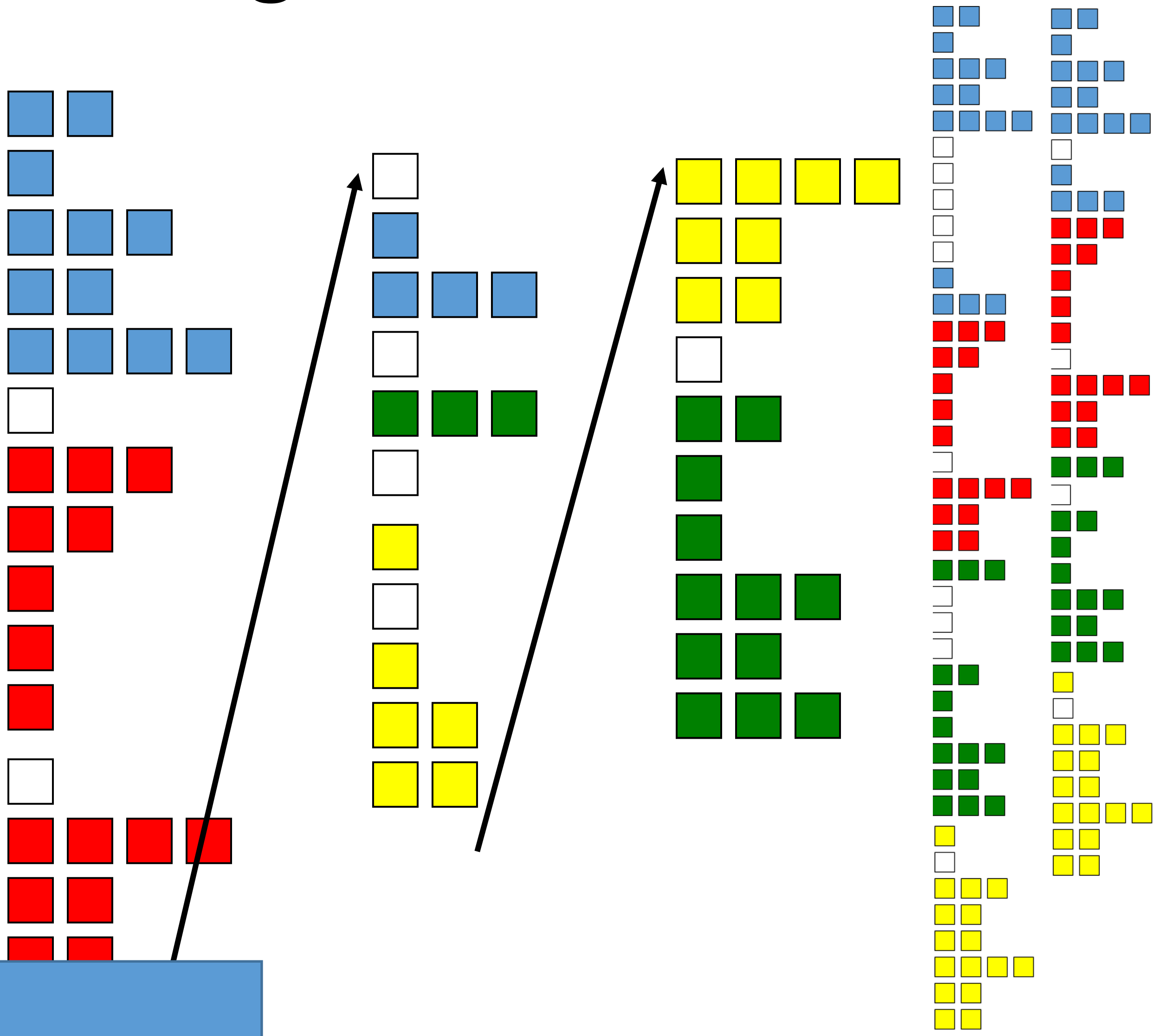
Coarse- grained multithreading

- Single thread runs until a costly stall
  - E.g. 2nd level cache miss
- Another thread starts during stall
  - Pipeline fill time requires several cycles!
- Hardware support required
  - PC and register file for each thread
  - Looks like another physical CPU to OS/software



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!
- Hardware support required
  - PC and register file for each thread
  - Looks like another physical CPU to OS/software



Pros? Cons?

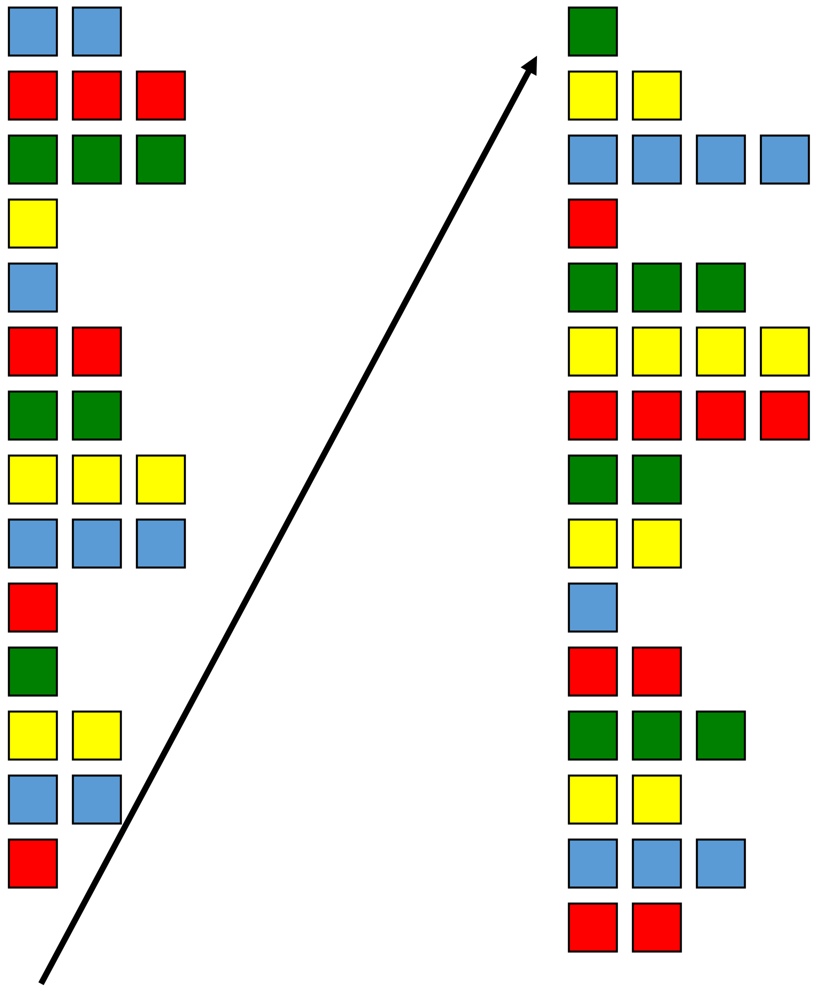
Fine-grained multithreading

Fine-grained multithreading

- Threads interleave instructions
    - Round-robin
    - Skip stalled threads

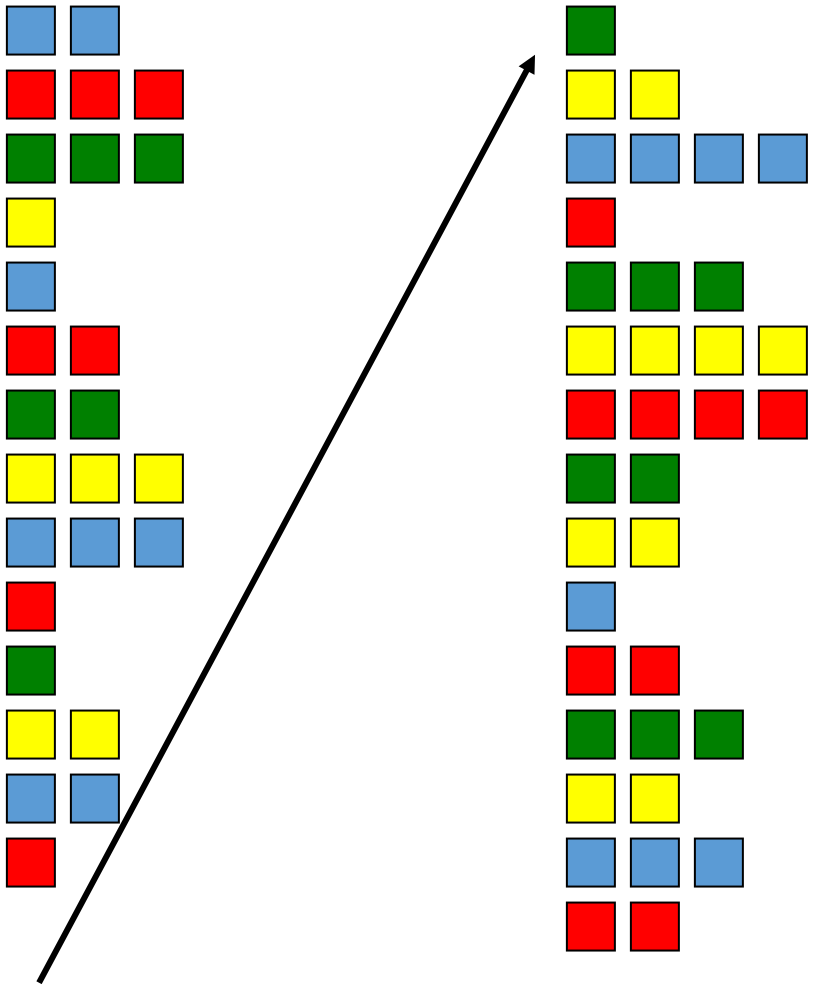
Fine-grained multithreading

- Threads interleave instructions
  - Round-robin
  - Skip stalled threads



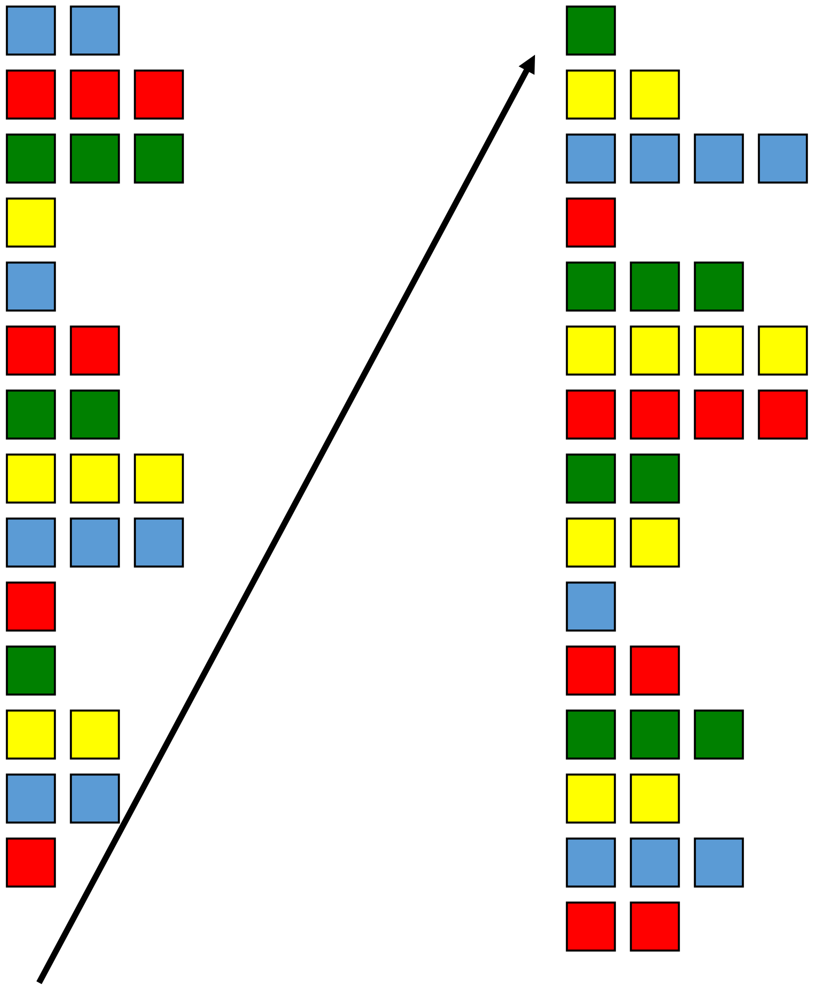
Fine-grained multithreading

- Threads interleave instructions
  - Round-robin
  - Skip stalled threads
- Hardware support required
  - Separate PC and register file per thread
  - Hardware to control alternating pattern



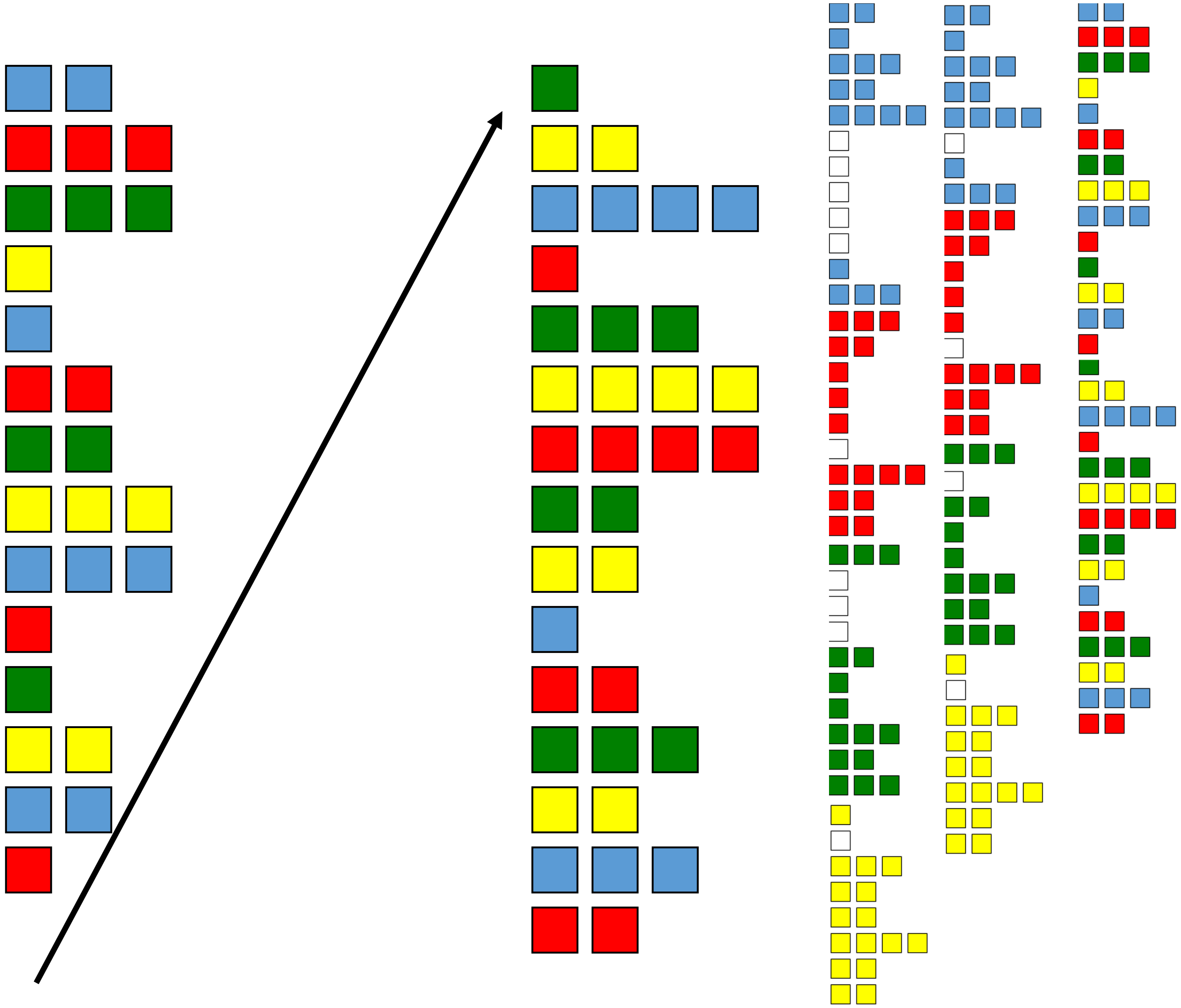
Fine-grained multithreading

- 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



Fine-grained multithreading

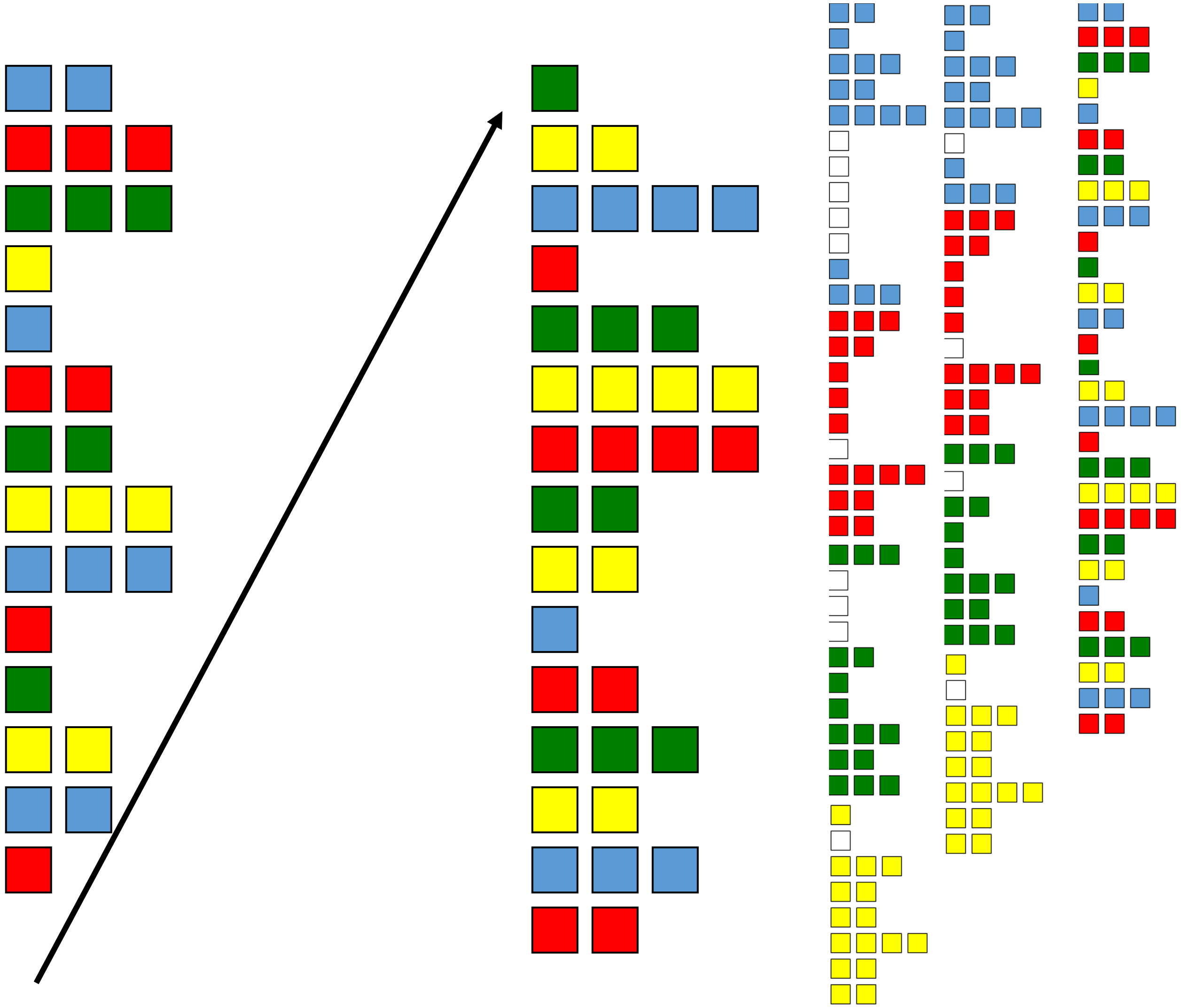
- Threads interleave instructions
  - Round-robin
  - Skip stalled threads
- Hardware support required
  - Separate PC and register file per thread
  - Hardware to control alternating pattern
- Naturally hides delays
  - Data hazards, Cache misses
  - Pipeline runs with rare stalls



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

Pros? Cons?



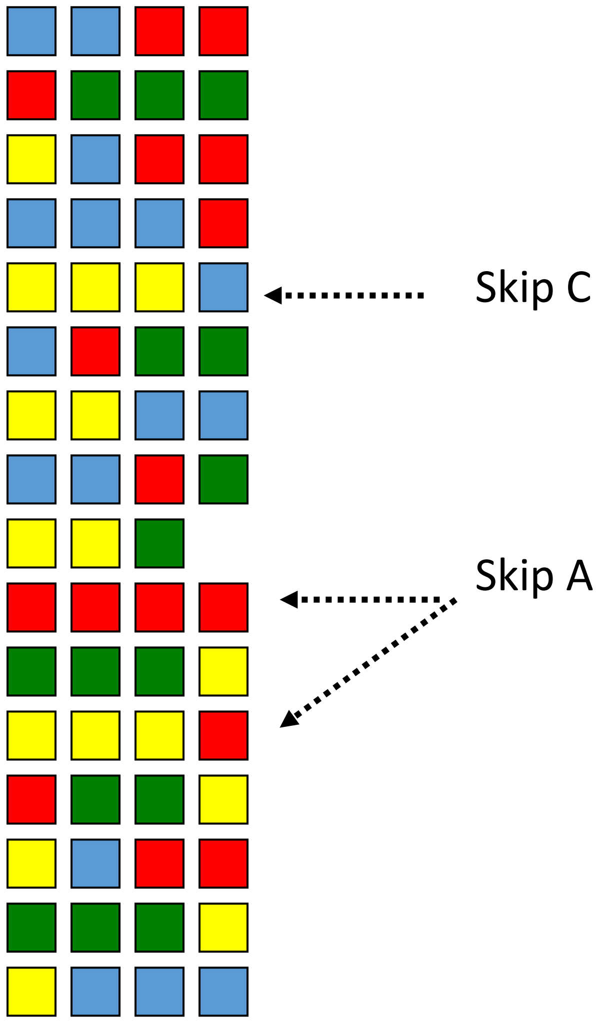
Simultaneous Multithreading (SMT)

Simultaneous Multithreading (SMT)

- Instructions from multiple threads issued on same cycle
  - Uses register renaming
  - dynamic scheduling facility of multi-issue architecture

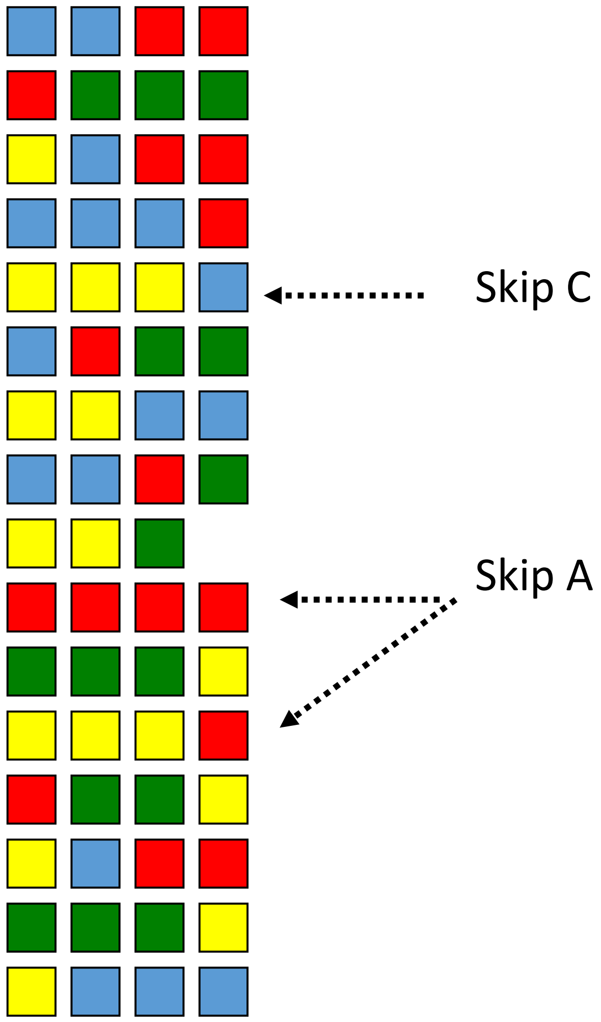
Simultaneous Multithreading (SMT)

- Instructions from multiple threads issued on same cycle
  - Uses register renaming
  - dynamic scheduling facility of multi-issue architecture



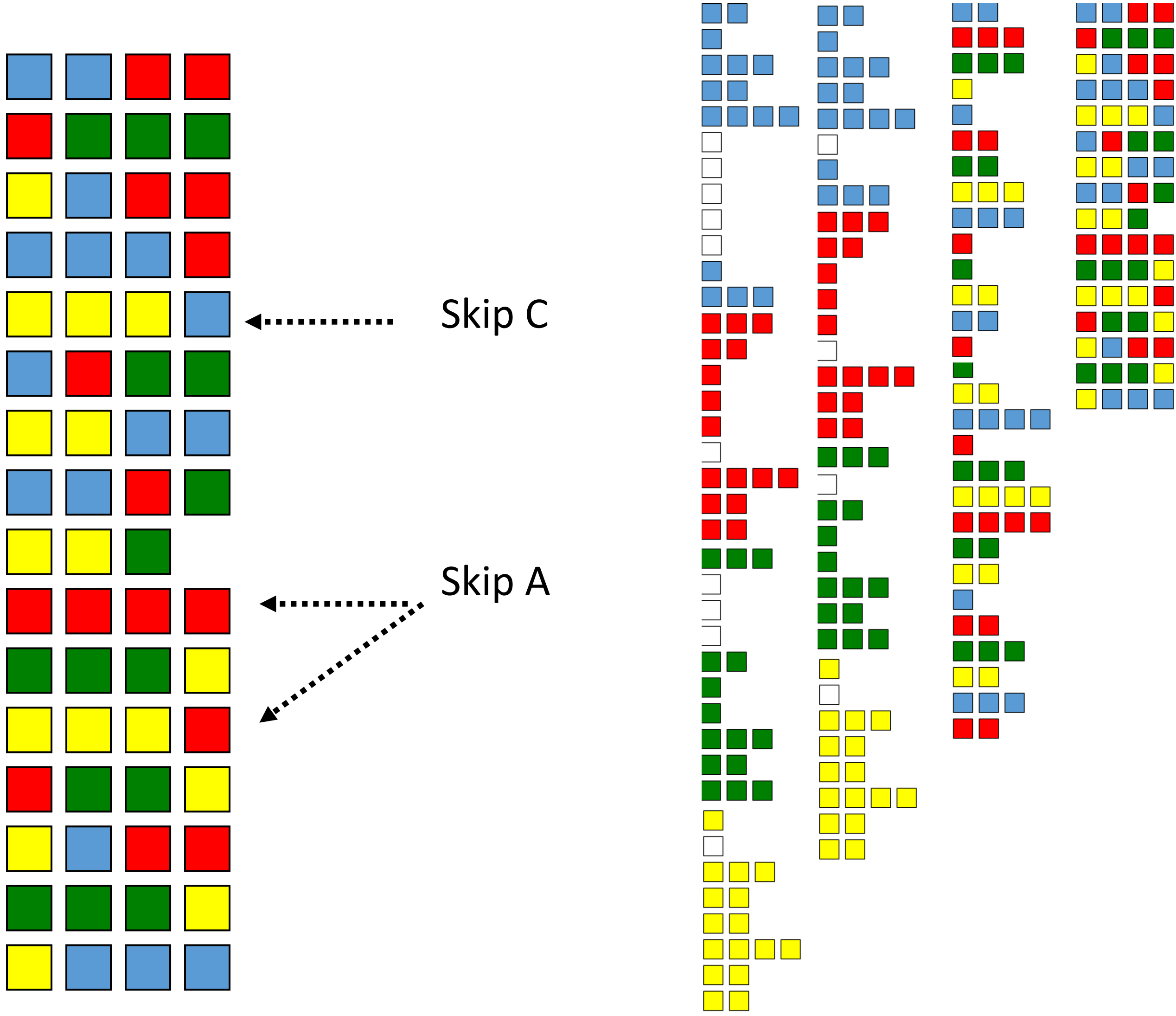
Simultaneous Multithreading (SMT)

- 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



Simultaneous Multithreading (SMT)

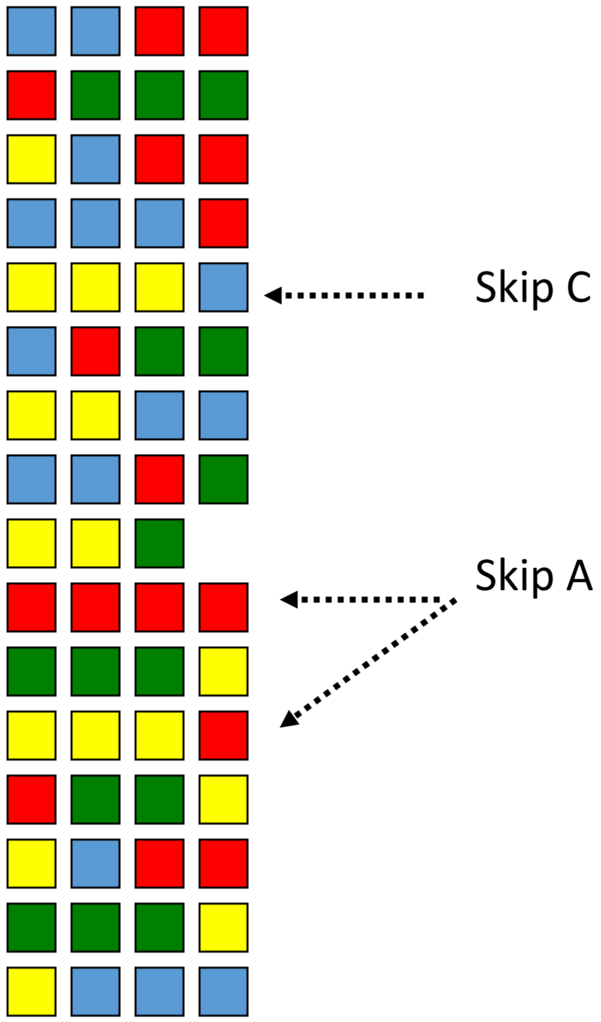
- Instructions from multiple threads issued on same cycle
  - Uses register renaming
  - dynamic scheduling facility of multi-issue architecture
- Hardware support:
  - Register files, PCs per thread
  - Temporary result registers pre commit
  - Support to sort out which threads get results from which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

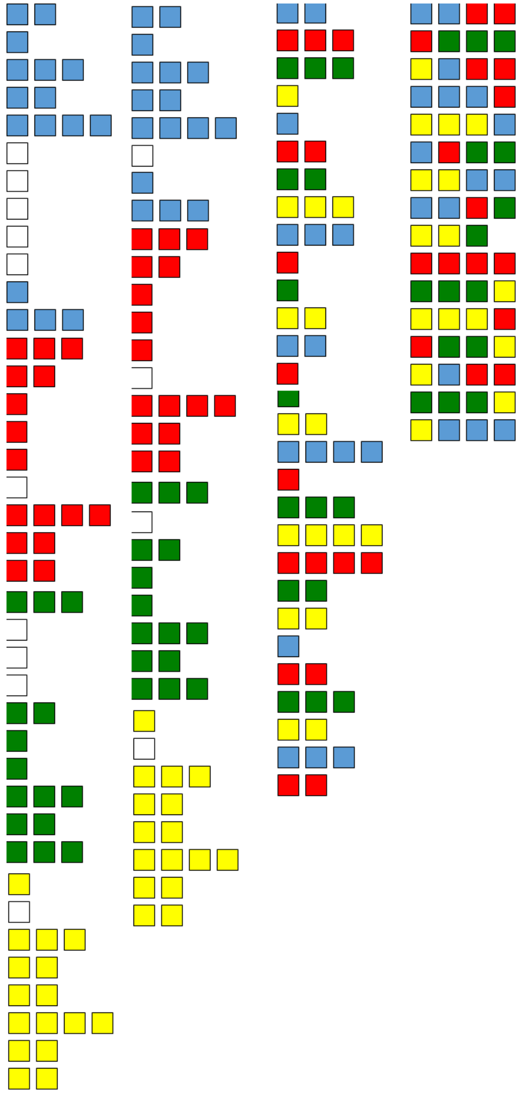


Simultaneous Multithreading (SMT)

- 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

Pros? Cons?





Why Vector and Multithreading Background?

Why Vector and Multithreading Background?

GPU:

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

Graphics ~= Rendering

Graphics ~= Rendering

Inputs

Graphics ~= Rendering

Inputs

- 3D world model(objects, materials)
    - Geometry modeled w triangle meshes, surface normals
    - GPUs subdivide triangles into “fragments” (rasterization)
    - Materials modeled with “textures”
    - Texture coordinates, sampling “map” textures → geometry

Graphics ~= Rendering

Inputs

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

Graphics ~= Rendering

Inputs

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

Graphics ~= Rendering

Inputs

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

Output

Graphics ~= Rendering

Inputs

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

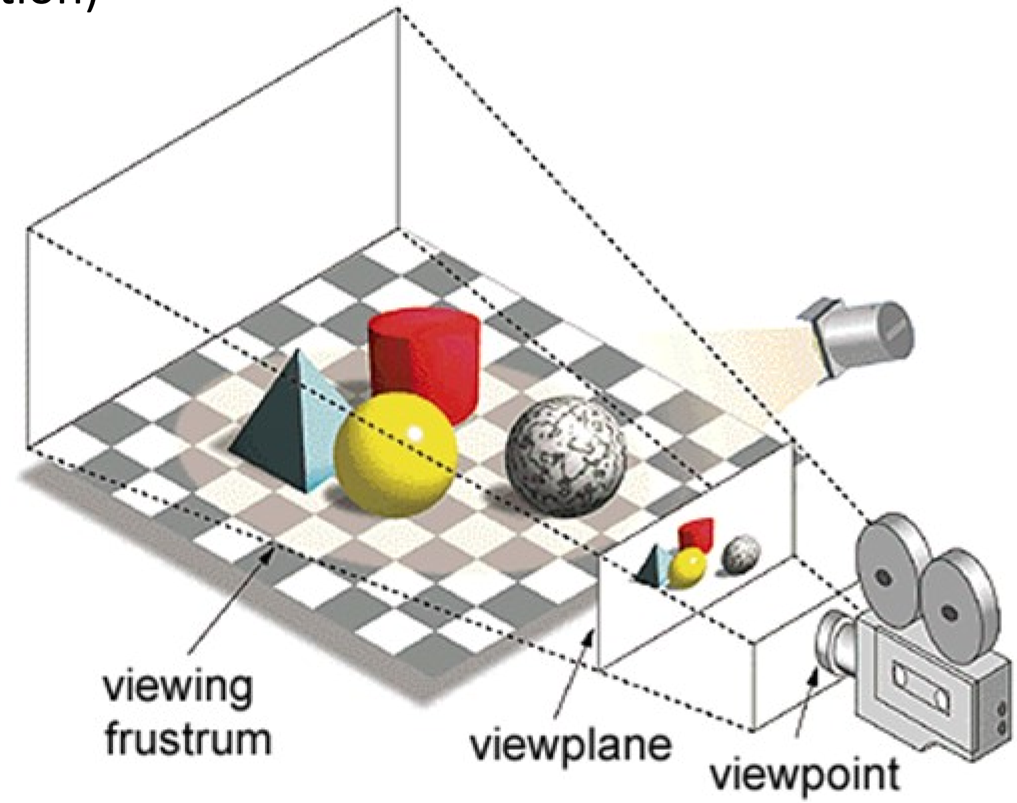
Output

- 2D projection seen from the view-point

Graphics ~= Rendering

Inputs

- 3D world model(objects, materials)
  - Geometry modeled w triangle meshes, surface normals
  - GPUs subdivide triangles into "fragments" (rasterization)
  - Materials modeled with "textures"
  - Texture coordinates, sampling "map" textures → 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

Grossly over-simplified rendering algorithm

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

Grossly over-simplified rendering algorithm

```
foreach(vertex v in model)
        map $v_{model}$ → $v_{view}$
```

Grossly over-simplified rendering algorithm

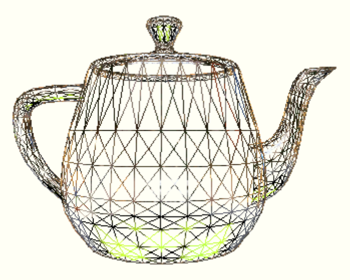
```
foreach(vertex v in model)
        map $v_{model} \rightarrow v_{view}$
```

Grossly over-simplified rendering algorithm

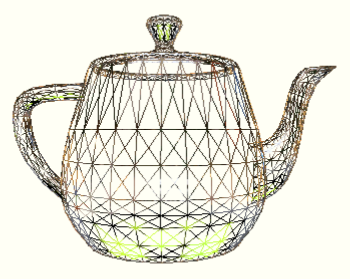
```
foreach(vertex v in model)
        map $v_{model}$ → $v_{view}$
fragment[] frags = {};
```

Grossly over-simplified rendering algorithm

```
foreach(vertex v in model)
        map $v_{model}$ → $v_{view}$
fragment[] frags = {};
foreach triangle t ($v_{0,}$ $v_{1,}$ $v_2$)
```

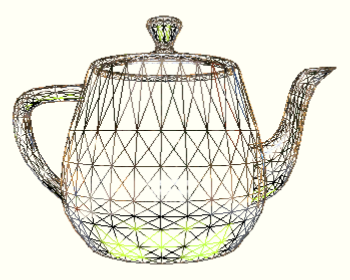


Grossly over-simplified rendering algorithm



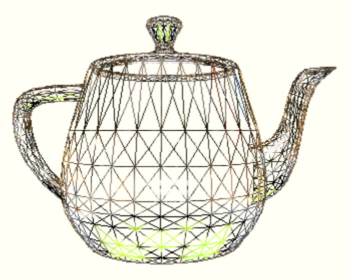
```
foreach(vertex v in model)
        map $v_{model}$ → $v_{view}$
fragment[] frags = {};
foreach triangle t ($v_{0,}$ $v_{1,}$ $v_2$)
        frags.add(rasterize(t));
```

Grossly over-simplified rendering algorithm



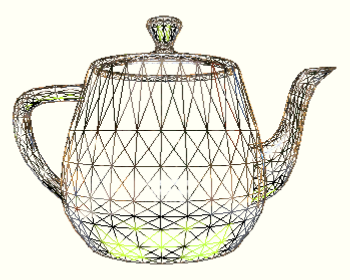
```
foreach(vertex v in model)
        map $v_{model}$ → $v_{view}$
fragment[] frags = {};
foreach triangle t ($v_{0,}$ $v_{1,}$ $v_2$)
        frags.add(rasterize(t));
foreach fragment f in frags
```

Grossly over-simplified rendering algorithm



```
foreach(vertex v in model)
        map $v_{model}$ → $v_{view}$
fragment[] frags = {};
foreach triangle t ($v_{0,}$ $v_{1,}$ $v_2$)
        frags.add(rasterize(t));
foreach fragment f in frags
        choose_color(f);
```

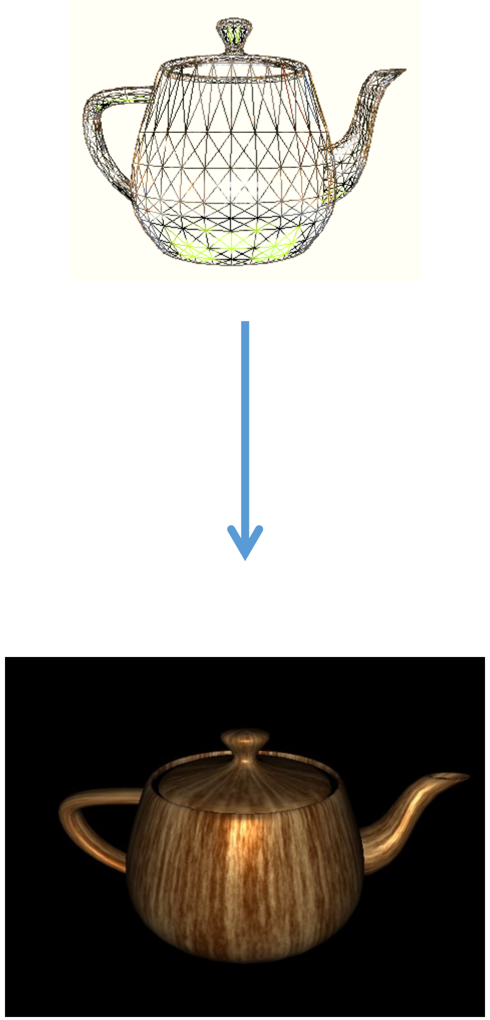
Grossly over-simplified rendering algorithm



```
foreach(vertex v in model)
        map $v_{model} \rightarrow v_{view}$
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));
```

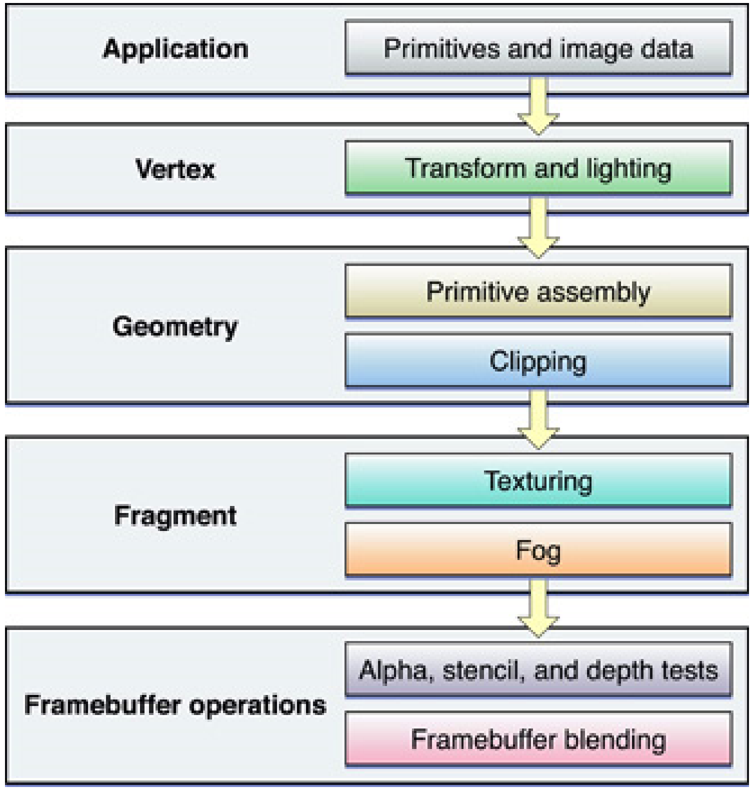
Grossly over-simplified rendering algorithm

```
foreach(vertex v in model)
        map $v_{model}$ → $v_{view}$
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 → Graphics Pipeline

```
foreach(vertex v in model)
        map $v_{model}$ → $v_{view}$
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));
```

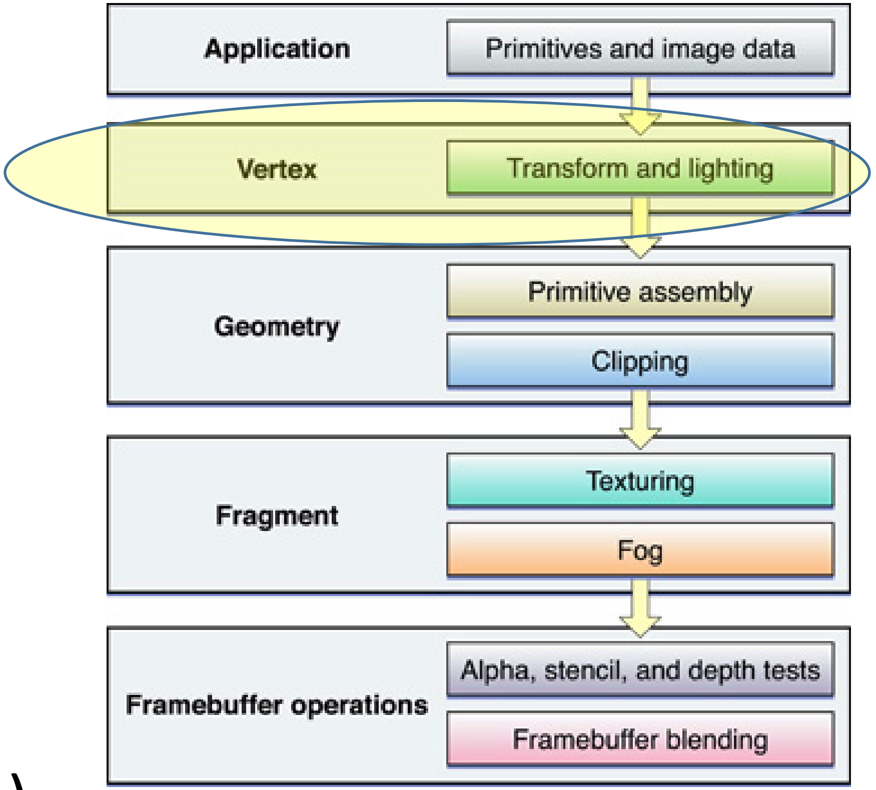


OpenGL pipeline

To first order, DirectX looks the same!

Algorithm → Graphics Pipeline

```
foreach(vertex v in model)
        map $v_{model} \rightarrow v_{view}$
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));
```

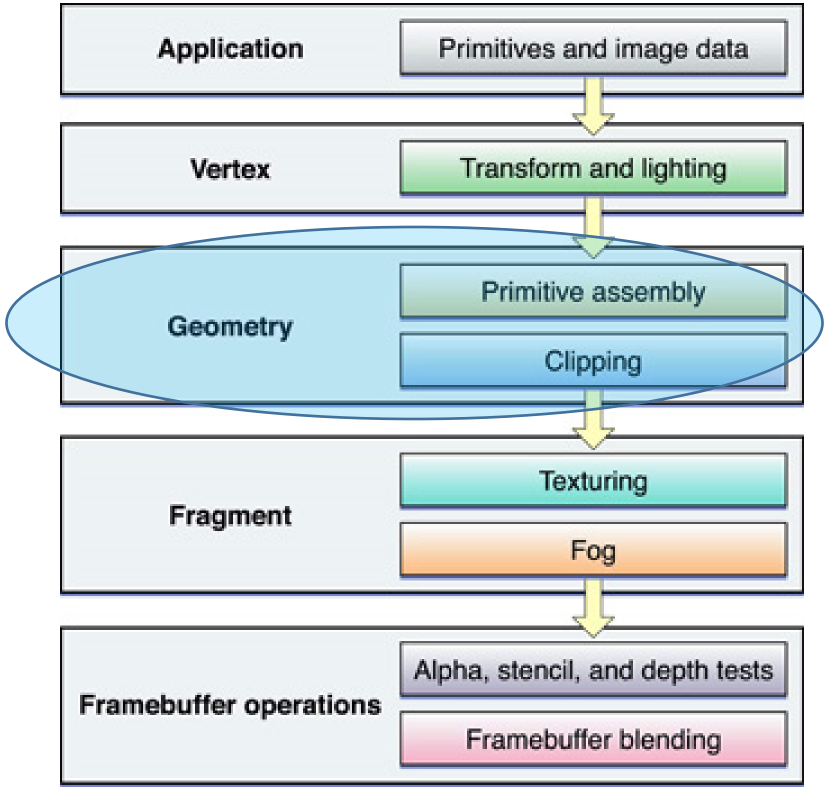


OpenGL pipeline

To first order, DirectX looks the same!

Algorithm → Graphics Pipeline

```
foreach(vertex v in model)
        map $v_{model}$ → $v_{view}$
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));
```

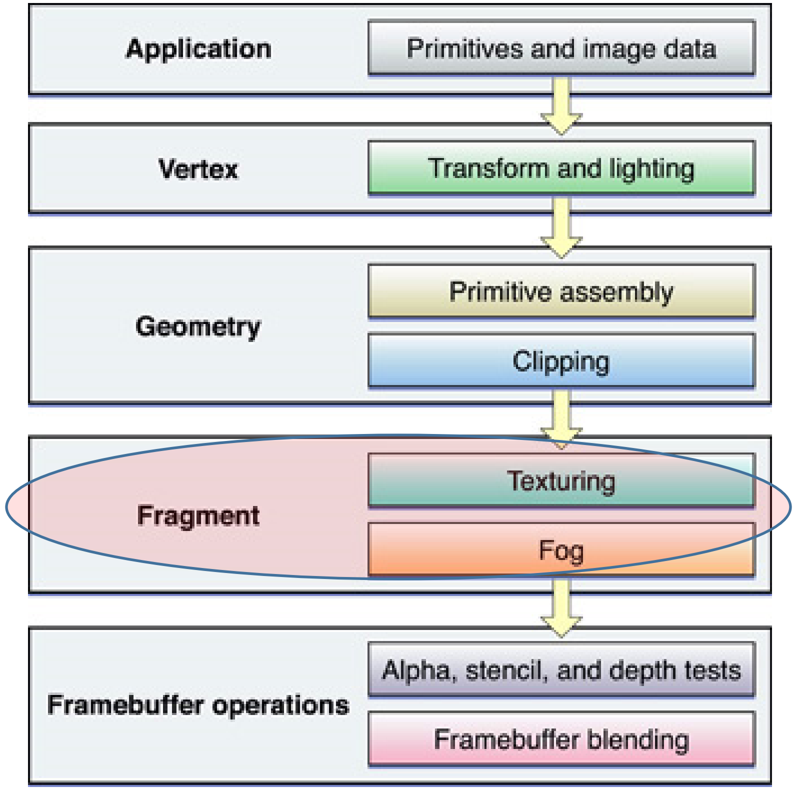


OpenGL pipeline

To first order, DirectX looks the same!

Algorithm → Graphics Pipeline

```
foreach(vertex v in model)
        map $v_{model}$ → $v_{view}$
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));
```



Application: Primitives and image data

Vertex: Transform and lighting

Geometry: Primitive assembly, Clipping

Fragment: Texturing, Fog

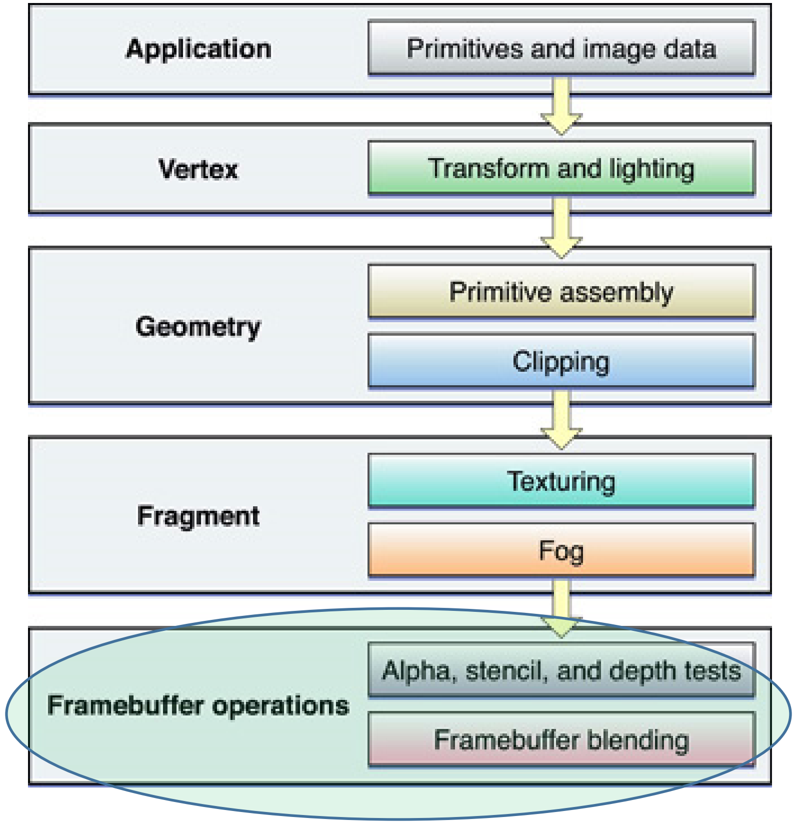
Framebuffer operations: Alpha, stencil, and depth tests, Framebuffer blending

OpenGL pipeline

To first order, DirectX looks the same!

Algorithm → Graphics Pipeline

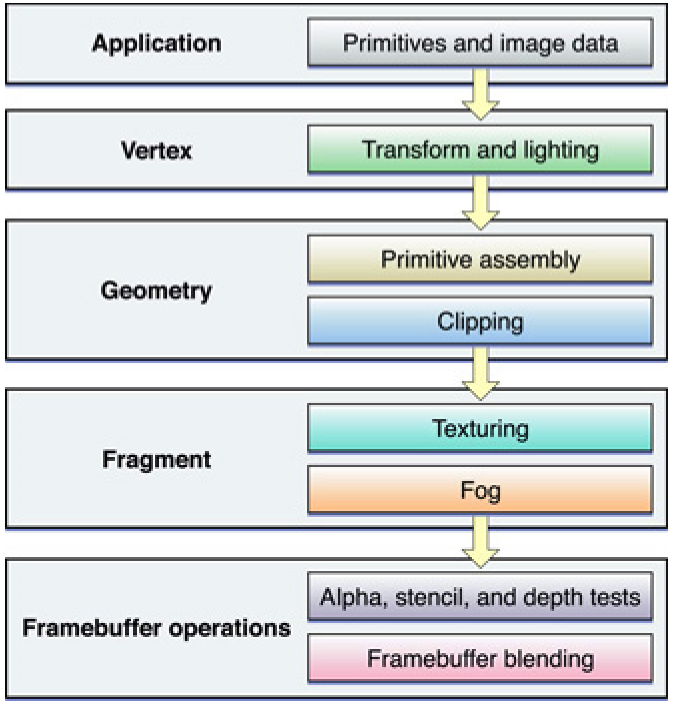
```
foreach(vertex v in model)
    map $v_{model}$ → $v_{view}$
fragment[] frags = {};
foreach triangle t ($v_{0,}$ $v_{1,}$ $v_2$)
    frags.add(rasterize(t));
foreach fragment f in frags
    choose_color(f);
display(visible_fragments(frags));
```

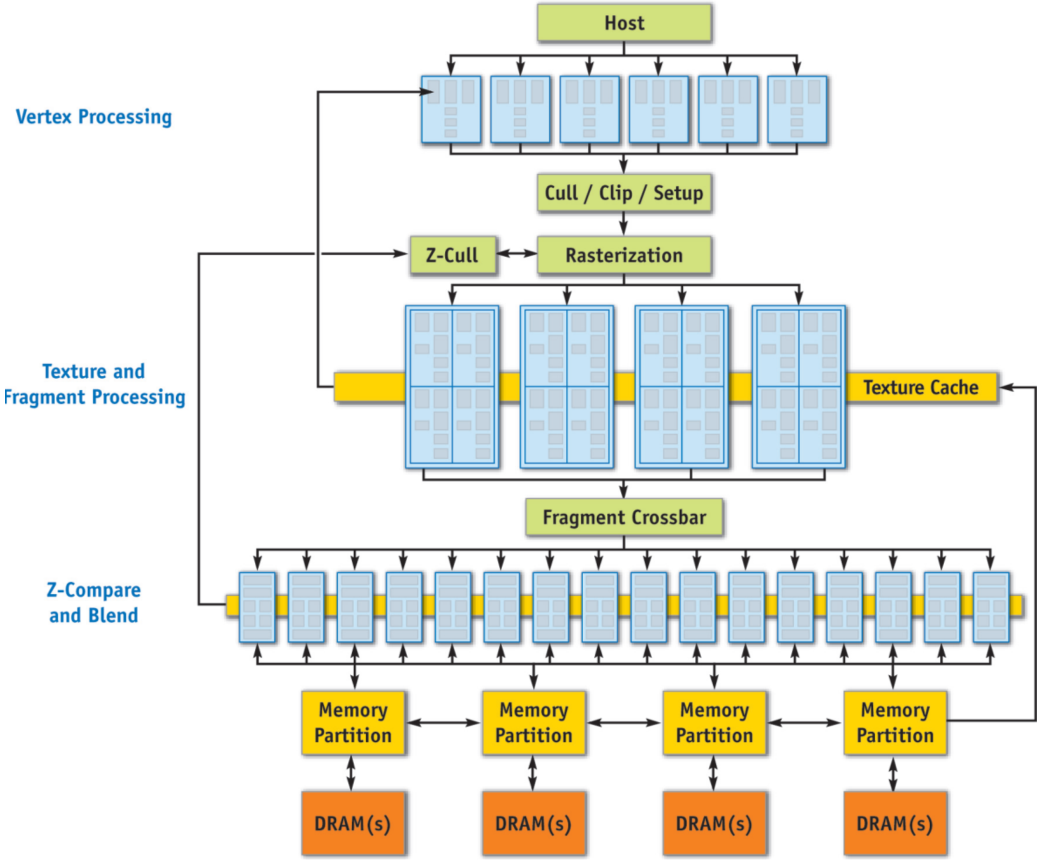


OpenGL pipeline

To first order, DirectX looks the same!

Graphics pipeline → GPU architecture

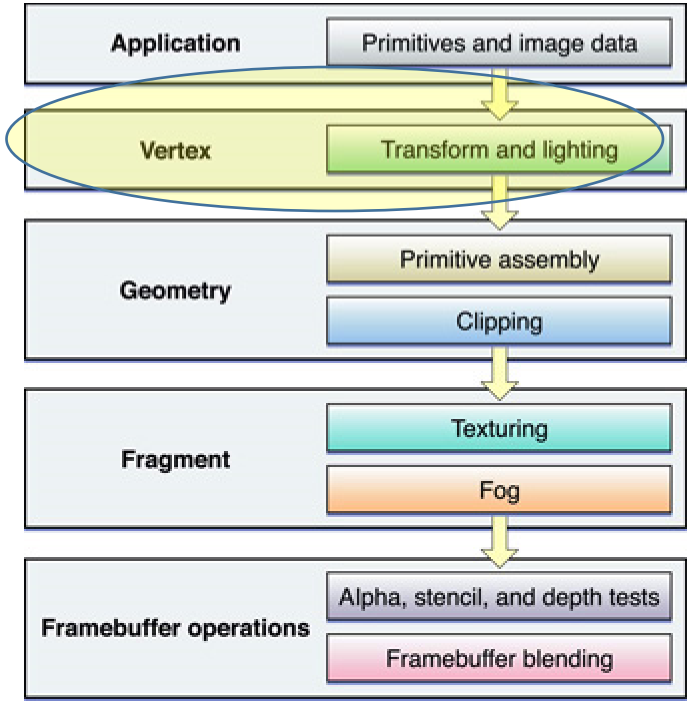


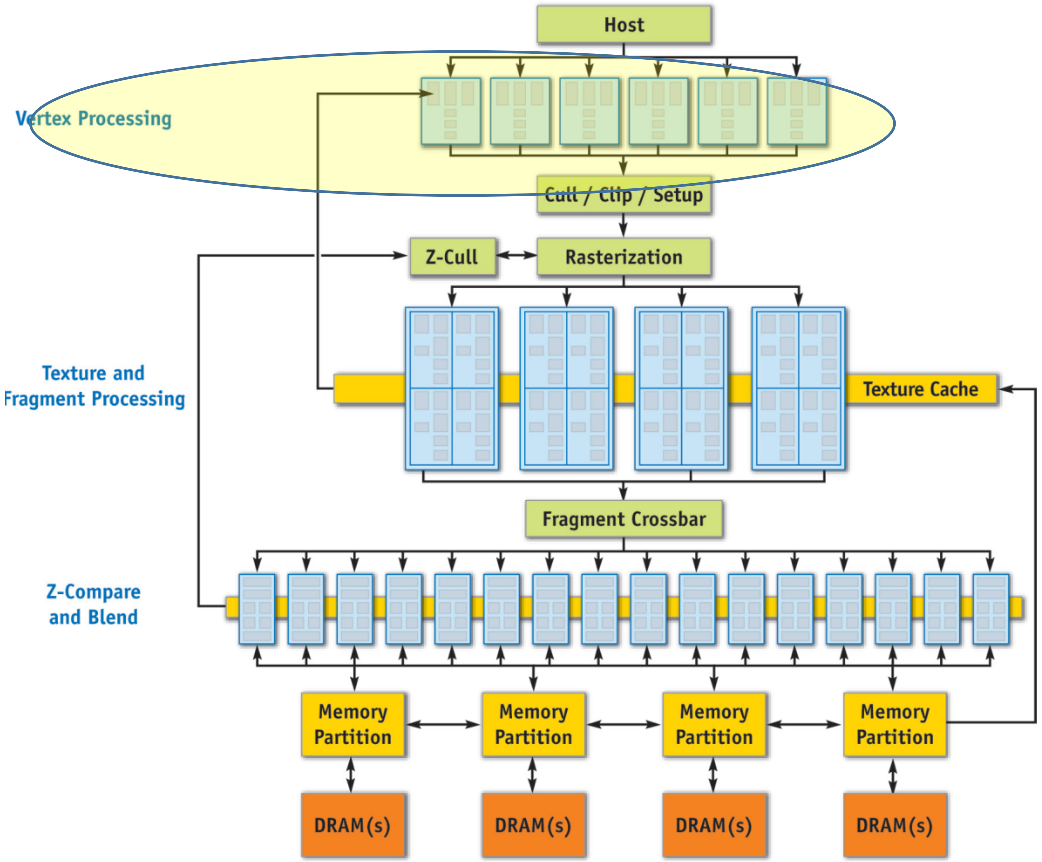


Limited "programmability" of shaders:
Minimal/no control flow
Maximum instruction count

GeForce 6 series

Graphics pipeline → GPU architecture

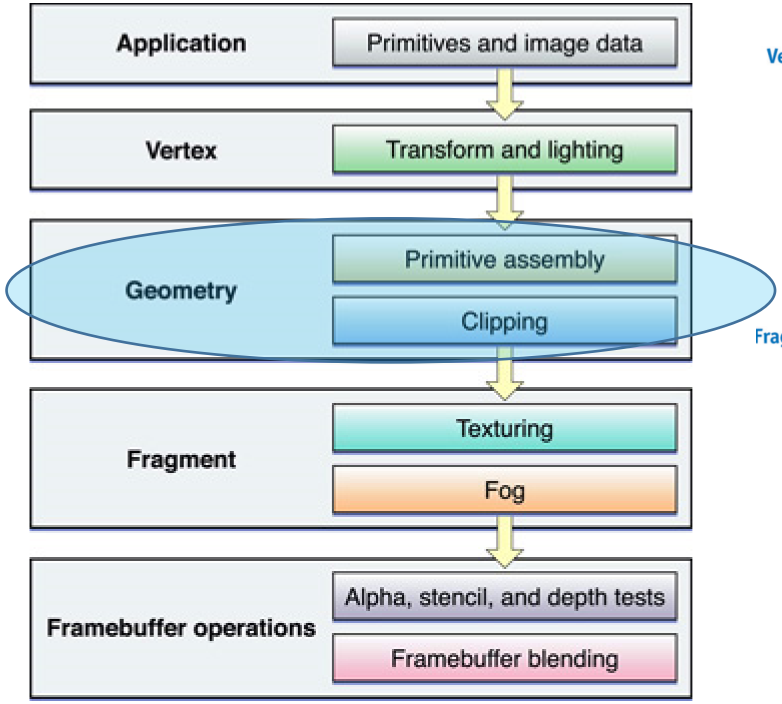


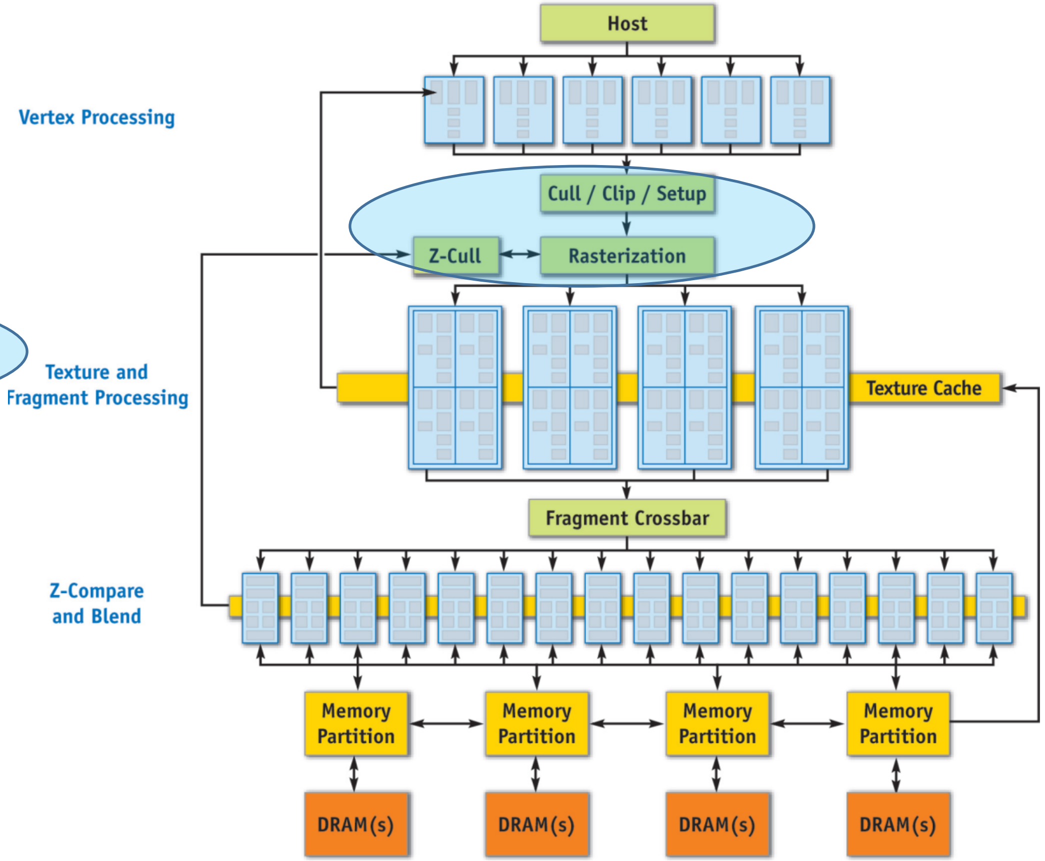


GeForce 6 series

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

Graphics pipeline → GPU architecture

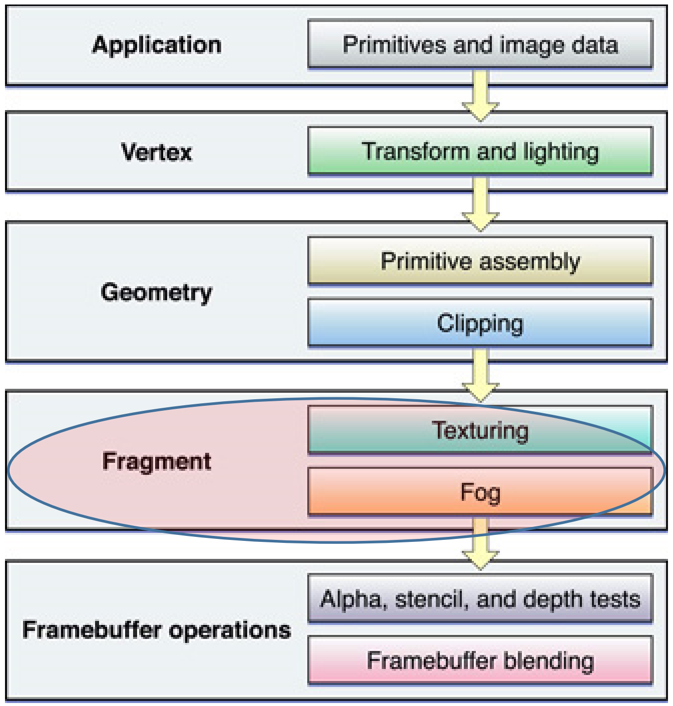


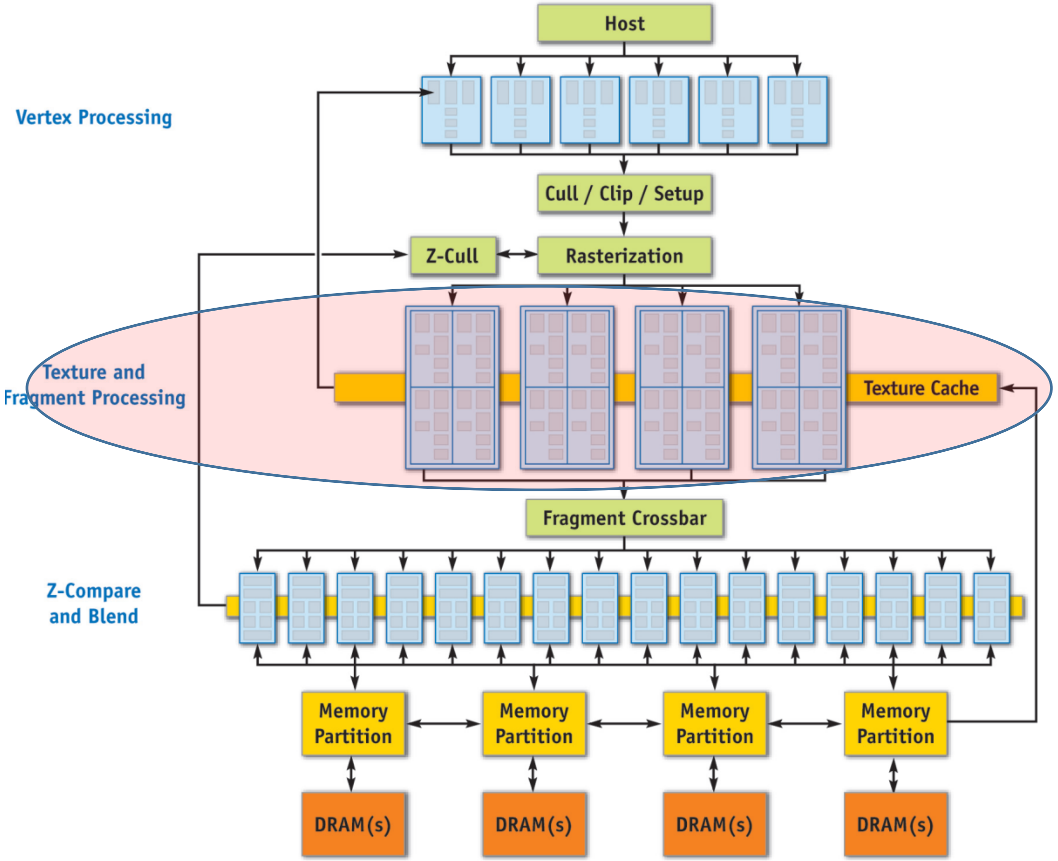


GeForce 6 series

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

Graphics pipeline → GPU architecture

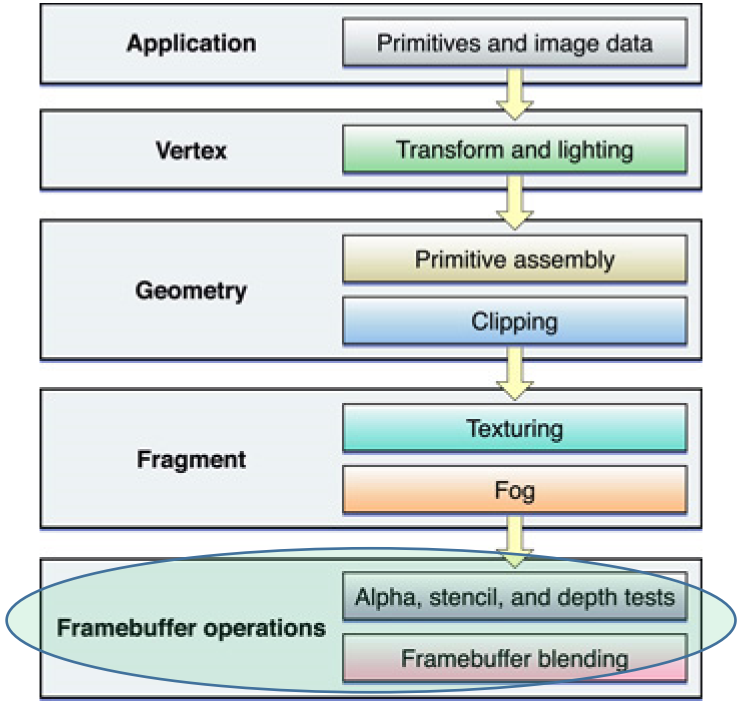


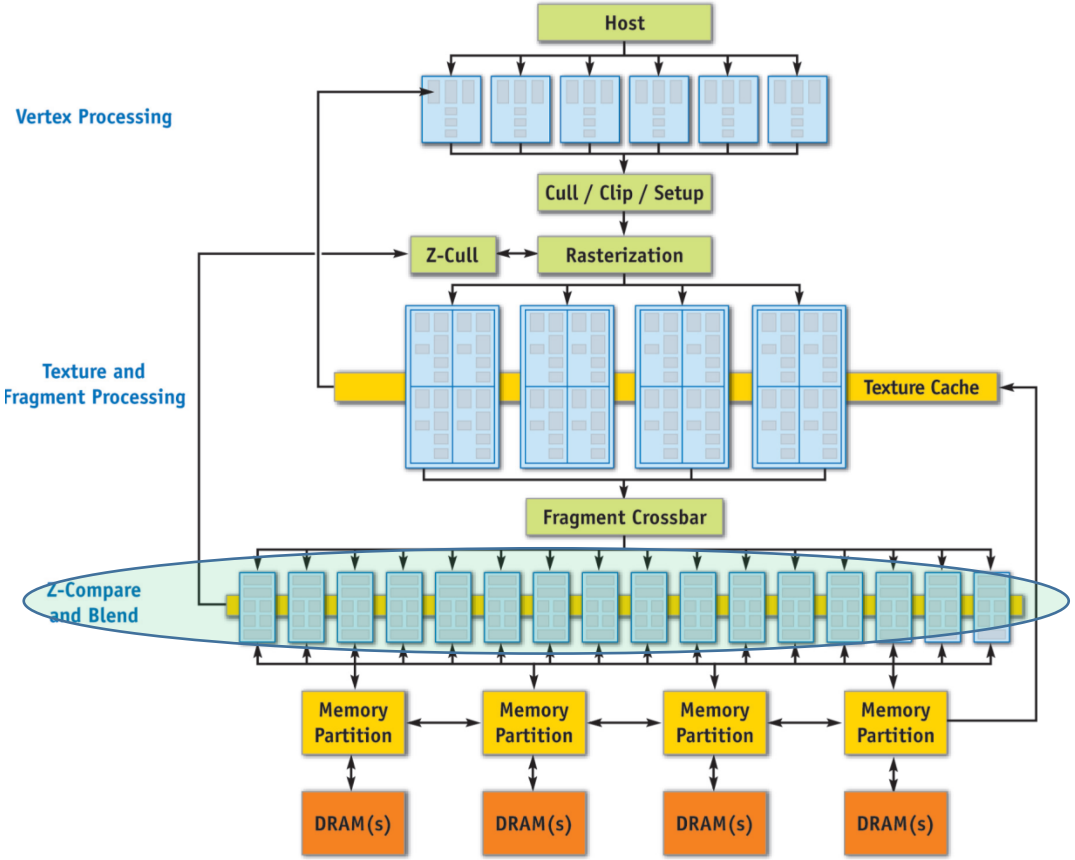


GeForce 6 series

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

Graphics pipeline → GPU architecture

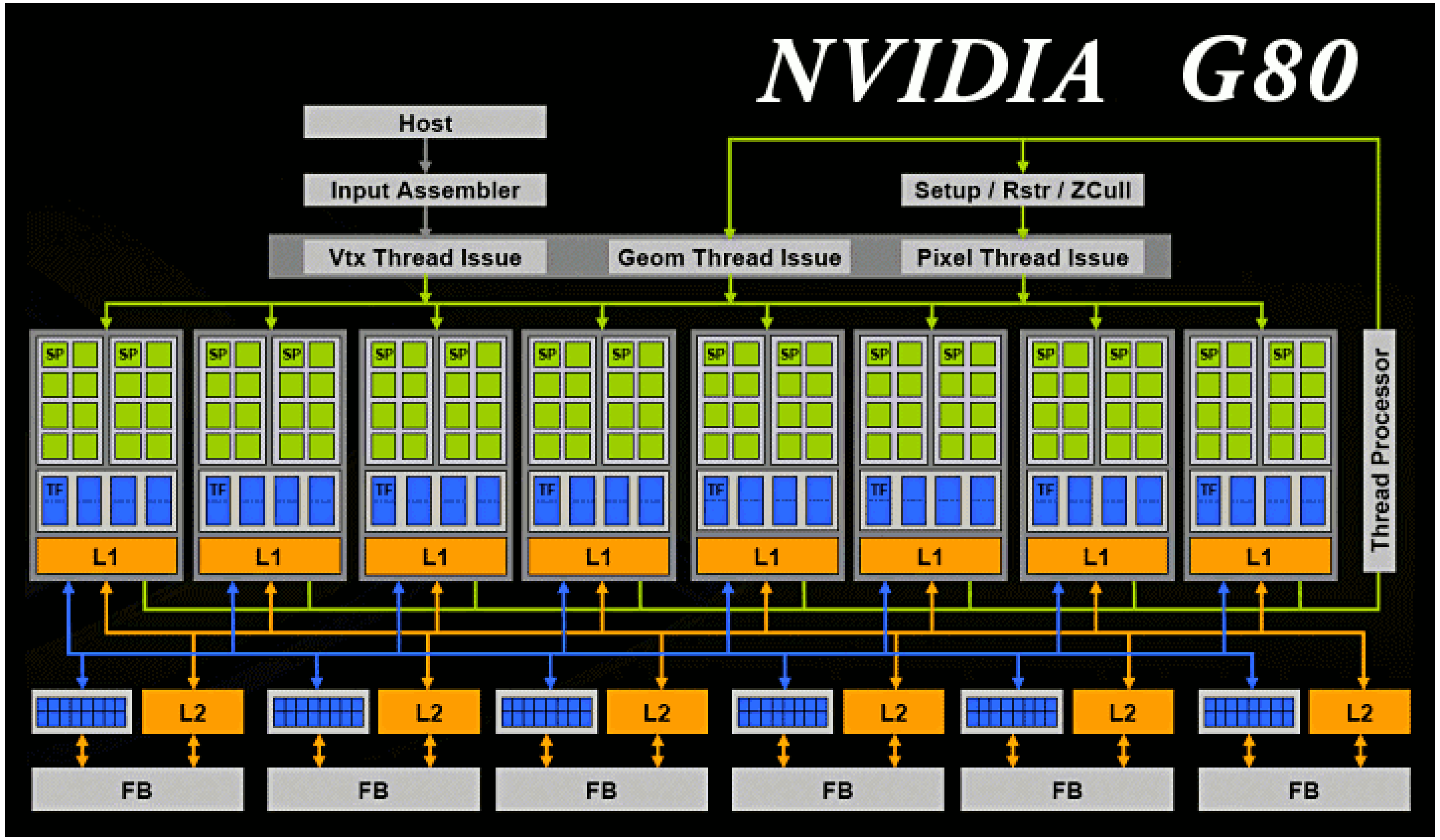




Limited "programmability" of shaders:
Minimal/no control flow
Maximum instruction count

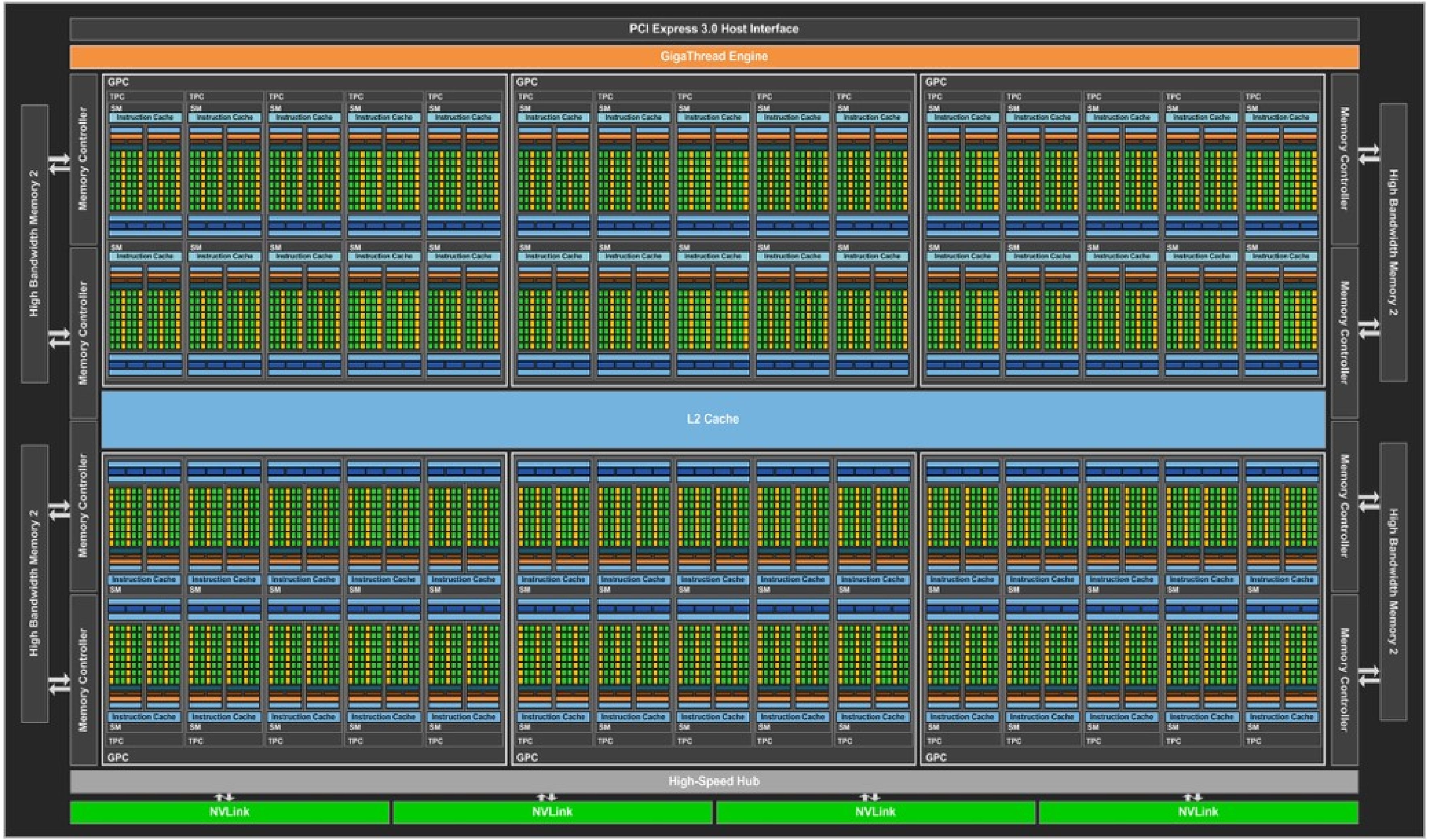
GeForce 6 series

Late Modernity: unified shaders

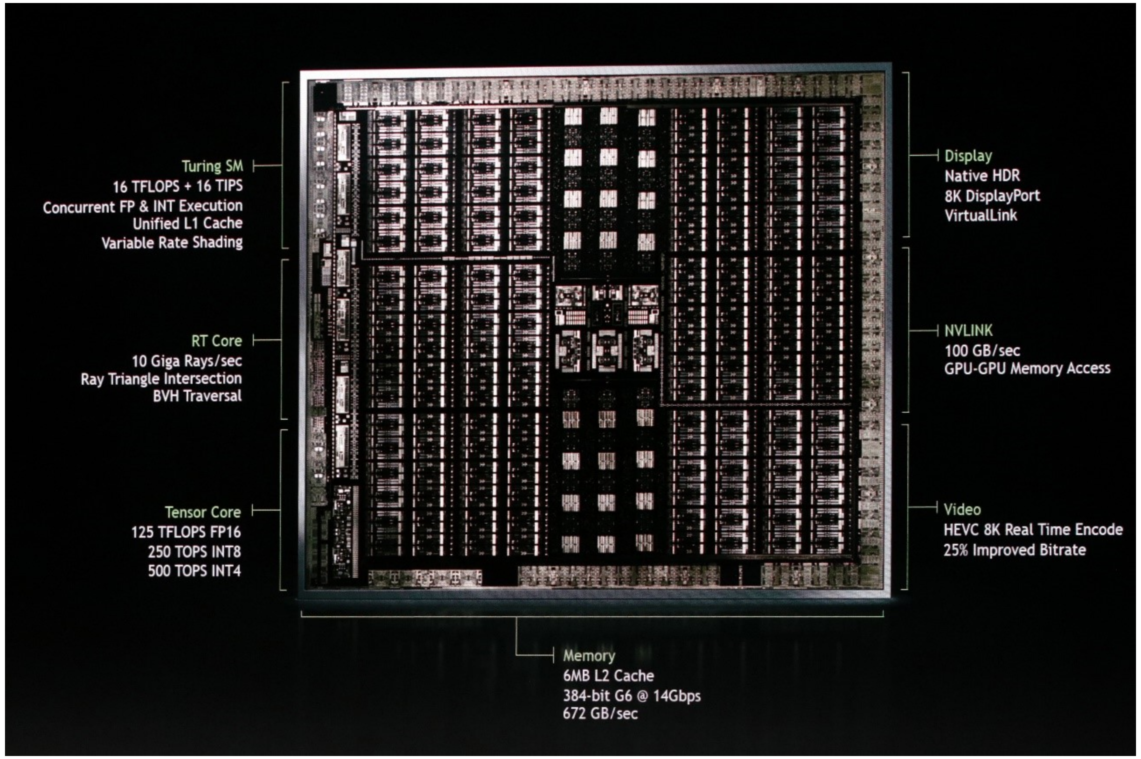


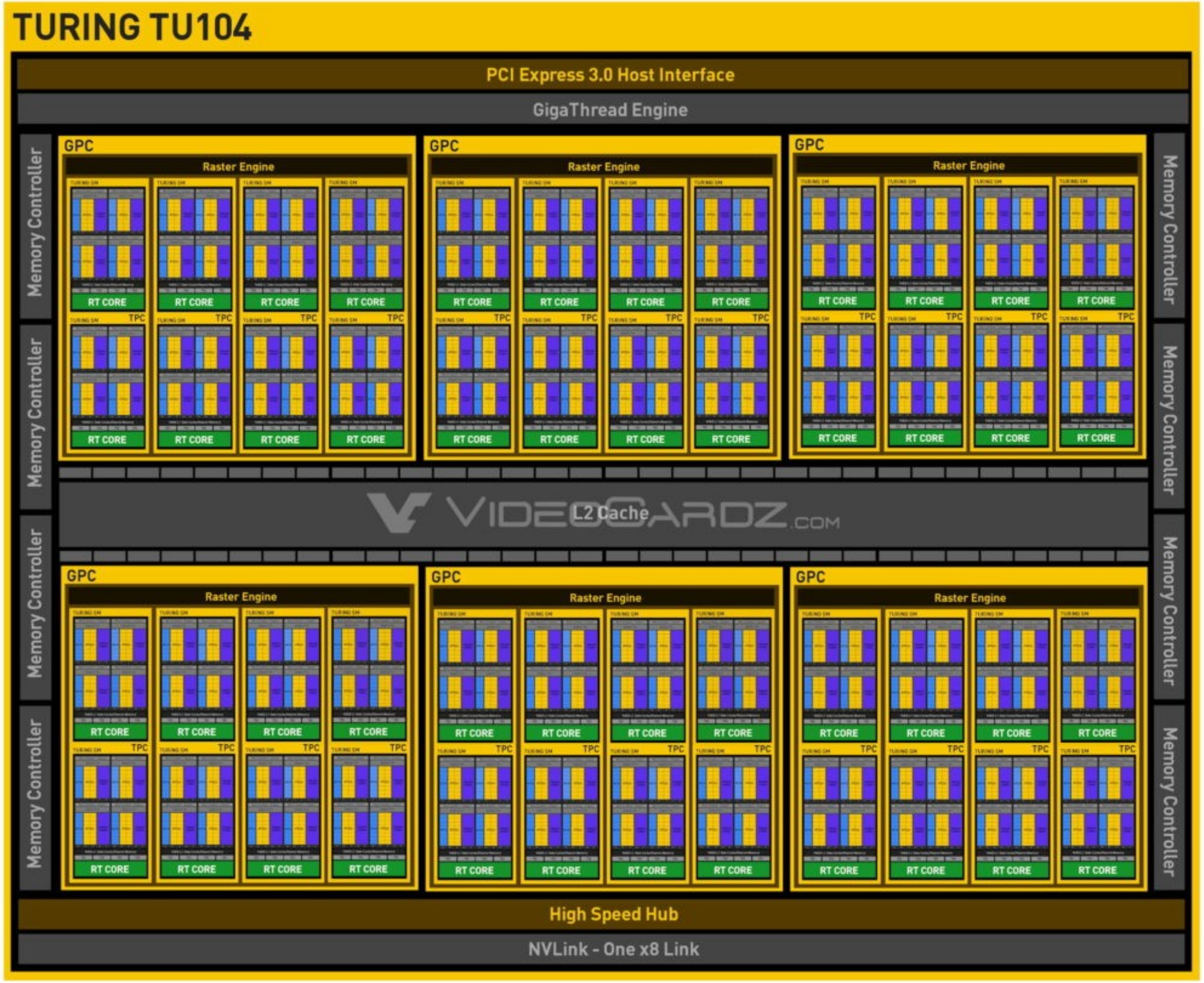
Mapping to Graphics pipeline no longer apparent
Processing elements no longer specialized to a particular role
Model supports *real* control flow, larger instr count

Mostly Modern: Pascal



Definitely Modern: Turing





Cross-generational GPU 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

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

Cross-generational GPU 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

- Simple cores
- Single instruction stream

OR

ring

n

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

Cross-generational GPU 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

- Simple cores
- Single instruction stream

OR

ing

n

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?!!?!

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)

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)

Need codes that are 1000s-X parallel….

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

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

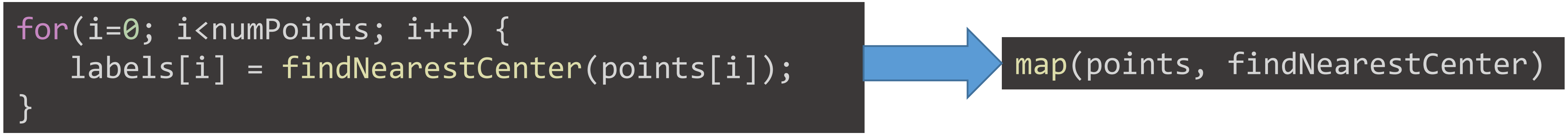
If you can express your algorithm using these patterns, an apparently fundamentally sequential algorithm can be made parallel

Map

- Inputs
  - Array A
  - Function f(x)
- map(A, f) → apply f(x) on all elements in A
- Parallelism trivially exposed
  - f(x) can be applied in parallel to all elements, in principle

Map

- Inputs
  - Array A
  - Function f(x)
- map(A, f) → 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

Scatter and Gather

- Gather:
    - Read multiple items to single /packed location

Scatter and Gather

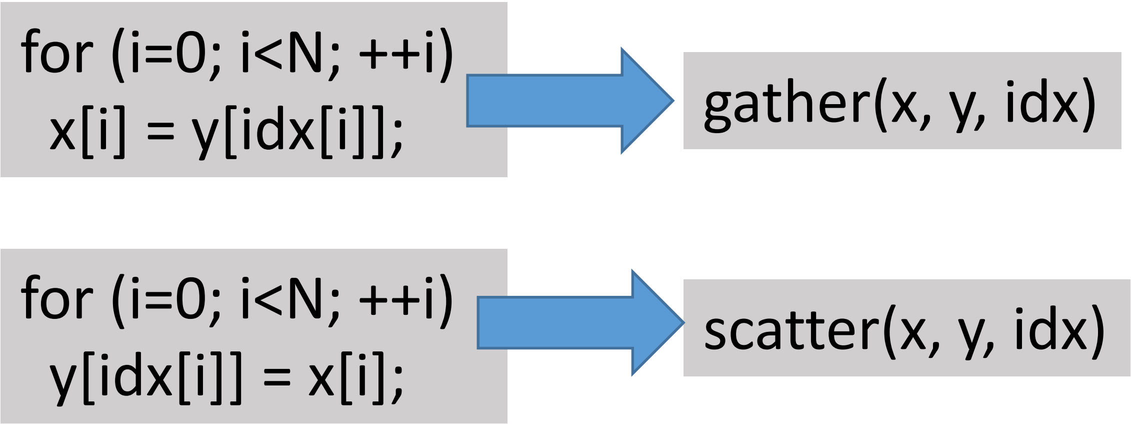
- Gather:
  - Read multiple items to single /packed location
- Scatter:
  - Write single/packed data item to multiple locations

Scatter and Gather

- Gather:
  - Read multiple items to single /packed location
- Scatter:
  - Write single/packed data item to multiple locations
- Inputs: x, y, indeces, N

Scatter and Gather

- Gather:
  - Read multiple items to single /packed location
- Scatter:
  - Write single/packed data item to multiple locations
- Inputs: x, y, indeces, N



```
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 and Gather

- Gather:
    - Read multiple items to single /packed location
- Scatter:
    - Write single/packed data item to multiple locations
- Inputs: x, y, indeces, N

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

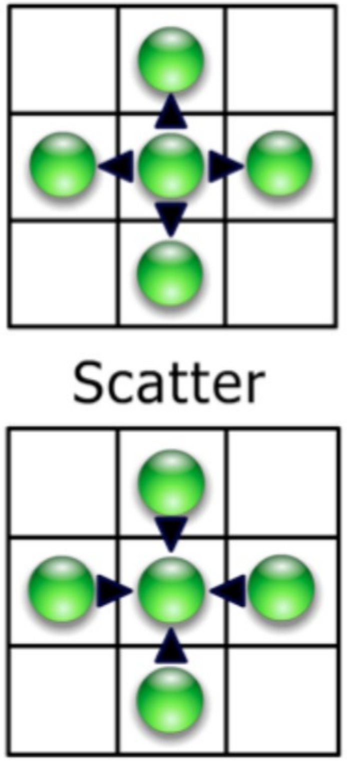
→

```
gather(x, y, idx)
```

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

→

```
scatter(x, y, idx)
```



Scatter

Gather

Scatter and Gather

Why is this useful on a box-drawing machine?

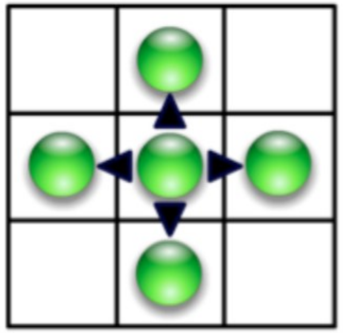
- Gather:
  - Read multiple items to single /packed location
- Scatter:
  - Write single/packed data item to multiple locations
- Inputs: x, y, indeces, N

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

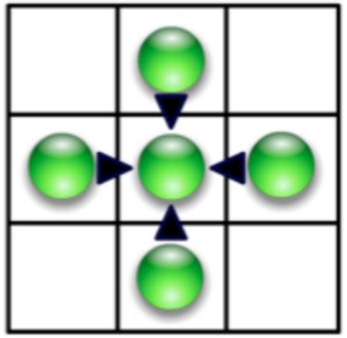
gather(x, y, idx)

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

scatter(x, y, idx)



Scatter



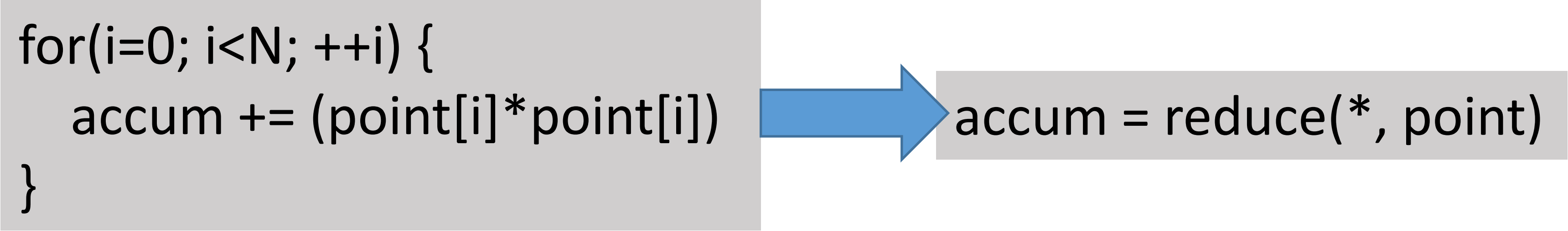
Gather

Reduce

- Input
  - Associative operator op
  - Ordered set s = [a, b, c, … z]
- Reduce(op, s) returns a op b op c … op z

Reduce

- Input
  - Associative operator op
  - Ordered set s = [a, b, c, … z]
- Reduce(op, s) returns a op b op c … op z

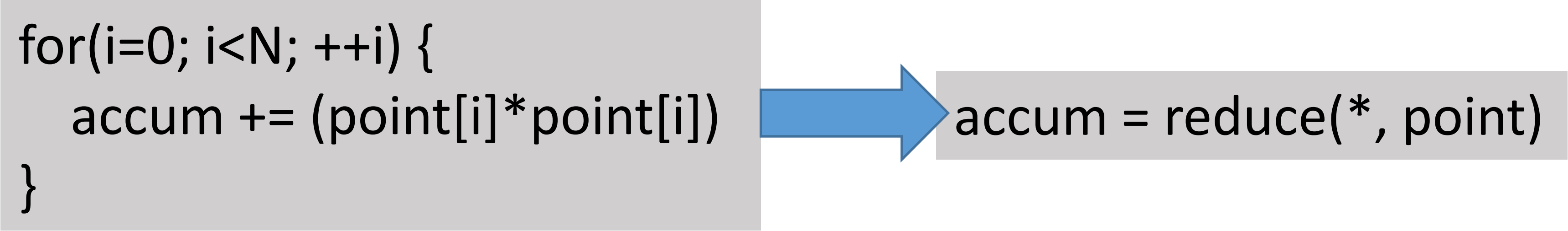


```
for(i=0; i<N; ++i) {
   accum += (point[i]*point[i])
}
```

```
accum = reduce(*, point)
```

Reduce

- Input
  - Associative operator op
  - Ordered set s = [a, b, c, … z]
- Reduce(op, s) returns a op b op c … op z



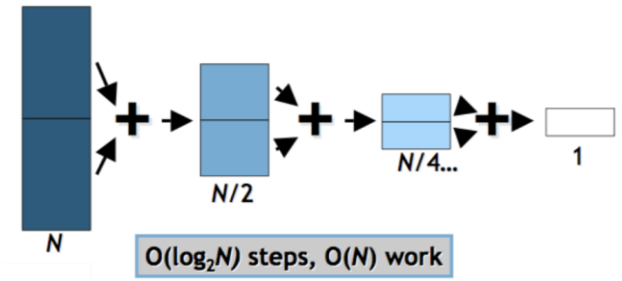
```
for(i=0; i<N; ++i) {
    accum += (point[i]*point[i])
}
```

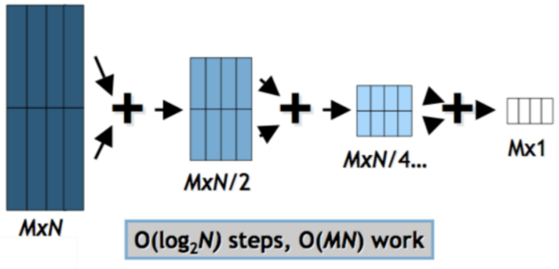
```
accum = reduce(*, point)
```

Why must op be associative?

Reduce

- Input
  - Associative operator op
  - Ordered set s = [a, b, c, … z]
- Reduce(op, s) returns a op b op c … op z





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

```
accum = reduce(*, point)
```

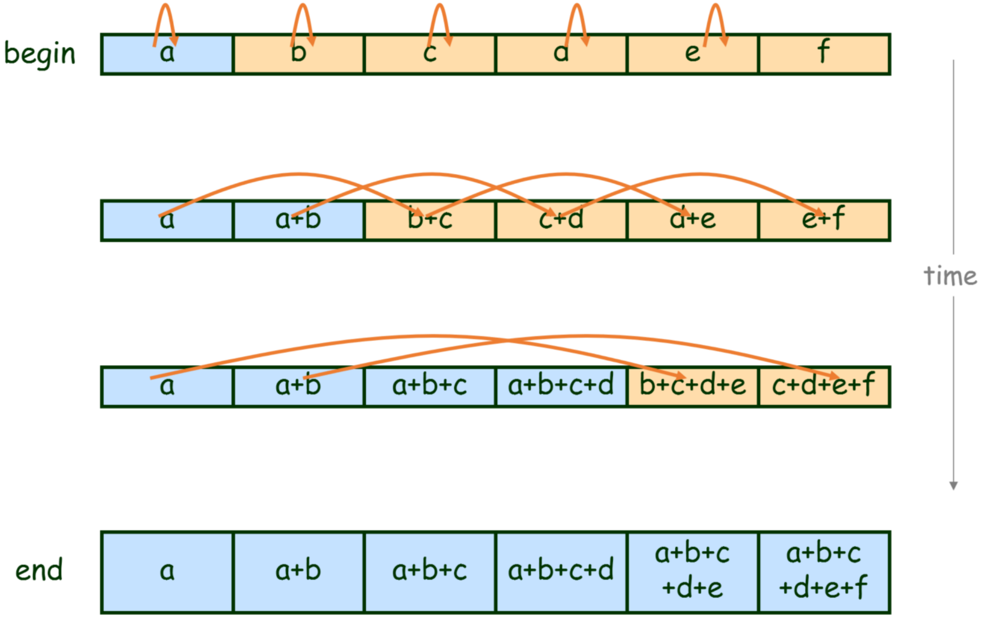
Why must op be associative?

Scan (prefix sum)

- Input
  - Associative operator op
  - Ordered set s = [a, b, c, … z]
  - Identity I

- scan(op, s) = [I, a, (a op b), (a op b op c) …]

- Scan is the workhorse of parallel algorithms:
  - Sort, histograms, sparse matrix, string compare, …



begin a b c d e f

a a+b b+c c+d d+e e+f

a a+b a+b+c a+b+c+d b+c+d+e c+d+e+f

end a a+b a+b+c a+b+c+d a+b+c +d+e a+b+c +d+e+f

GroupBy

- Group a collection by key
- Lambda function maps elements → key

GroupBy

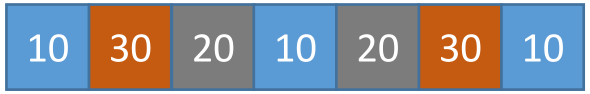
- Group a collection by key
- Lambda function maps elements → key

```
var res = ints.GroupBy(x => x);
```

GroupBy

- Group a collection by key
- Lambda function maps elements → key

```
var res = ints.GroupBy(x => x);
```

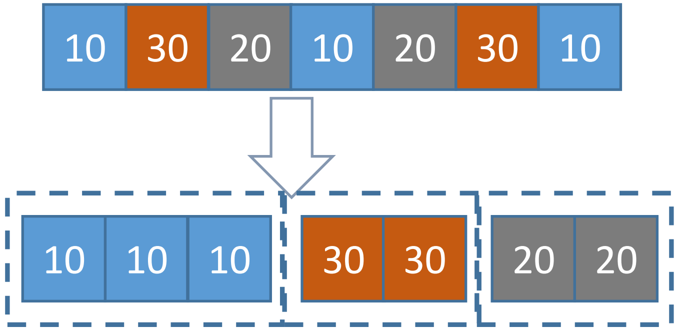


10	30	20	10	20	30	10

GroupBy

- Group a collection by key
- Lambda function maps elements → key

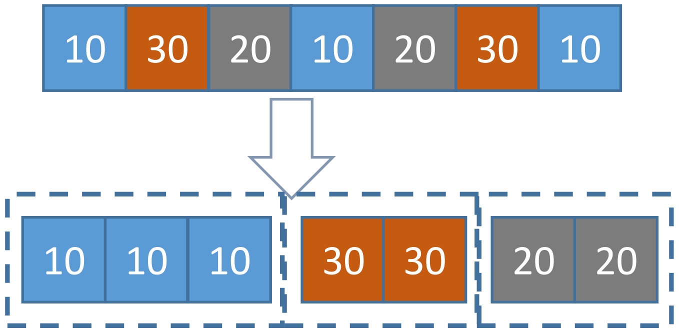
```
var res = ints.GroupBy(x => x);
```



GroupBy

- Group a collection by key
- Lambda function maps elements → key

```
var res = ints.GroupBy(x => x);
```

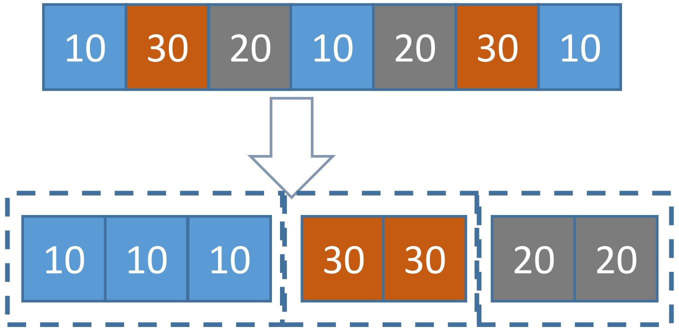


```
foreach(T elem in ints)
{
   key   = KeyLambda(elem);

   group = GetGroup(key);

   group.Add(elem);
}
```

GroupBy

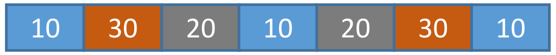
- Group a collection by key
- Lambda function maps elements → key

```
var res = ints.GroupBy(x => x);
```



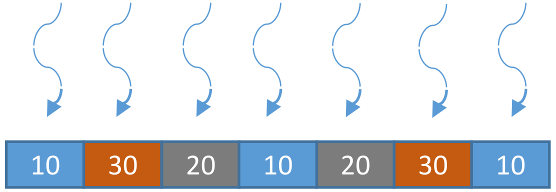
```
foreach(T elem in PF(ints))
{
   key   = KeyLambda(elem);

   group = GetGroup(key);

   group.Add(elem);
}
```

GroupBy using parallel primitives



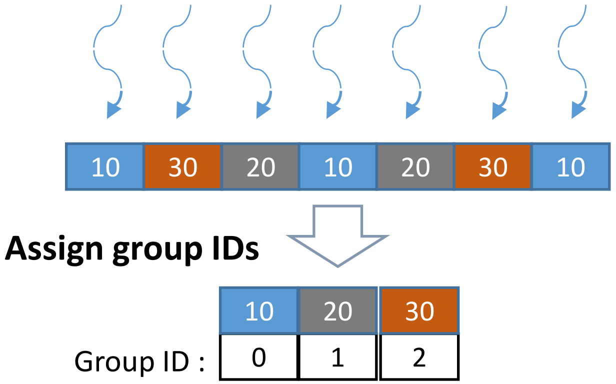
10	30	20	10	20	30	10

GroupBy using parallel primitives



10	30	20	10	20	30	10

GroupBy using parallel primitives

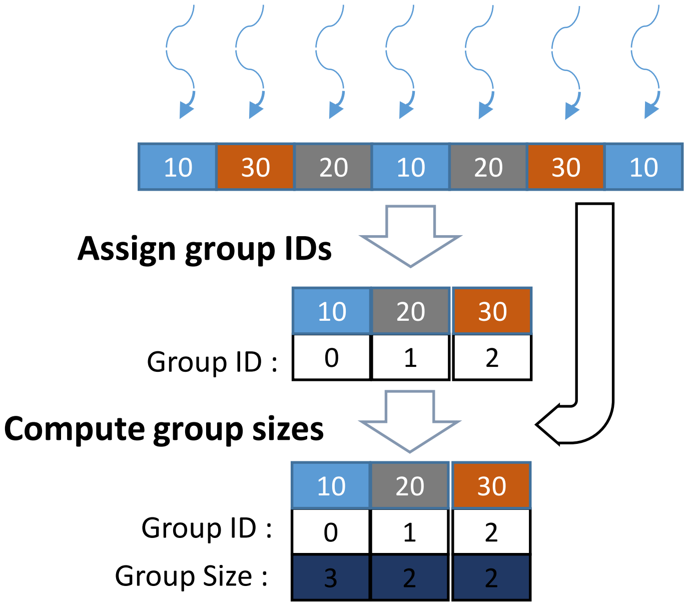


10	30	20	10	20	30	10

Assign group IDs

	10	20	30
Group ID :	0	1	2

GroupBy using parallel primitives



10	30	20	10	20	30	10

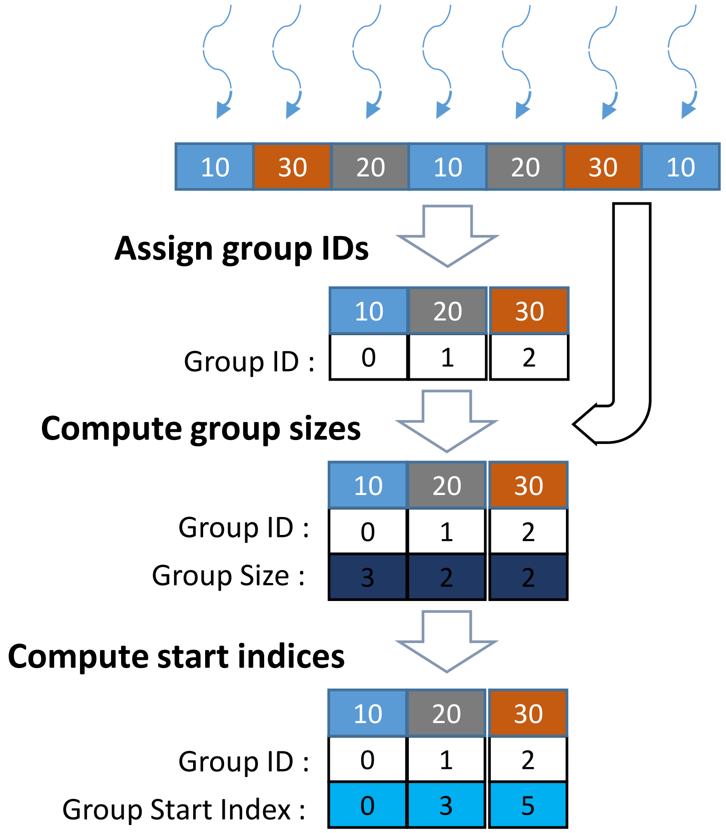
Assign group IDs

	10	20	30
Group ID :	0	1	2

Compute group sizes

	10	20	30
Group ID :	0	1	2
Group Size :	3	2	2

GroupBy using parallel primitives



10	30	20	10	20	30	10

Assign group IDs

	10	20	30
Group ID :	0	1	2

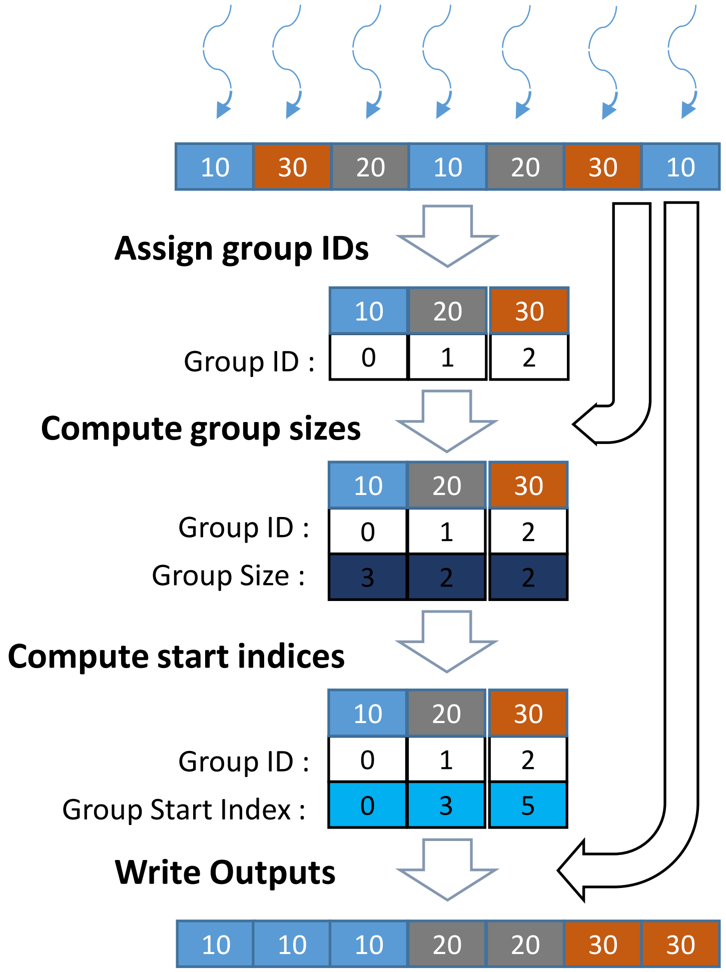
Compute group sizes

	10	20	30
Group ID :	0	1	2
Group Size :	3	2	2

Compute start indices

	10	20	30
Group ID :	0	1	2
Group Start Index :	0	3	5

GroupBy using parallel primitives



10 | 30 | 20 | 10 | 20 | 30 | 10

Assign group IDs

	10	20	30
Group ID :	0	1	2

Compute group sizes

	10	20	30
Group ID :	0	1	2
Group Size :	3	2	2

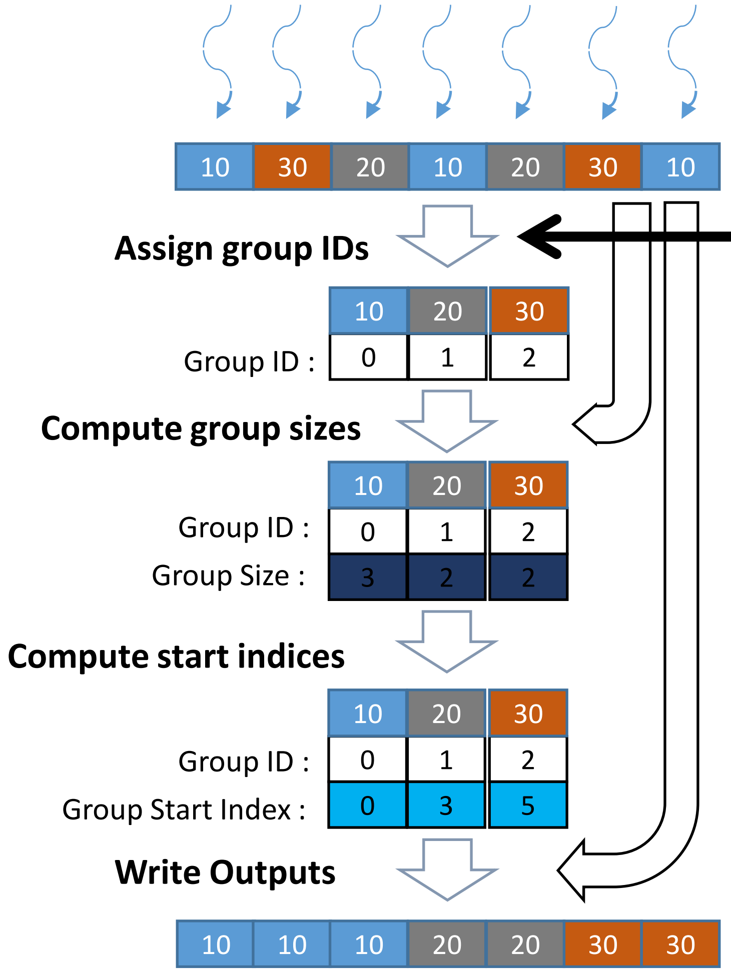
Compute start indices

	10	20	30
Group ID :	0	1	2
Group Start Index :	0	3	5

Write Outputs

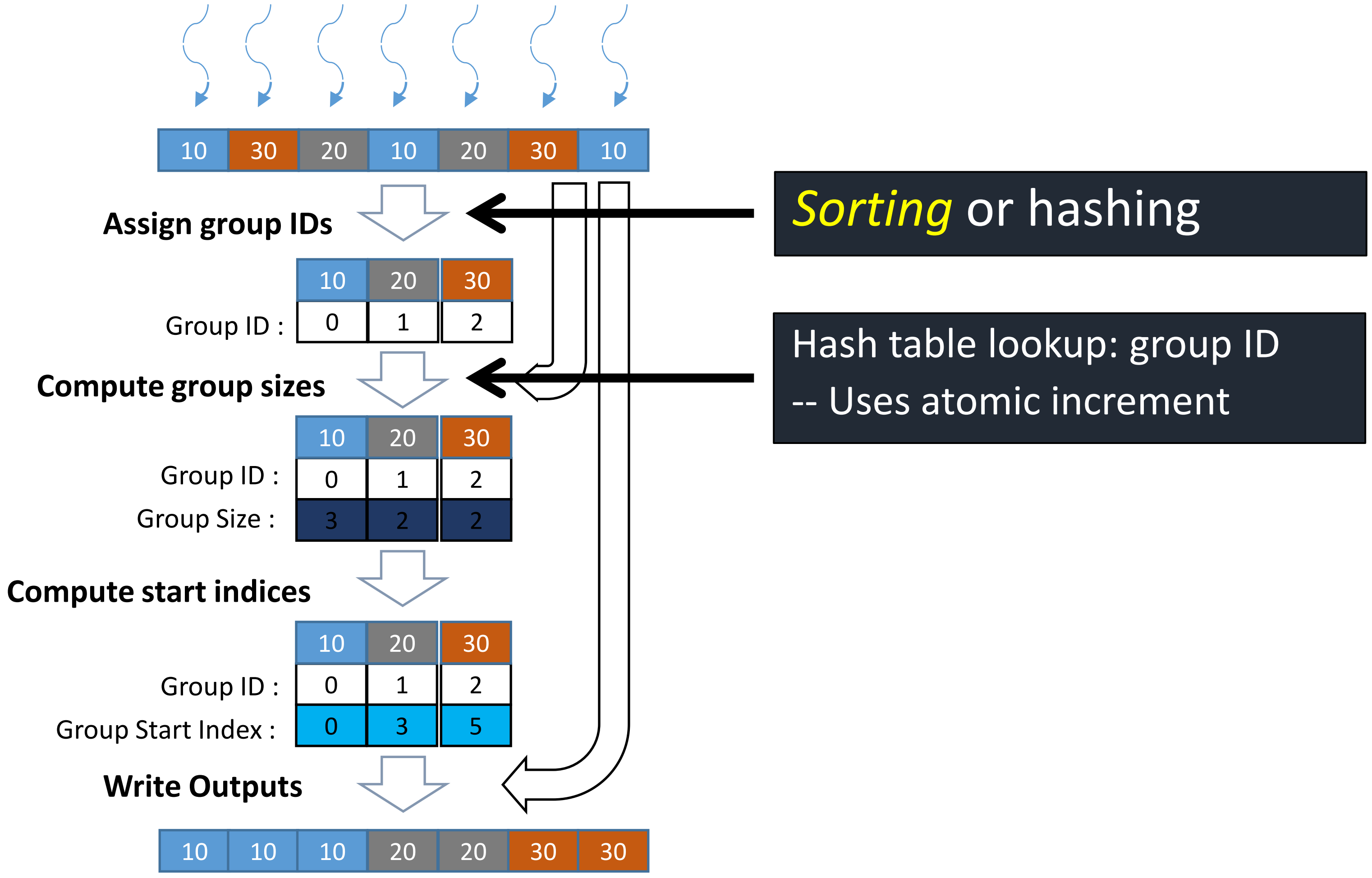
10 | 10 | 10 | 20 | 20 | 30 | 30

GroupBy using parallel primitives



Sorting or hashing

GroupBy using parallel primitives



10	30	20	10	20	30	10

Assign group IDs

	10	20	30
Group ID :	0	1	2

Compute group sizes

	10	20	30
Group ID :	0	1	2
Group Size :	3	2	2

Compute start indices

	10	20	30
Group ID :	0	1	2
Group Start Index :	0	3	5

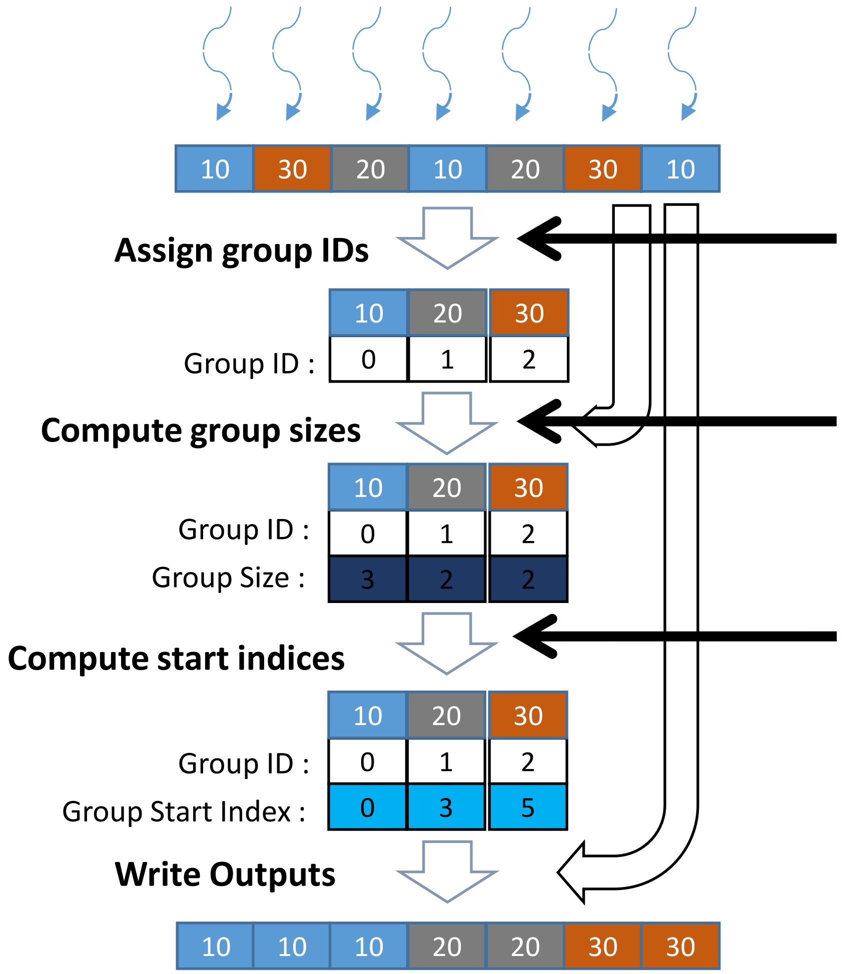
Write Outputs

10	10	10	20	20	30	30

Sorting or hashing

Hash table lookup: group ID
-- Uses atomic increment

GroupBy using parallel primitives



10	30	20	10	20	30	10

Assign group IDs

	10	20	30
Group ID :	0	1	2

Compute group sizes

	10	20	30
Group ID :	0	1	2
Group Size :	3	2	2

Compute start indices

	10	20	30
Group ID :	0	1	2
Group Start Index :	0	3	5

Write Outputs

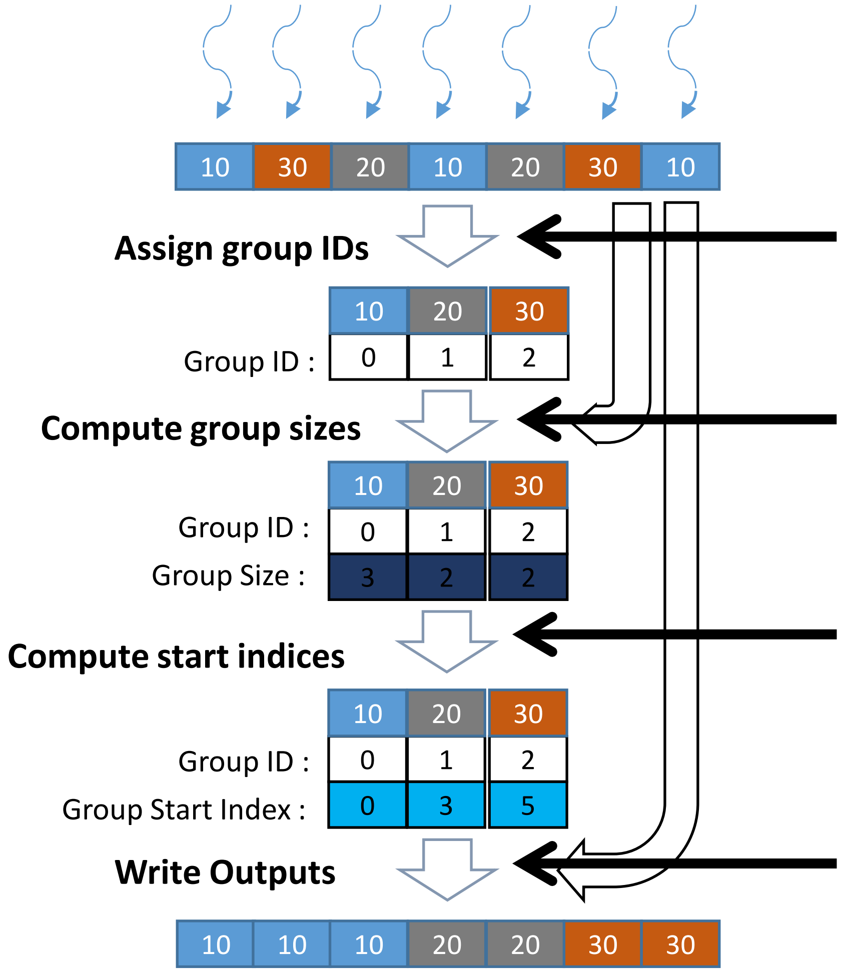
10	10	10	20	20	30	30

Sorting or hashing

Hash table lookup: group ID
-- Uses atomic increment

prefix sum of group sizes

GroupBy using parallel primitives



10	30	20	10	20	30	10

Assign group IDs

	10	20	30
Group ID :	0	1	2

Compute group sizes

	10	20	30
Group ID :	0	1	2
Group Size :	3	2	2

Compute start indices

	10	20	30
Group ID :	0	1	2
Group Start Index :	0	3	5

Write Outputs

10	10	10	20	20	30	30

Sorting or hashing

Hash table lookup: group ID
-- Uses atomic increment

prefix sum of groups sizes

Write to output location
- Uses atomic increment
- ***Scatter gather***