Parallel Architectures

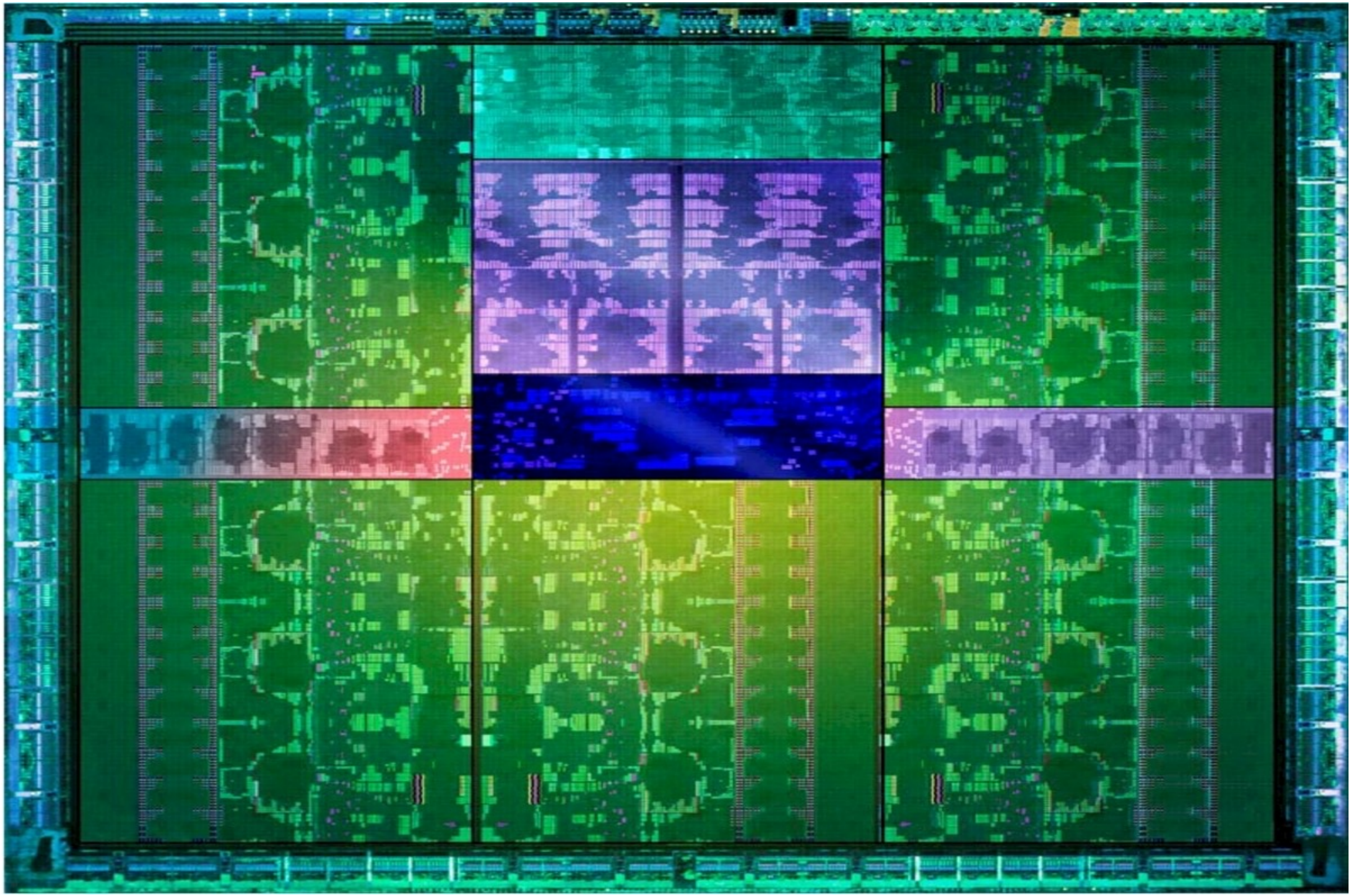
Chris Rossbach

Outline for Today

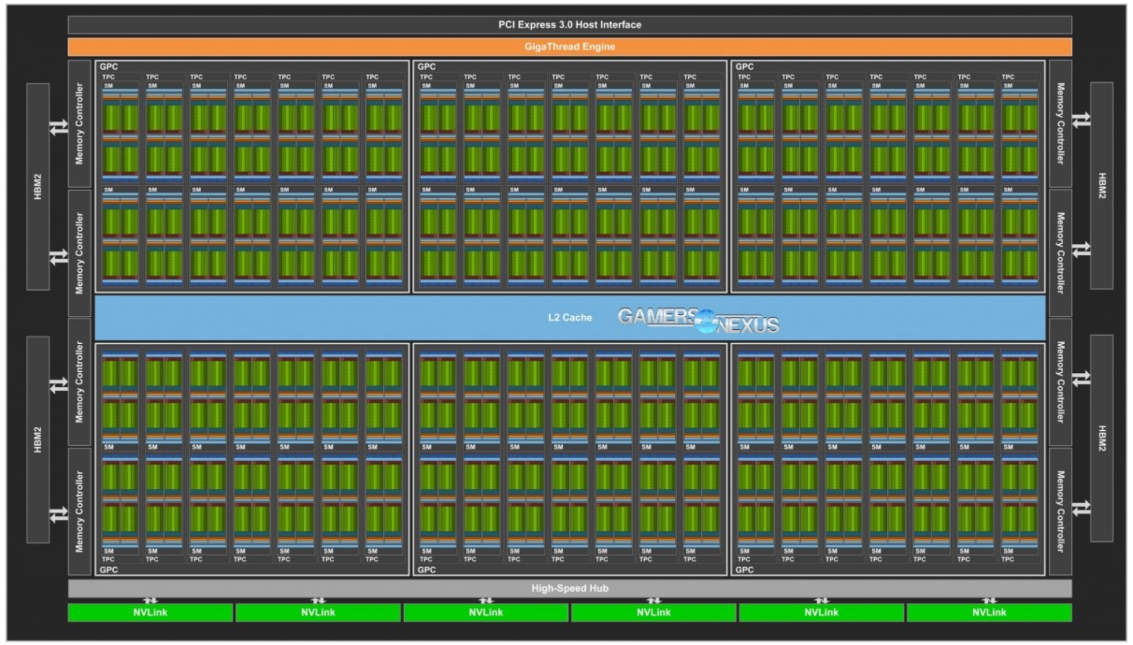
- Questions?
- Administrivia
    - Exam soon
- Agenda
    - Parallel Architectures (GPU background)

Faux Quiz questions

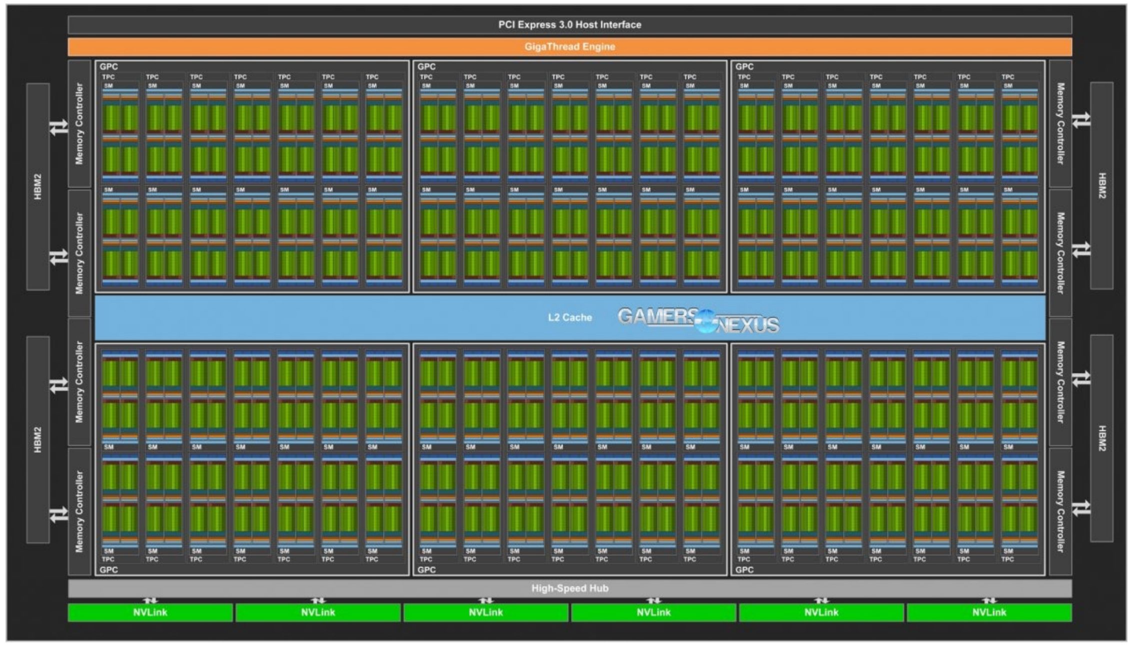
- What is hardware multi-threading; what problem does it solve?
- What is the difference between a vector processor and a scalar?
- Implement a parallel scan or reduction
- How are GPU workloads different from GPGPU workloads?
- How does SIMD differ from SIMT?
- List and describe some pros and cons of 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: Volta V100

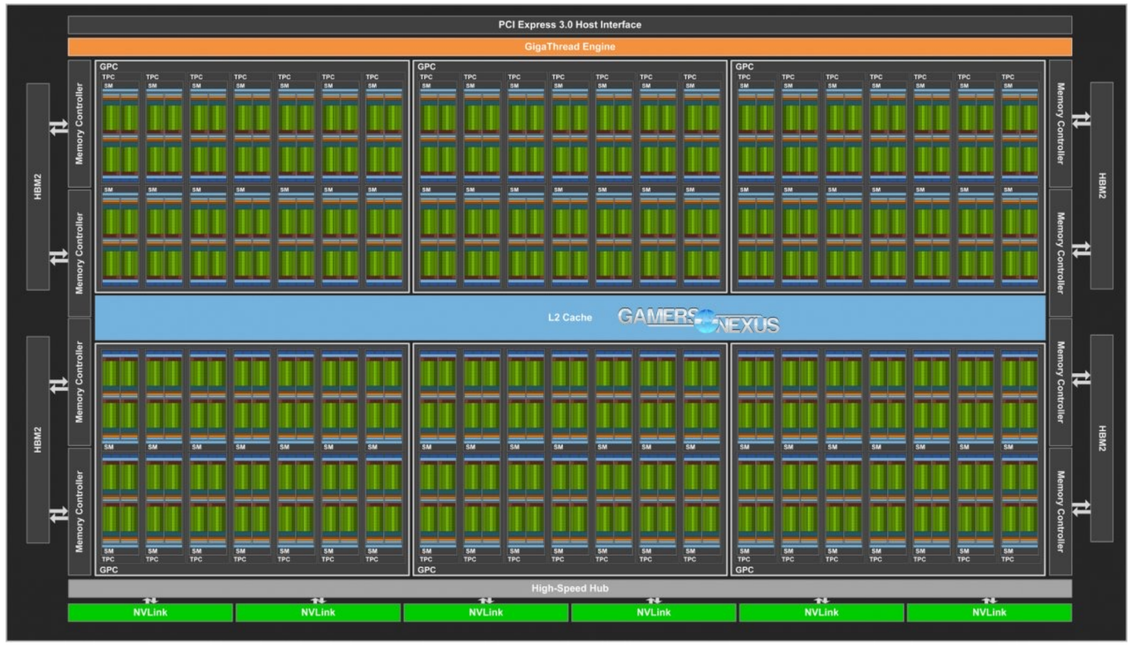


A modern GPU: Volta V100



- 80 SMs
  - Streaming Multiprocessor

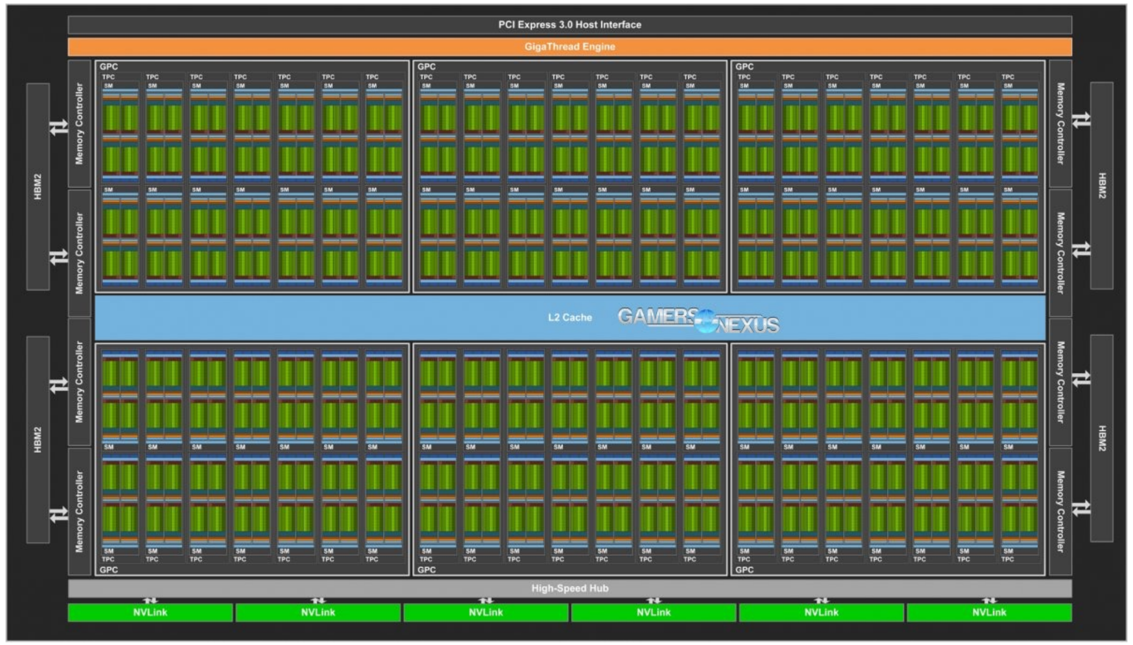
A modern GPU: Volta V100



- 80 SMs
  - Streaming Multiprocessor

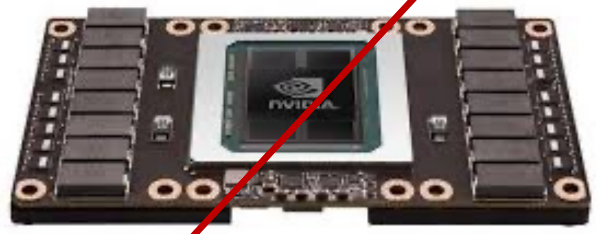
Also:
CU or *ACE*

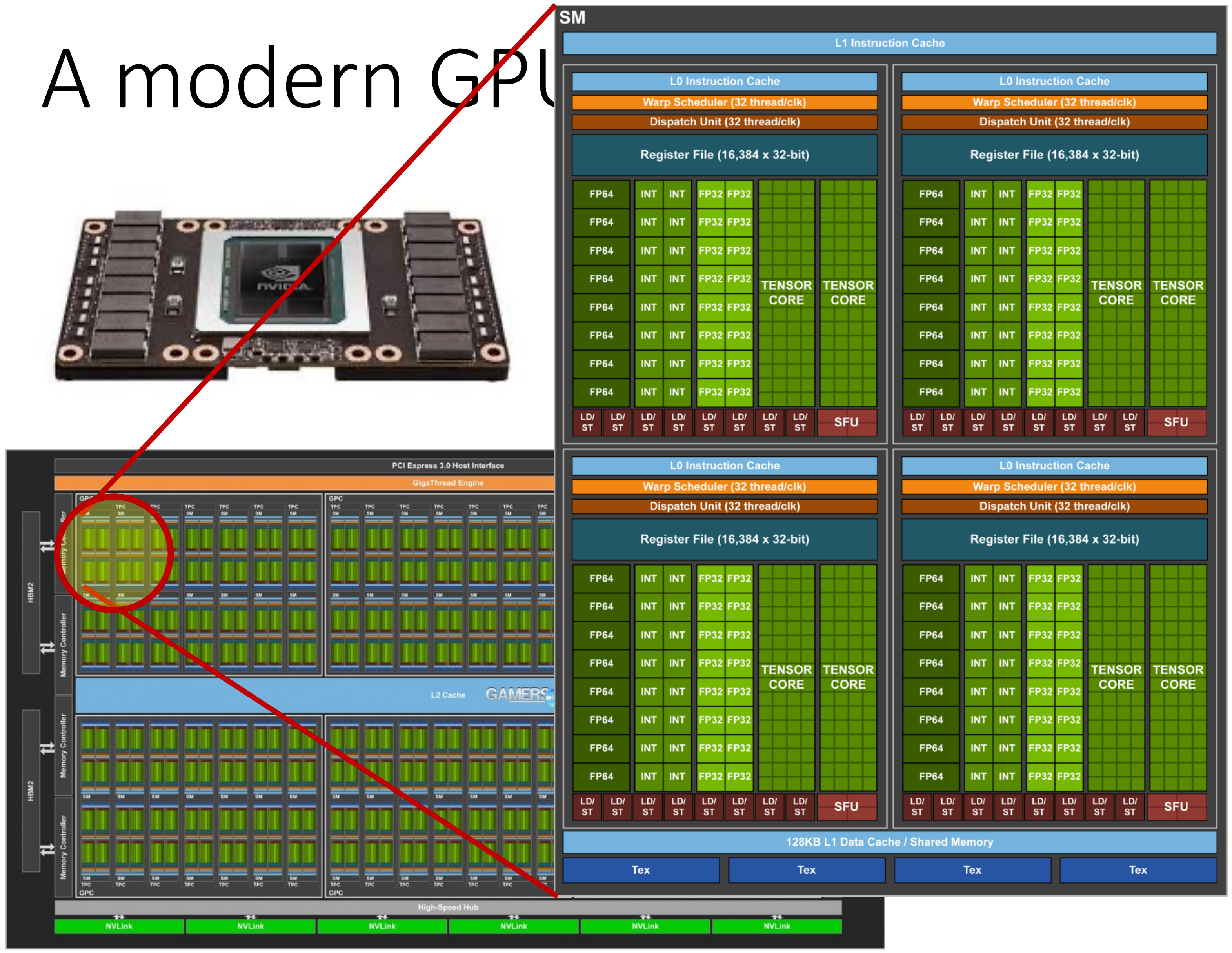
A modern GPU: Volta V100



- 80 SMs
  - Streaming Multiprocessor

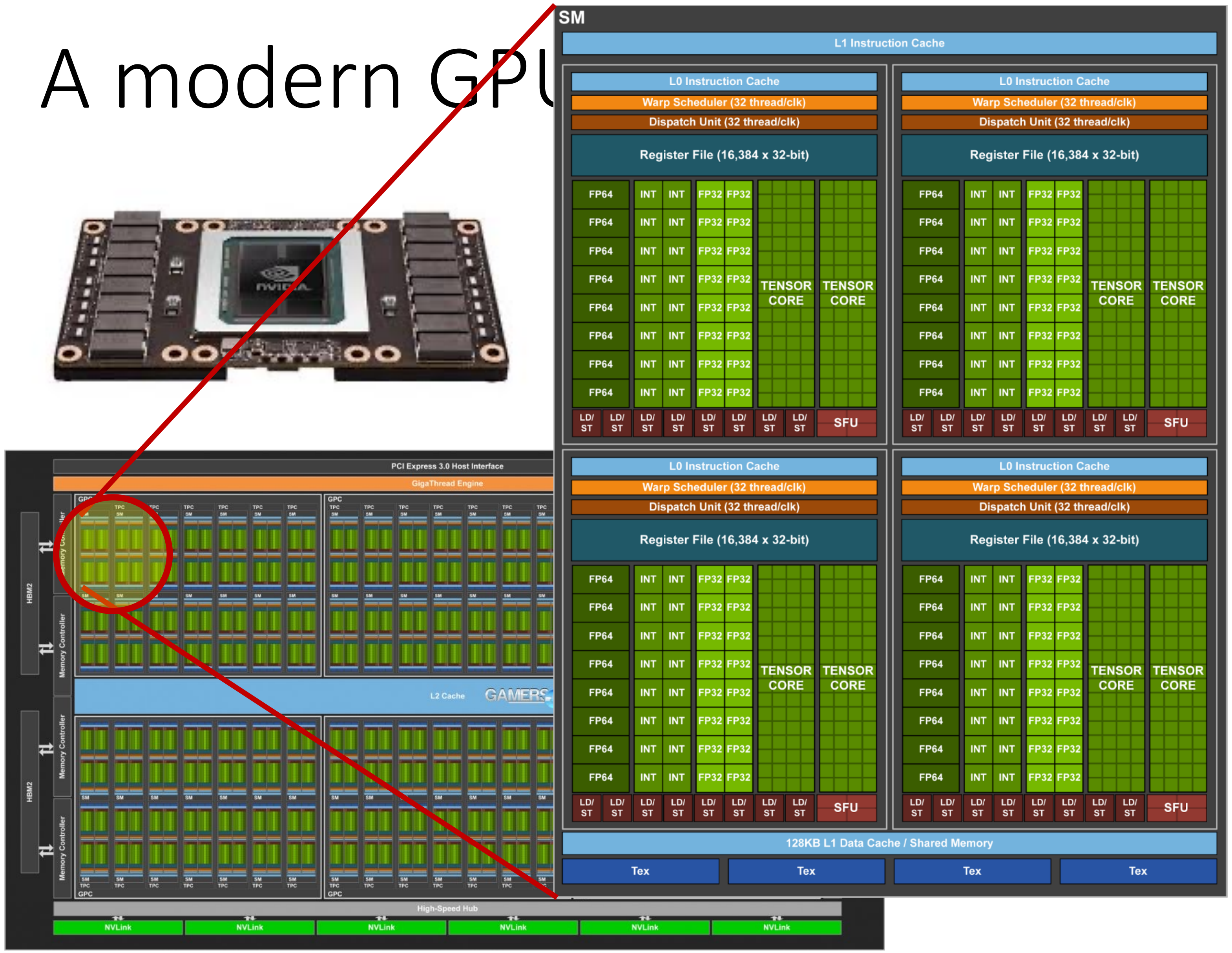
A modern GP




SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink

- 80 SMs
  - Streaming Multiprocessor
  - 64 cores/SM
  - 5210 threads!
  - 15.7 TFLOPS

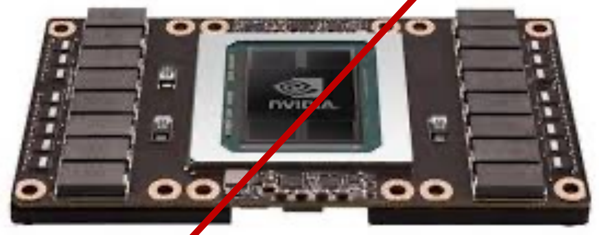
A modern GP

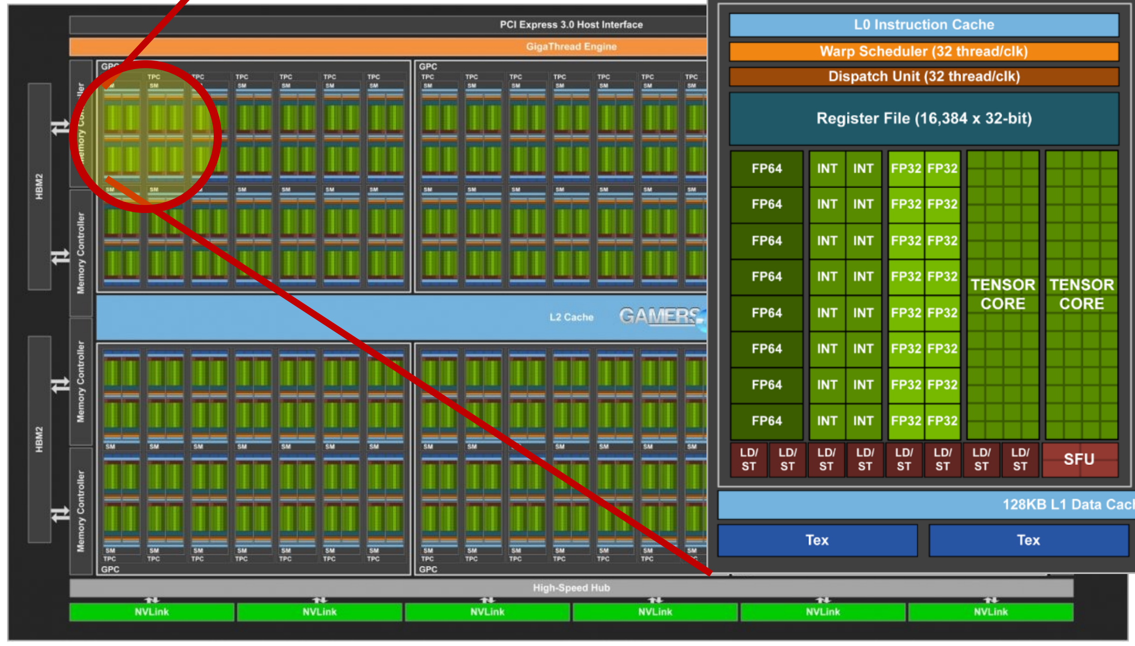

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Memory Controller
HBM2
L2 Cache
High-Speed Hub
NVLink

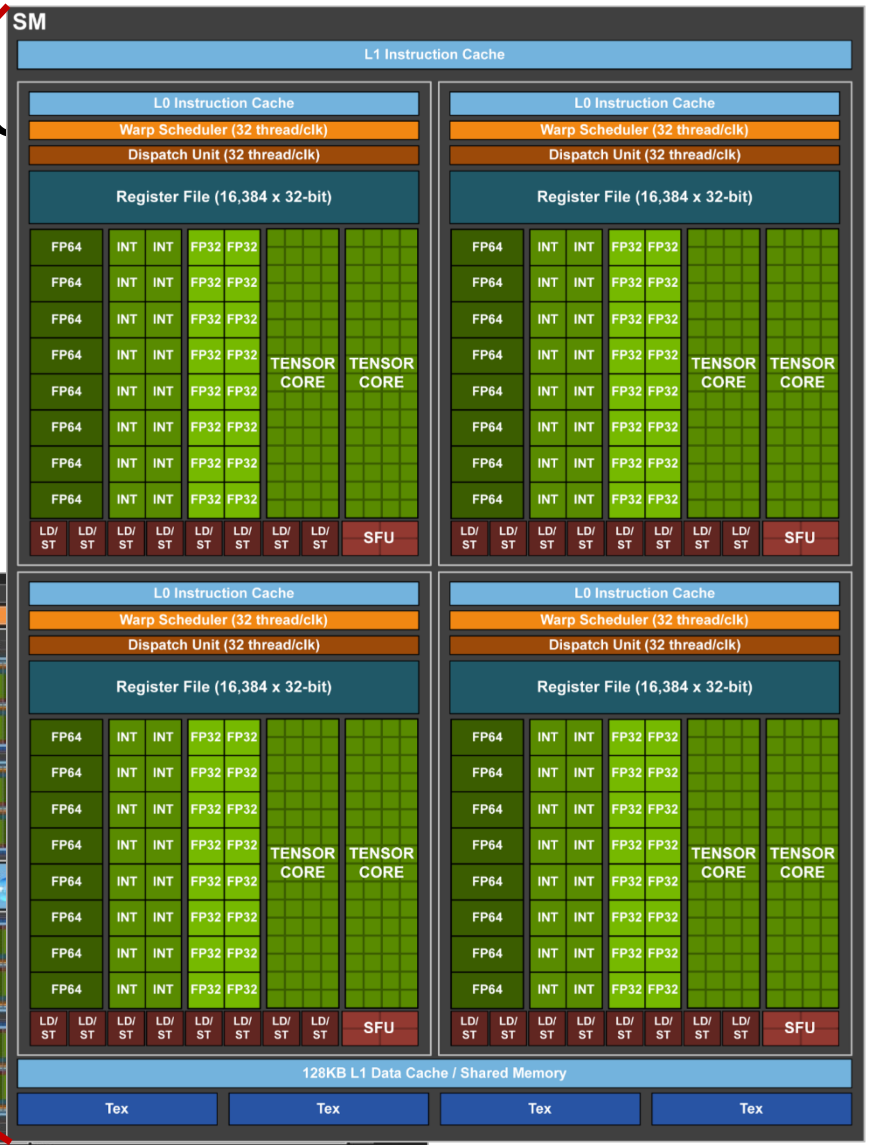
- 80 SMs
  - Streaming Multiprocessor
  - 64 cores/SM
  - 5210 threads!
  - 15.7 TFLOPS

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

A modern GP

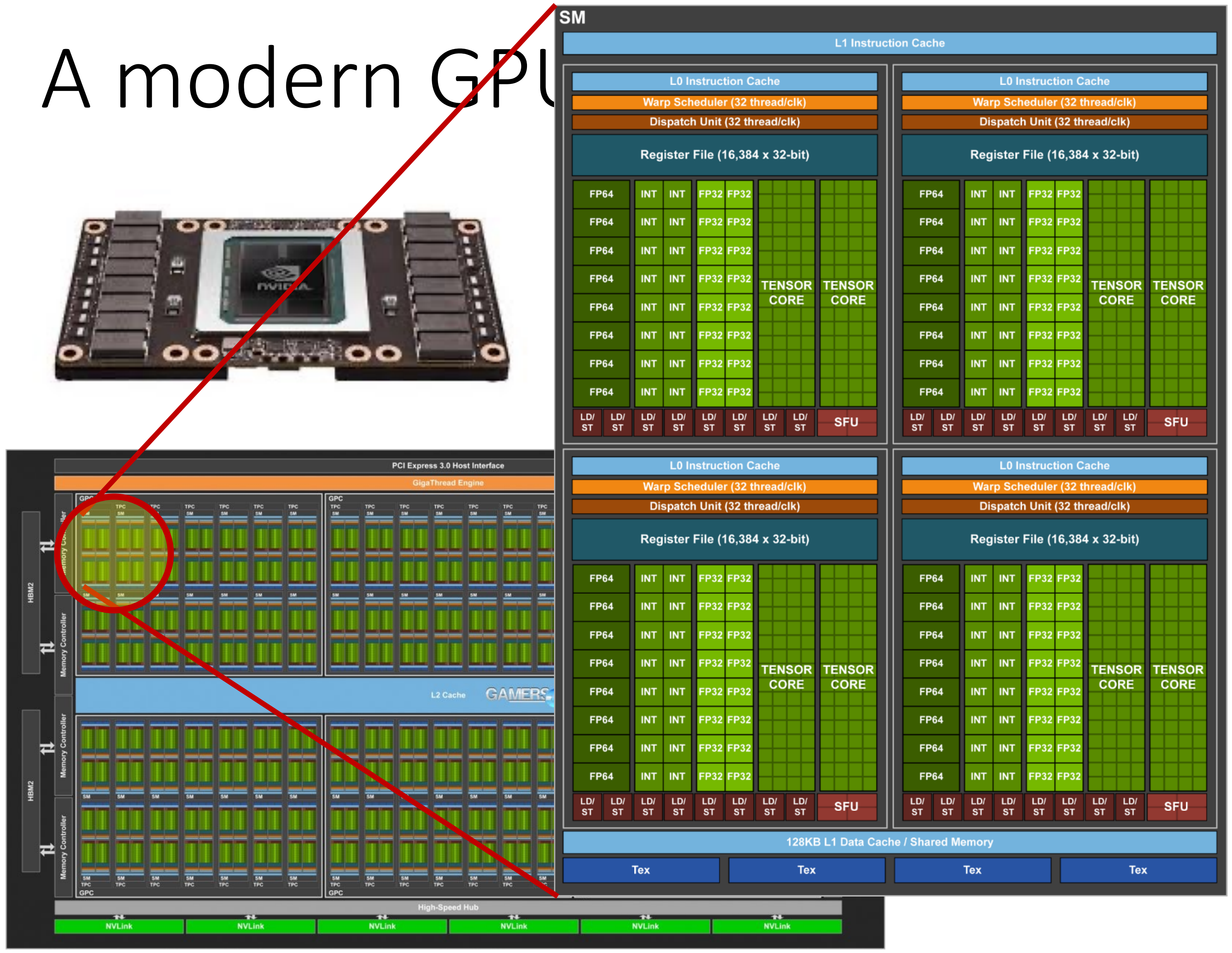





SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

- 80 SMs
  - Streaming Multiprocessor
  - 64 cores/SM
  - 5210 threads!
  - 15.7 TFLOPS

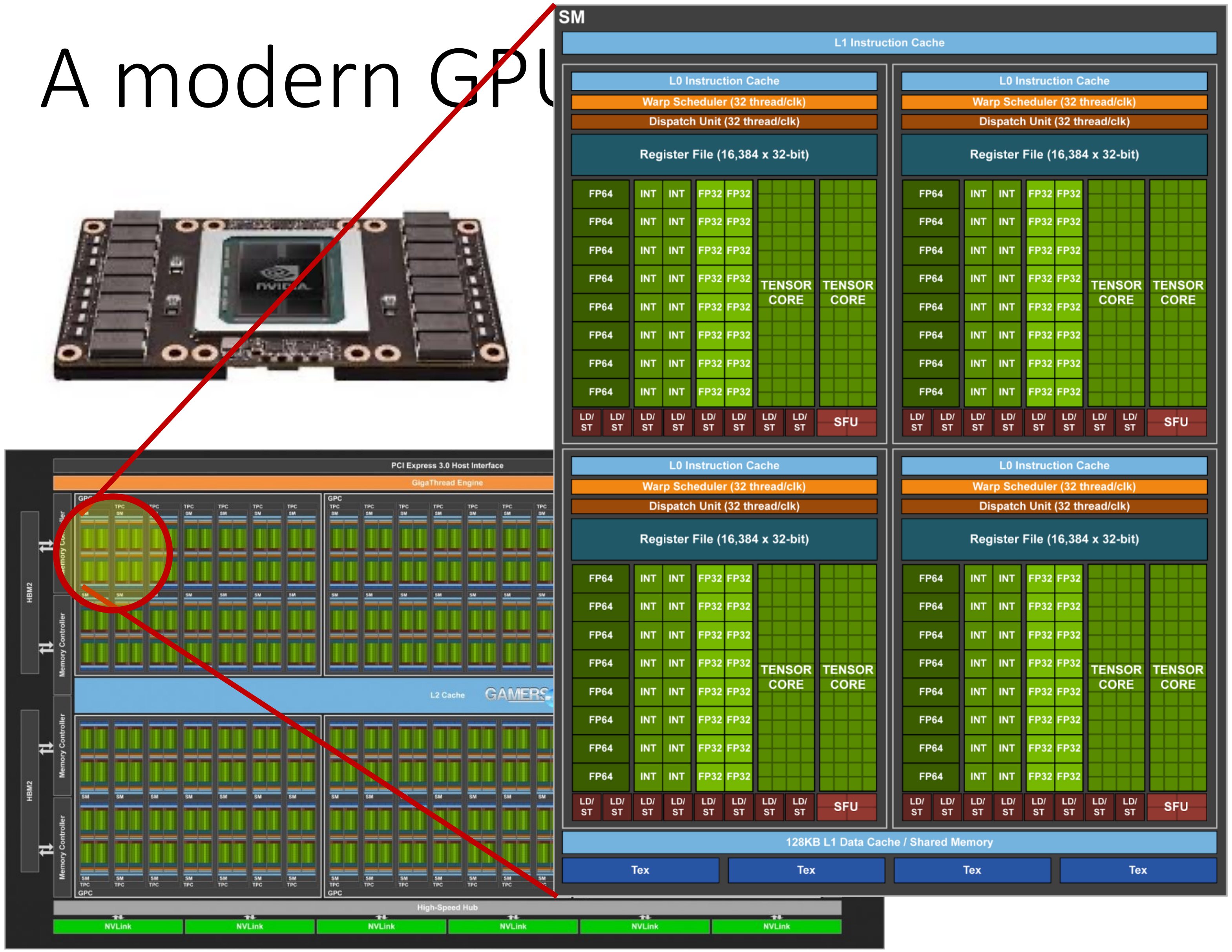
A modern GP



SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink

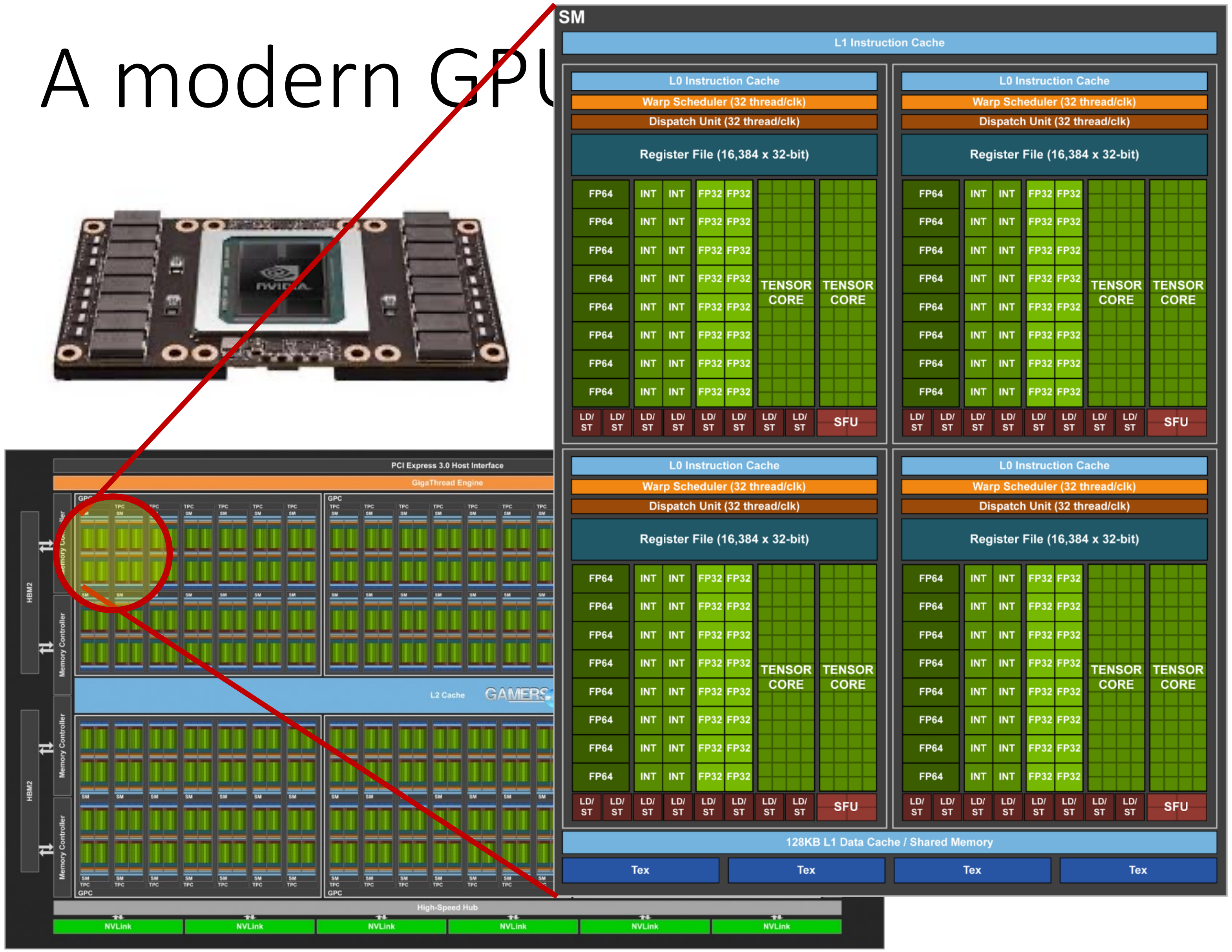
- 80 SMs
  - Streaming Multiprocessor
  - 64 cores/SM
  - 5210 threads!
  - 15.7 TFLOPS
- 640 Tensor cores

A modern GPU


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink

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

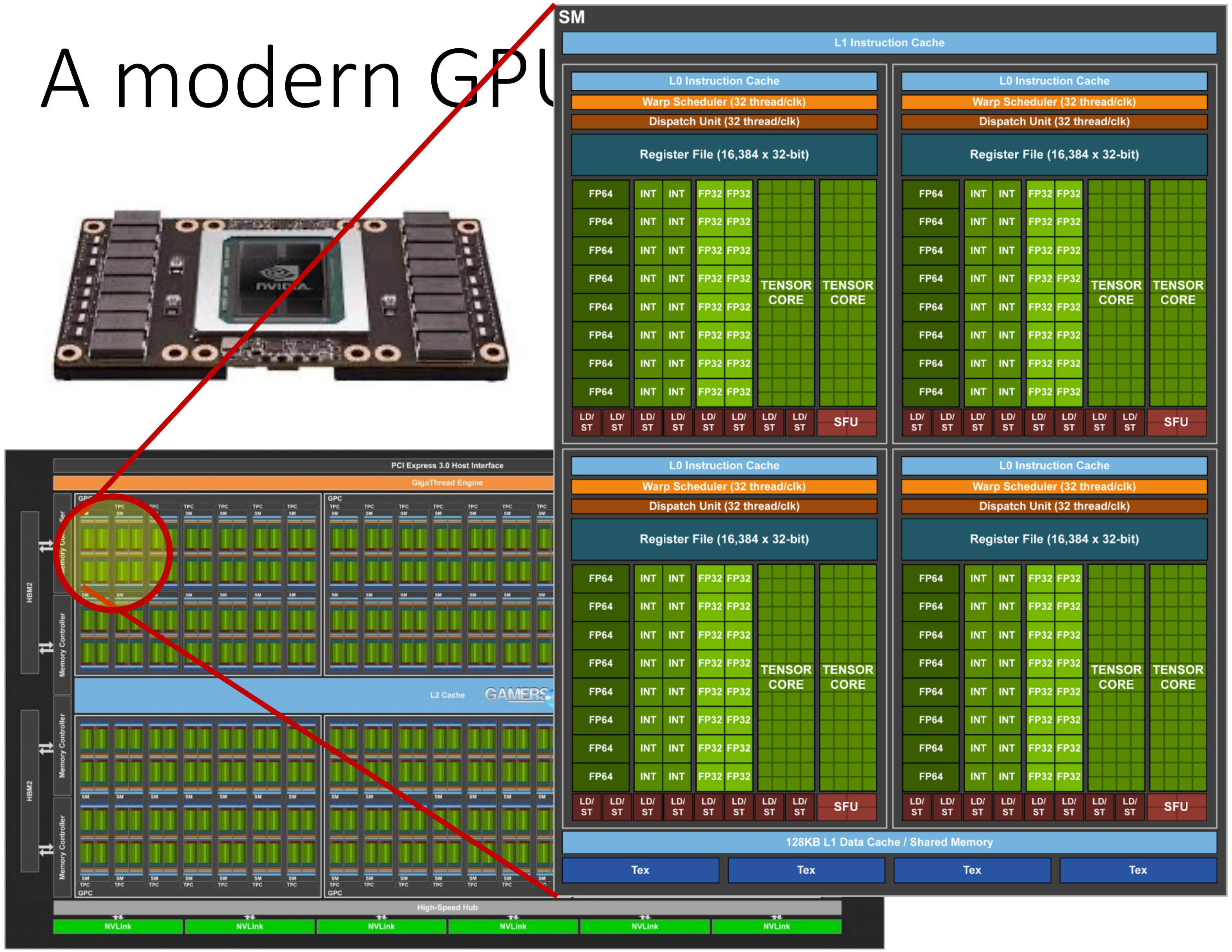
A modern GPU



SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink

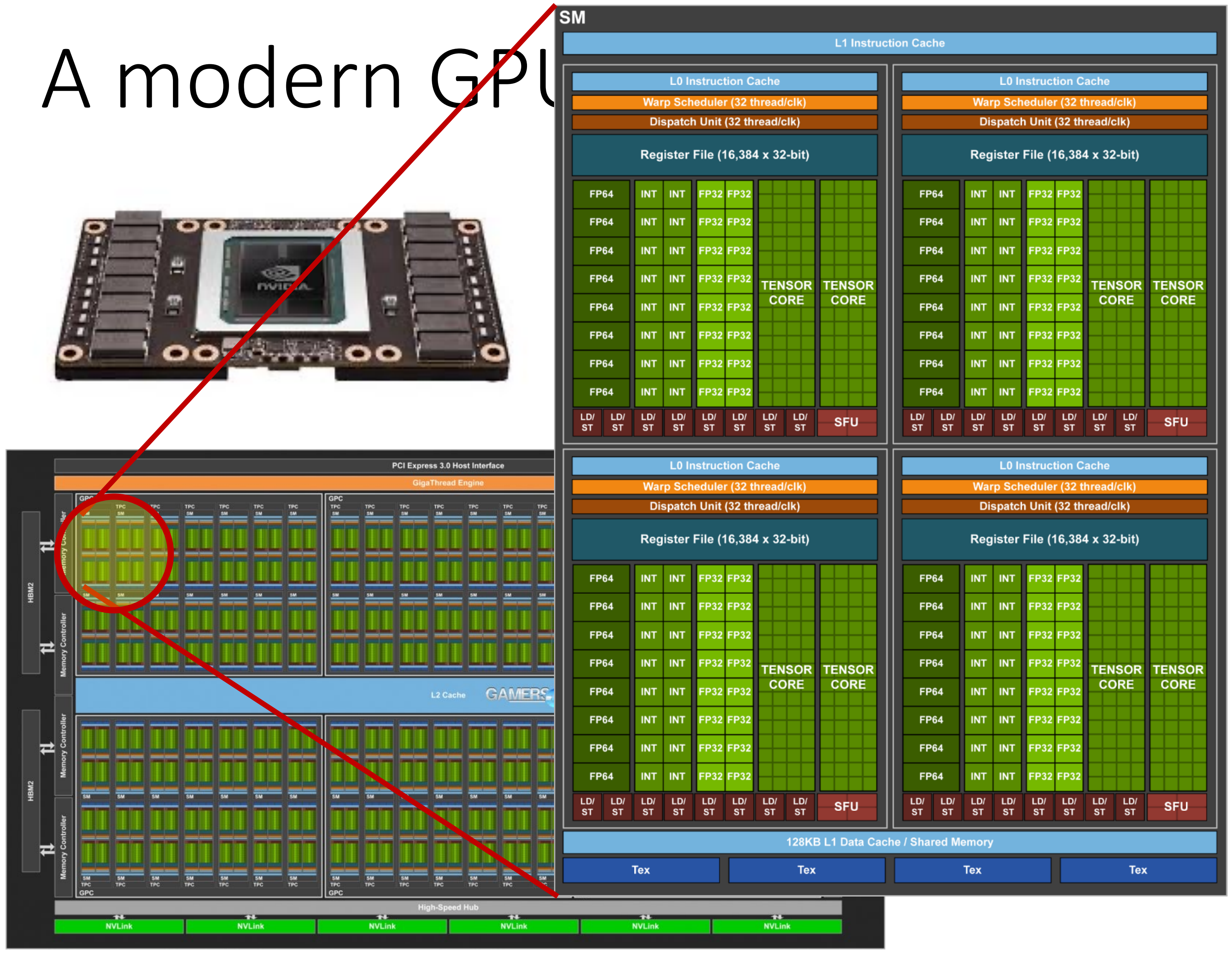
- 80 SMs
  - Streaming Multiprocessor
  - 64 cores/SM
  - 5210 threads!
  - 15.7 TFLOPS
- 640 Tensor cores
- HBM2 memory
  - 4096-bit bus
  - No cache coherence!
- 16 GB memory
  - PCIe-attached

A modern GPU


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Memory Controller
HBM2
L2 Cache
High-Speed Hub
NVLink

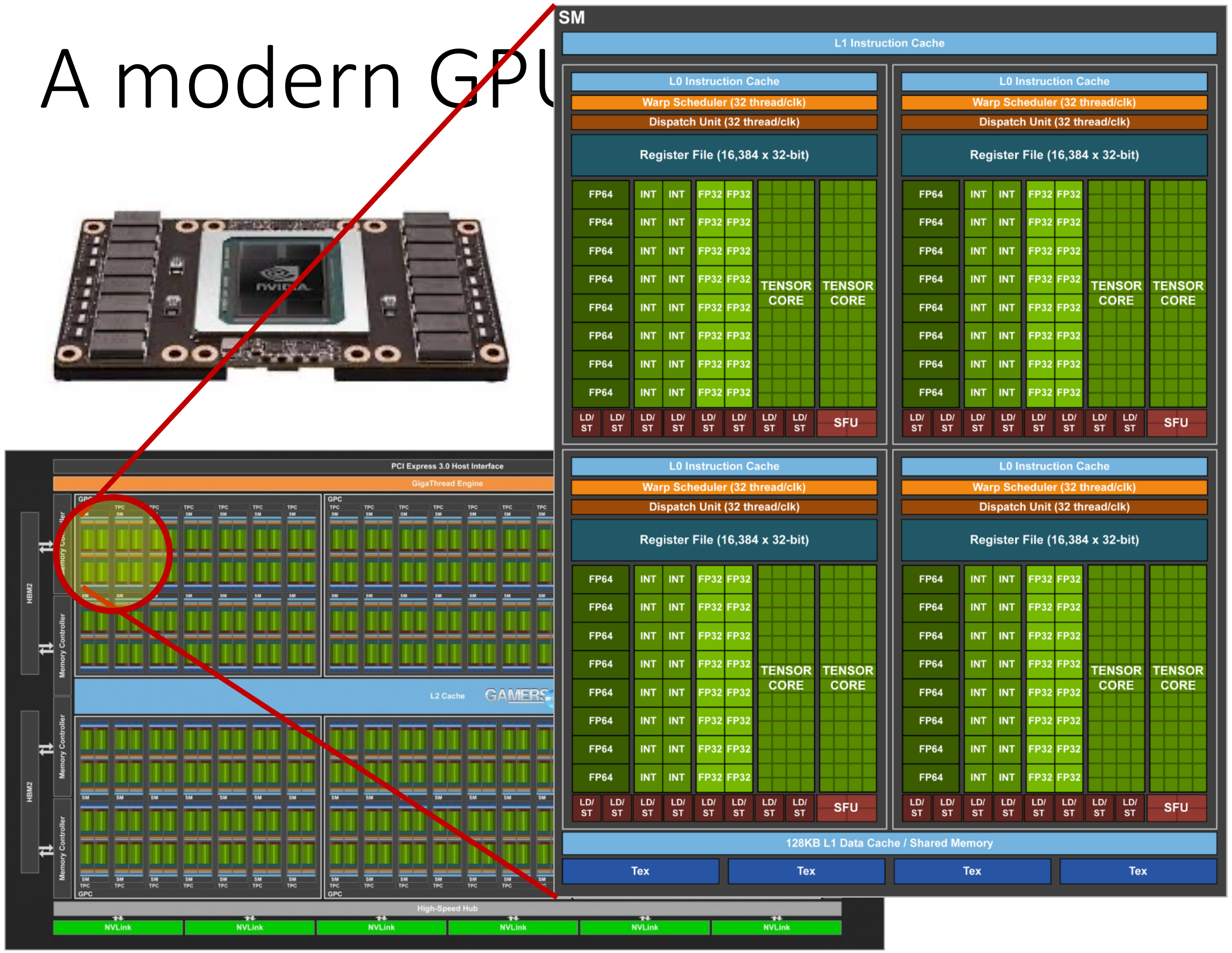
- 80 SMs
  - Streaming Multiprocessor
  - 64 cores/SM
  - 5210 threads!
  - 15.7 TFLOPS
- 640 Tensor cores
- HBM2 memory
  - 4096-bit bus
  - No cache coherence!
- 16 GB memory
  - PCIe-attached

A modern GP


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Memory Controller
HBM2
L2 Cache
High-Speed Hub
NVLink

- 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

A modern GPU


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Memory Controller
HBM2
L2 Cache
High-Speed Hub
NVLink

- 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 $\rightarrow$ parallel algorithms
- Programming GPUs
  - CUDA
  - Basics: getting something working
  - Advanced: making it perform

Architecture Review: Pipelines

Processor algorithm:

```
main() {
  while(true)
    do_next_instruction();
}
```

Architecture Review: Pipelines

Processor algorithm:

```
main() {
  while(true)
    do_next_instruction();
}
```

```
do_next_instruction() {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Architecture Review: Pipelines

Processor algorithm:

```
    main() {
      v
    
    }



  do_n
    inst
    ops
    exe
    access_memory(ops, regs);
    write_back(regs);
  }
```

```
main() {
  pthread_create(do_instructions);
  pthread_create(do_decode);
  pthread_create(do_execute);
  …
  pthread_join(… );
  …
}
```

Architecture Review: Pipelines

Processor algorithm:

```
main() {
  v
}

do_n
  inst
  ops
  exe
  access_memory(ops, regs);
  write_back(regs);
}
```

```
main() {
  pthread_create(do_instructions);
  pthread_create(do_decode);
  pthread_create(do_execute);
  …
  pthread_join(… );
  …
}
```

```
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_addrs(ops, regs);
    enqueue(MEM, instruction);
}}

….
```

Architecture Review: Pipelines

Processor algorithm:

```
main() {
  while(true) {
    do_next_instruction();
}
```

Architecture Review: Pipelines

Processor algorithm:

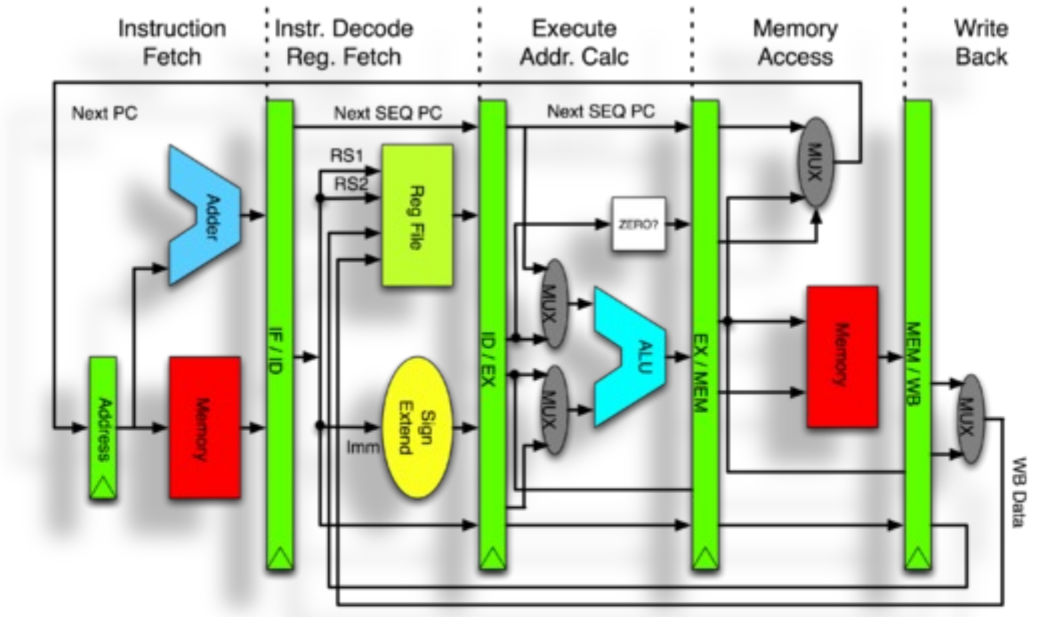
```
main() {
  while(true) {
    do_next_instruction();
  }
```

```
do_next_instruction() {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Architecture Review: Pipelines

Processor algorithm:

```
main() {
  while(true) {
    do_next_instruction();
}
```



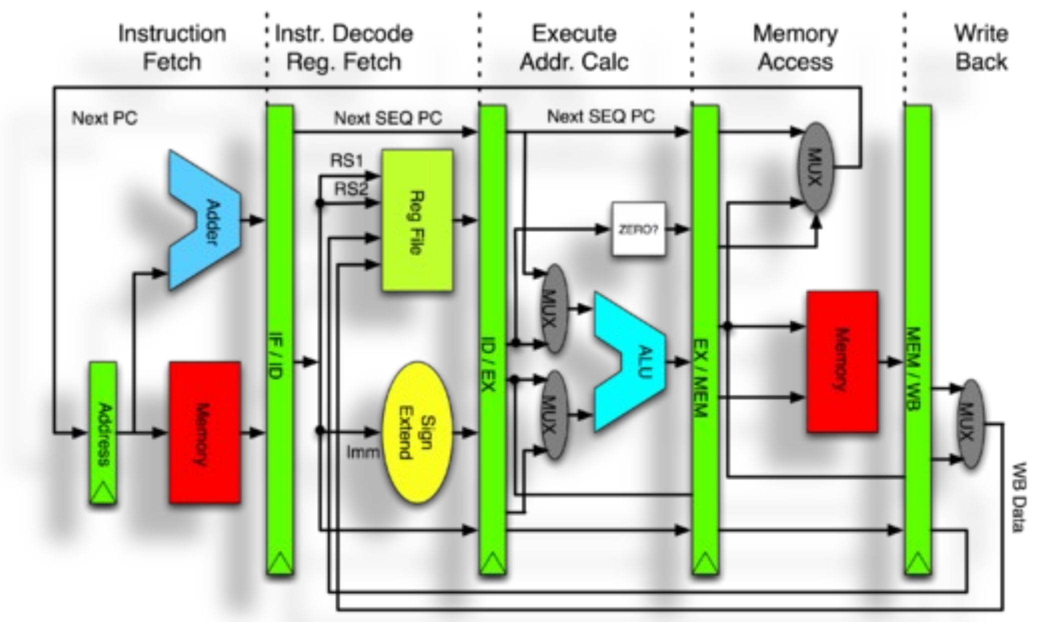
```
do_next_instruction() {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

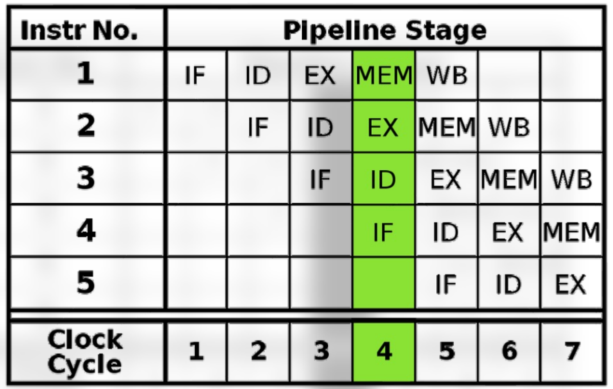
Architecture Review: Pipelines

Processor algorithm:

```
main() {
  while(true) {
    do_next_instruction();
}
```

```
do_next_instruction() {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```





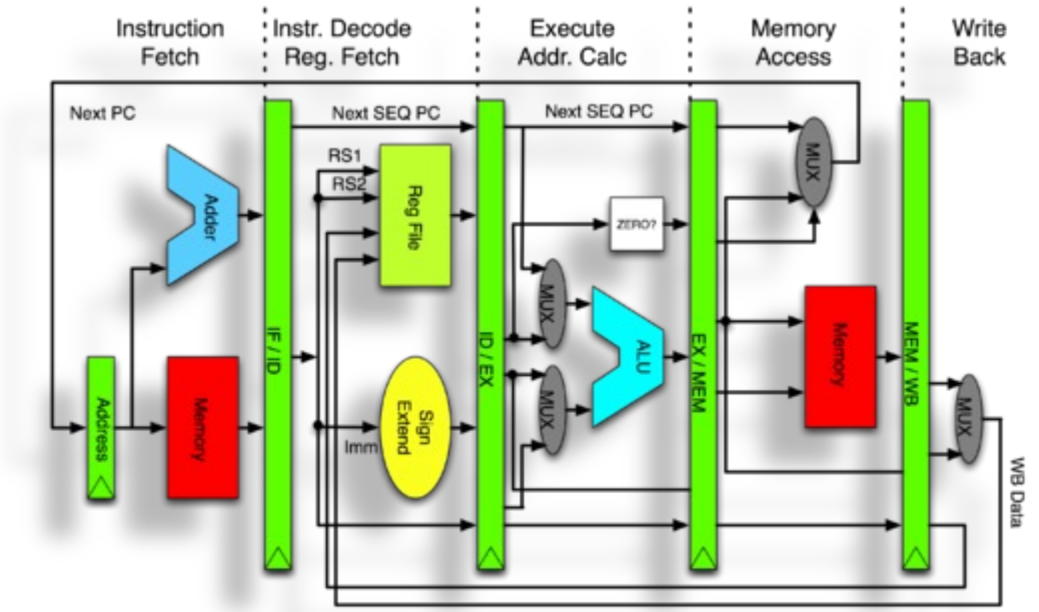
Instr No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

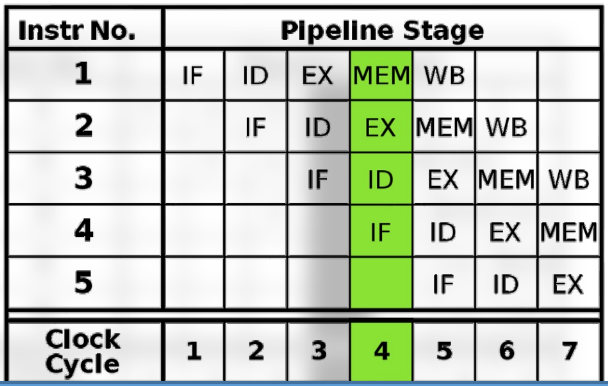
Architecture Review: Pipelines

Processor algorithm:

```
main() {
  while(true) {
    do_next_instruction();
}
```

```
do_next_instruction() {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```





Instr No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

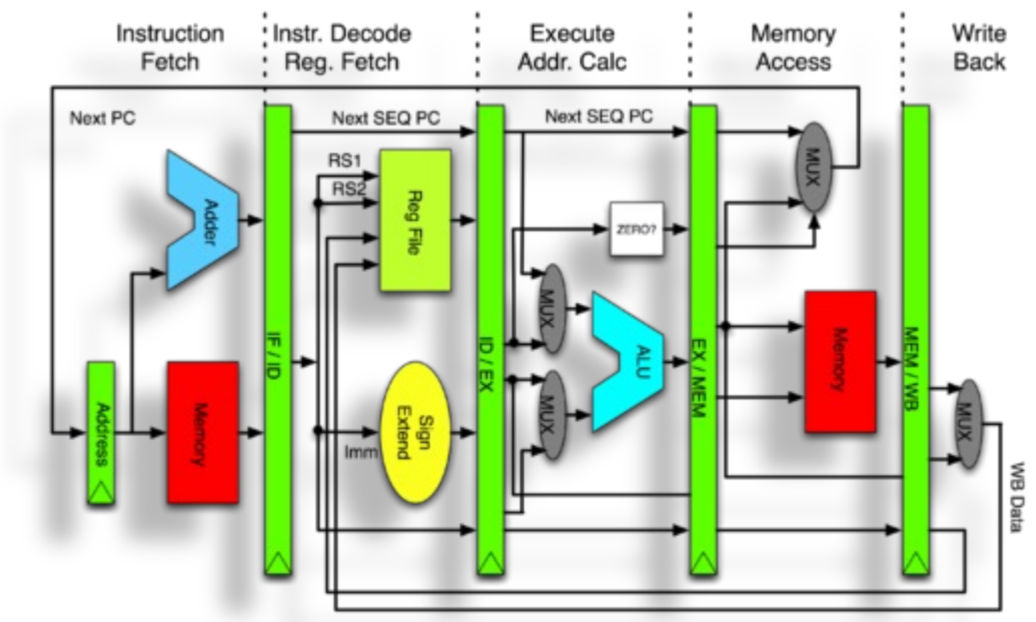
What is the name of this kind of parallelism?

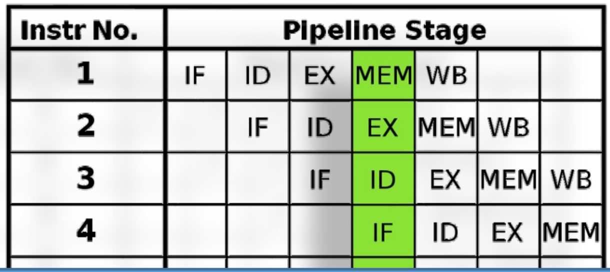
Architecture Review: Pipelines

Processor algorithm:

```
main() {
  while(true) {
    do_next_instruction();
}
```

```
do_next_instruction() {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```





Instr No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM

Works well if pipeline is kept full
What kinds of things cause "bubbles"/stalls?

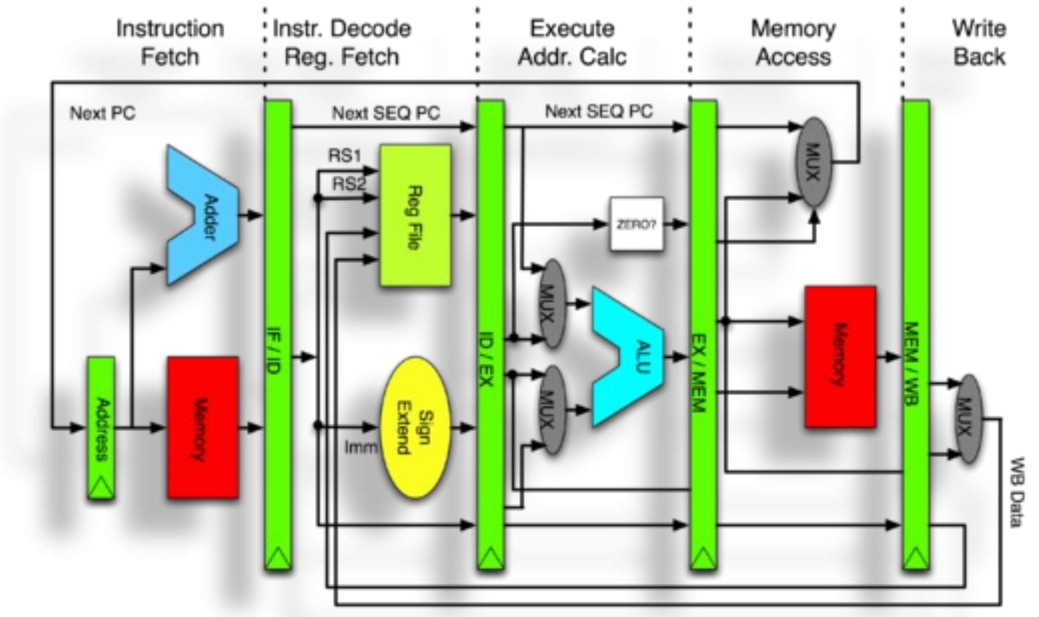
What is the name of this kind of parallelism?

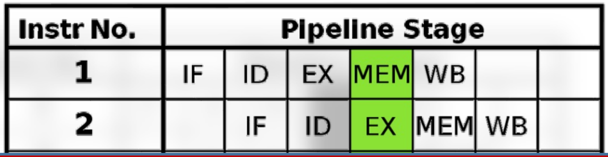
Architecture Review: Pipelines

Processor algorithm:

```
main() {
  while(true) {
    do_next_instruction();
}
```

```
do_next_instruction() {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```



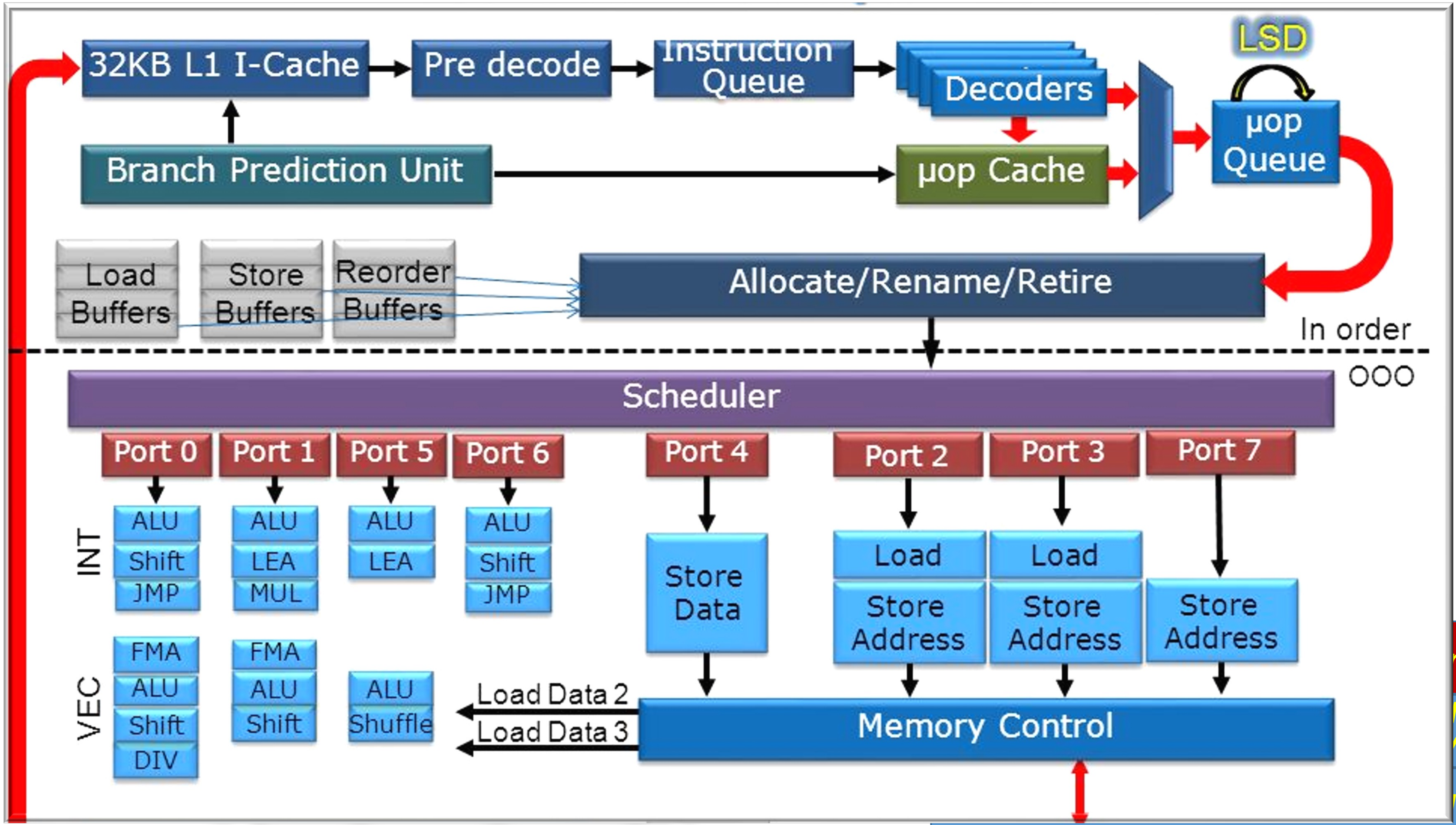


Instr No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	

How can we get *more* parallelism?

Works well if pipeline is kept full
What kinds of things cause "bubbles"/stalls?

What is the name of this kind of parallelism?



m?

/stalls?

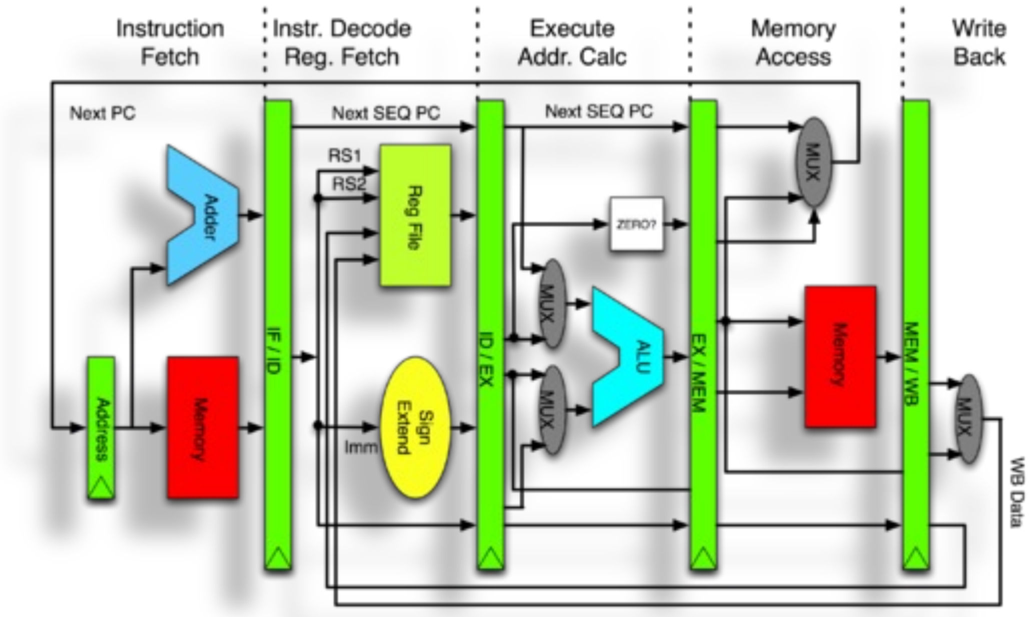
lelism?

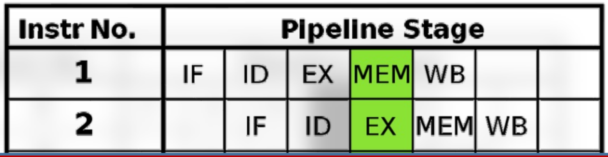
Architecture Review: Pipelines

Processor algorithm:

```
main() {
  while(true) {
    do_next_instruction();
}
```

```
do_next_instruction() {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```





Instr No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	

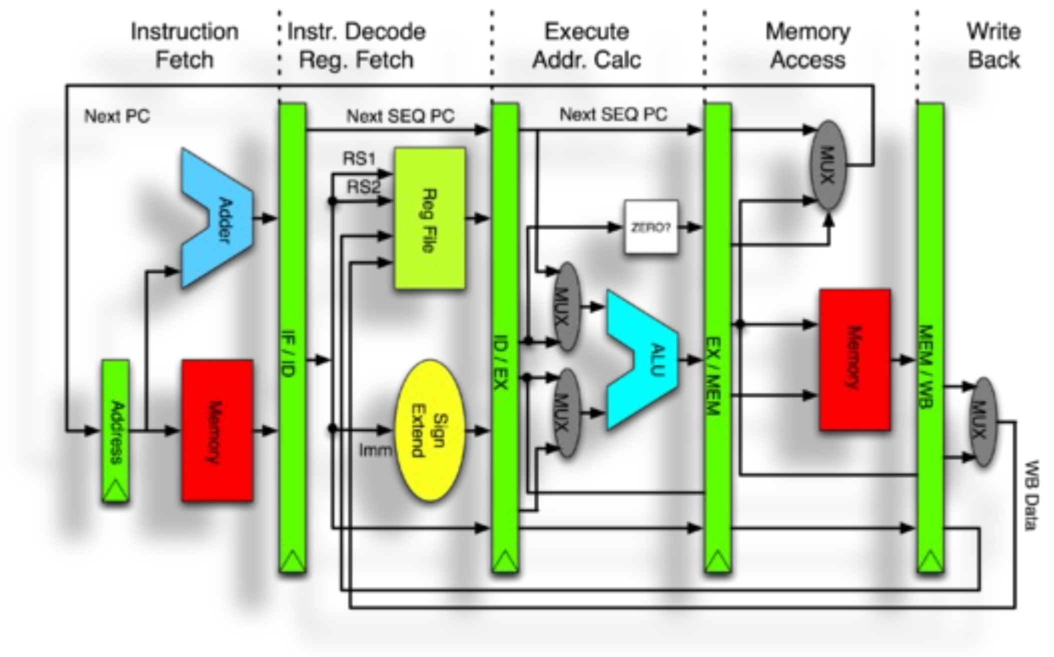
How can we get *more* parallelism?

Works well if pipeline is kept full
What kinds of things cause "bubbles"/stalls?

What is the name of this kind of parallelism?

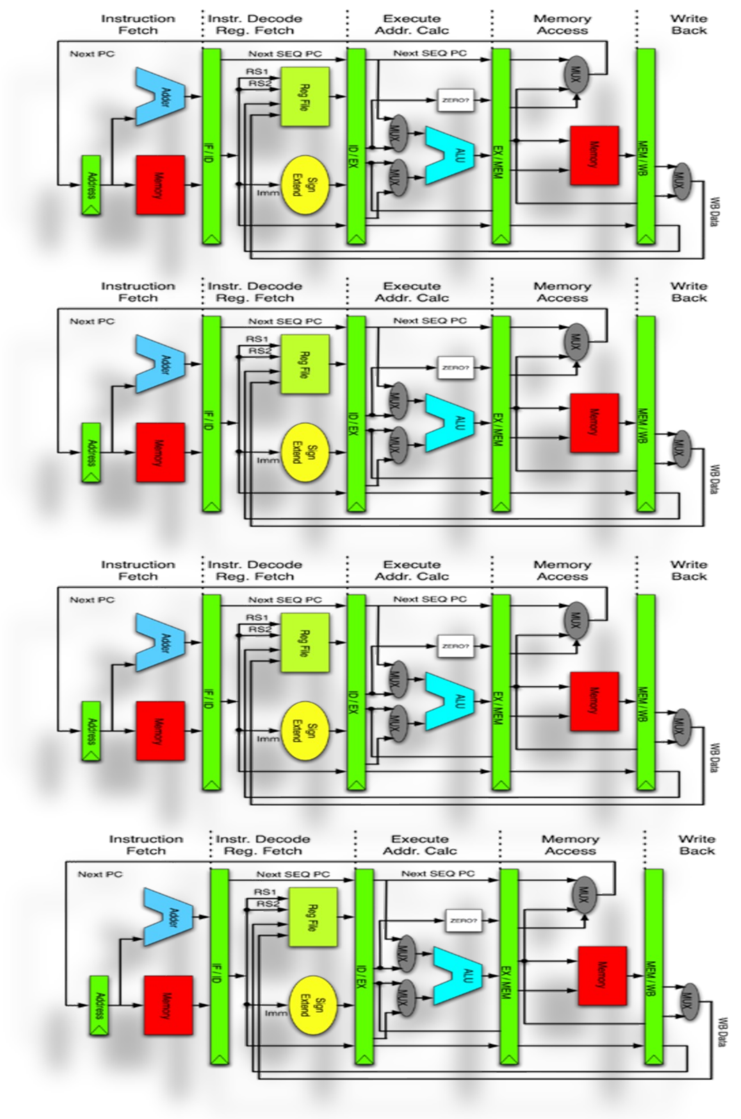
Multi-core/SMPs

Multi-core/SMPs

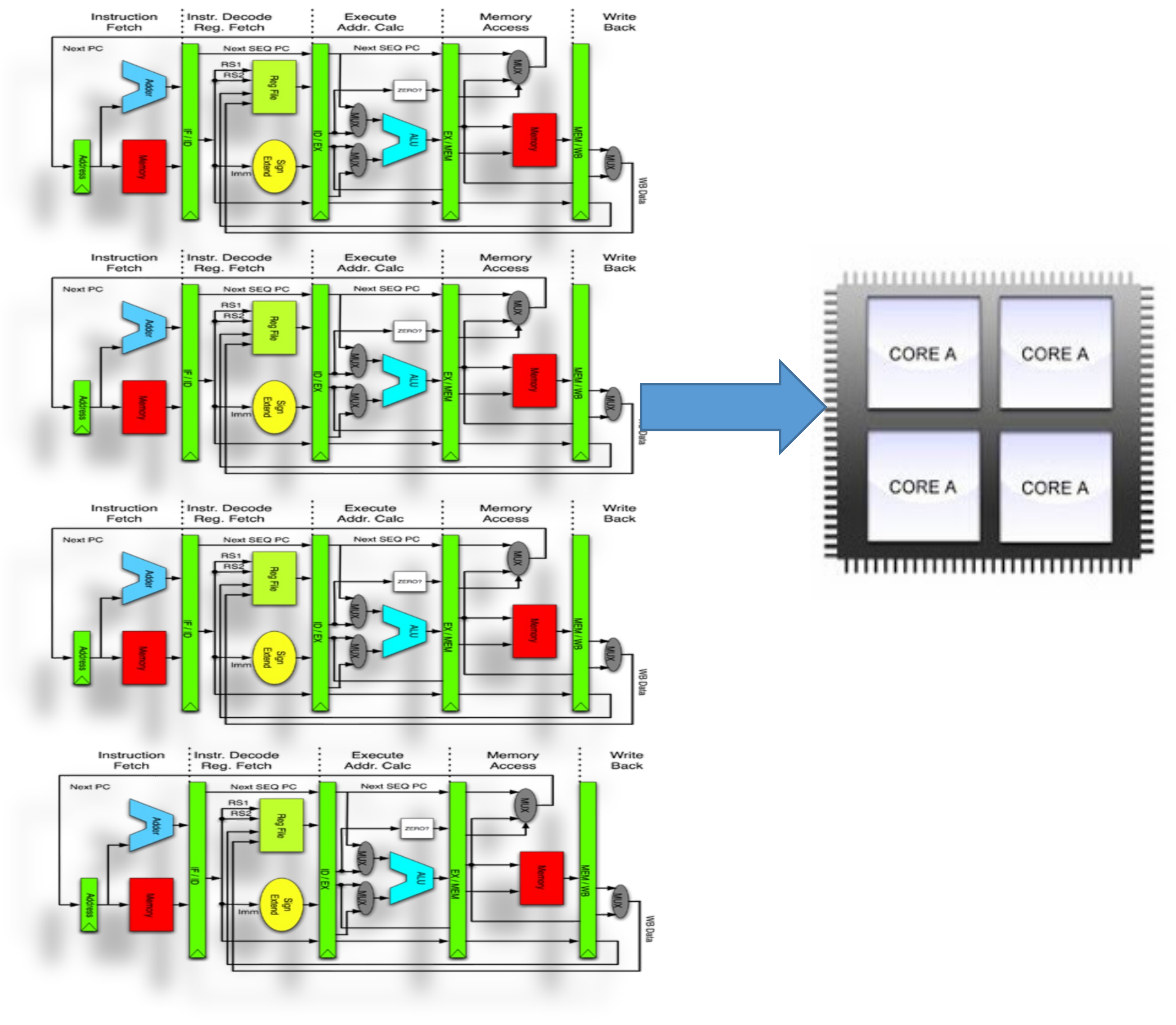


Multi-core/SMPs

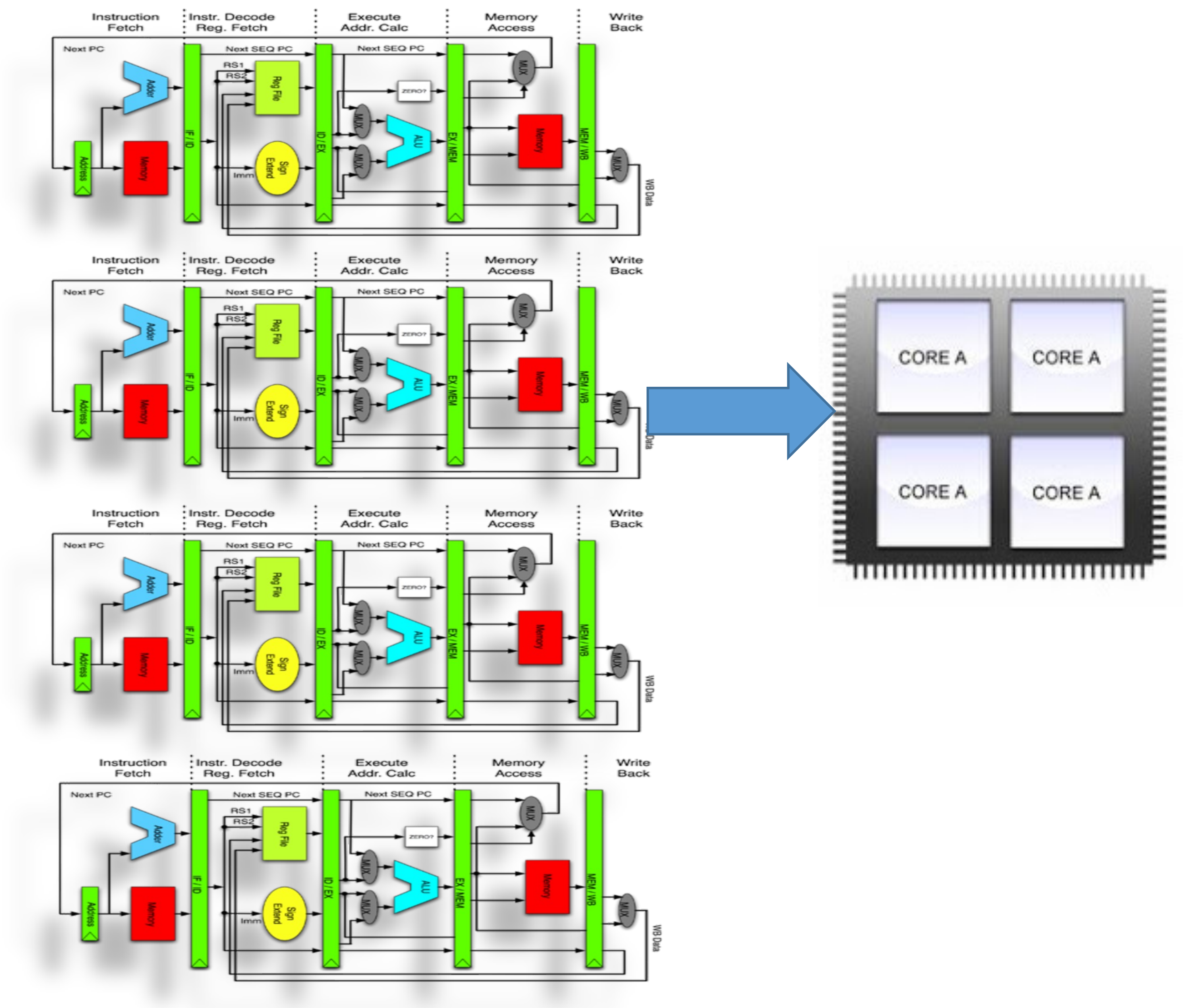
Multi-core/SMPs



Multi-core/SMPs



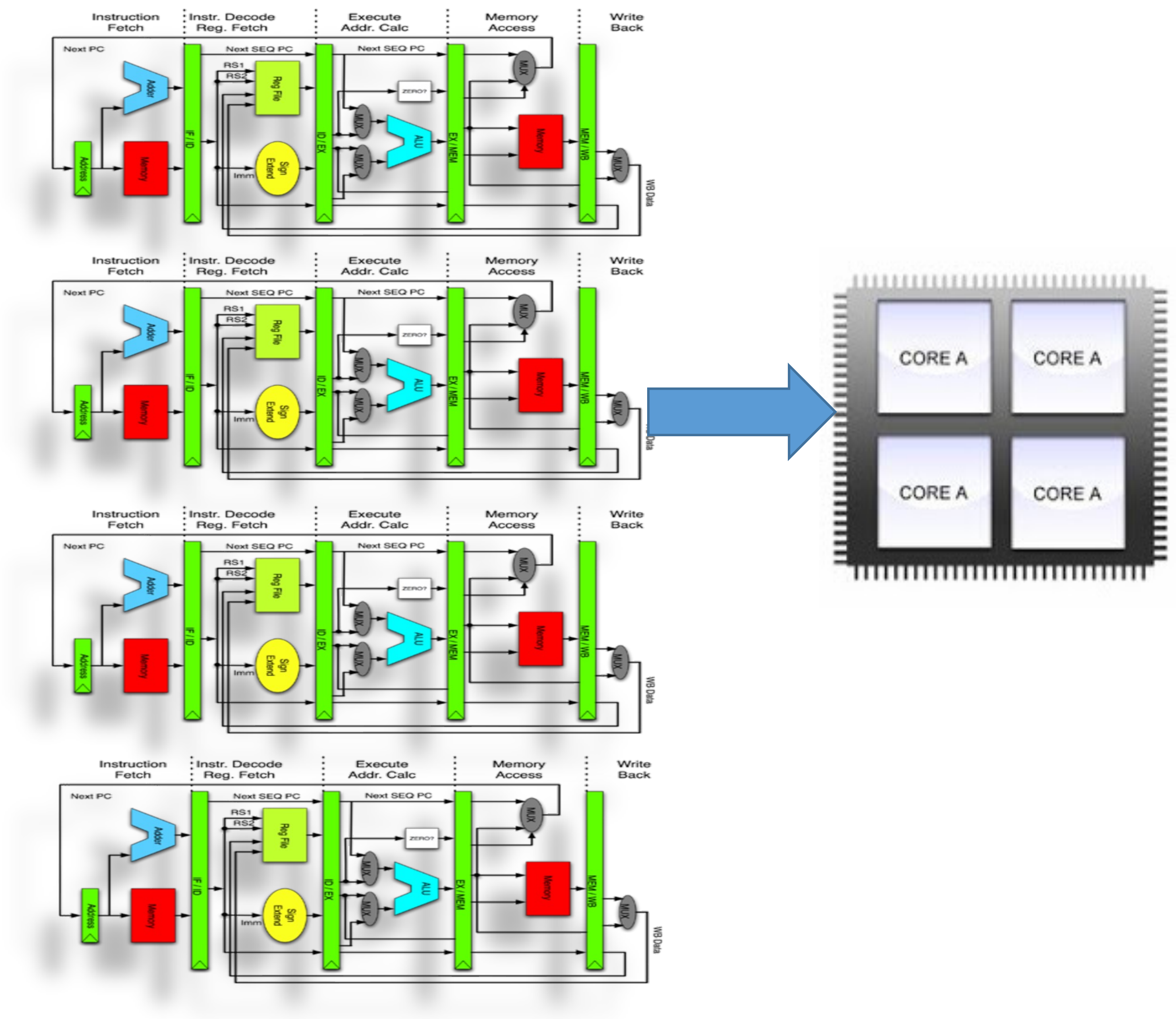
Multi-core/SMPs



```
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);
}}
```

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

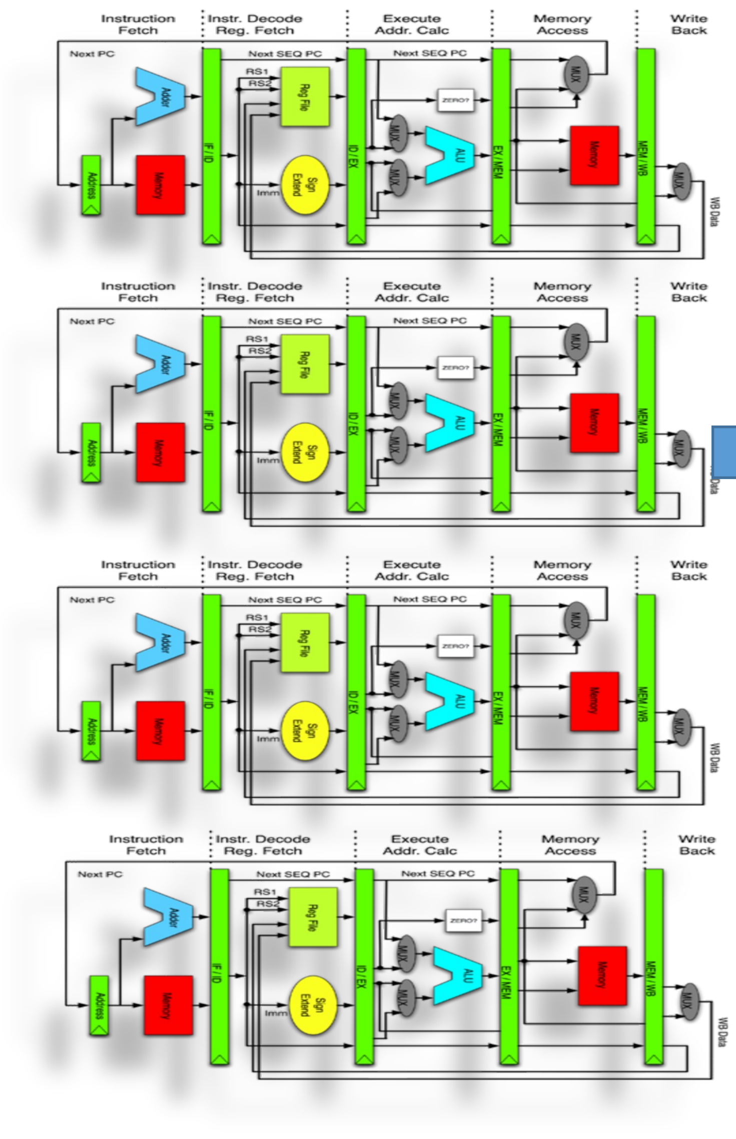
Multi-core/SMPs

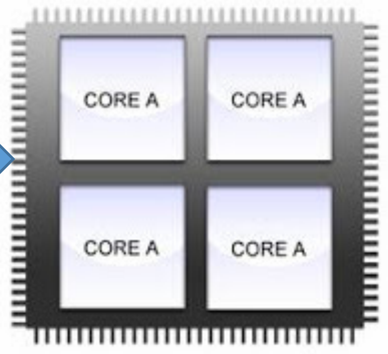


```
main() {
 for(i=0; i<CORES; i++) {
  pthread_create(
     do_instructions());
  }
}
do_instructions() {
 while(true) {
  instruction = fetch();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}}
```

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

Multi-core/SMPs

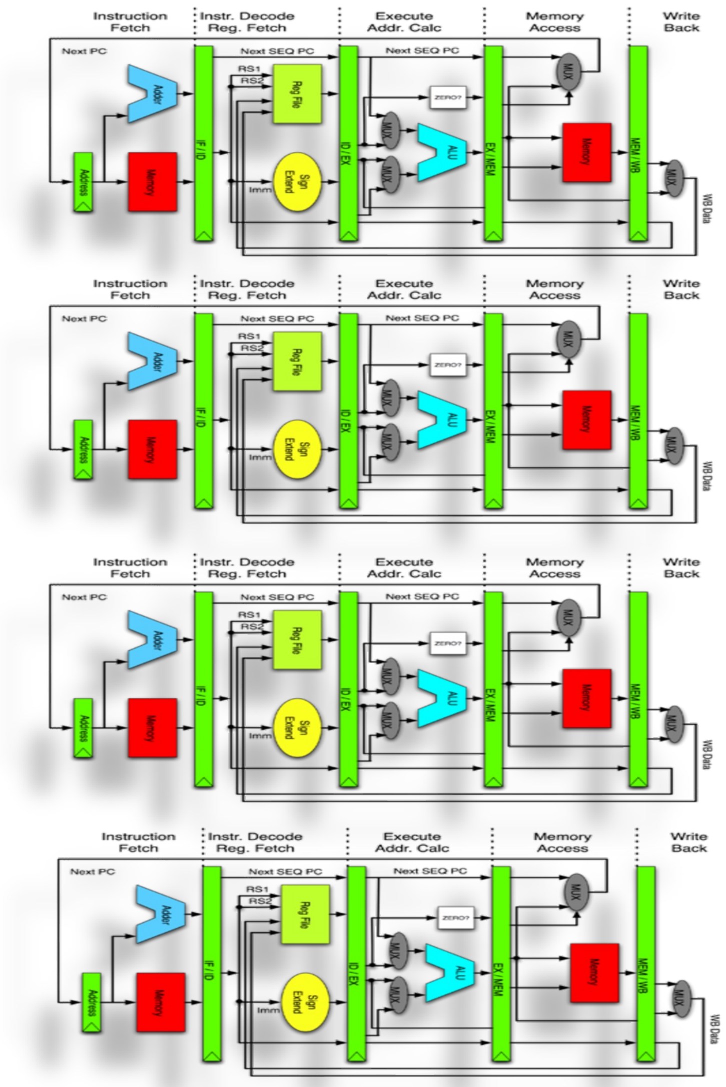




```
main() {
  for(i=0; i<CORES; i++) {
    pthread_create(
        do_instructions());
   }
}
do_instructions() {
  while(true) {
    instruction = fetch();
    ops, regs = decode(instruction);
    execute_calc_addrs(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}}
```

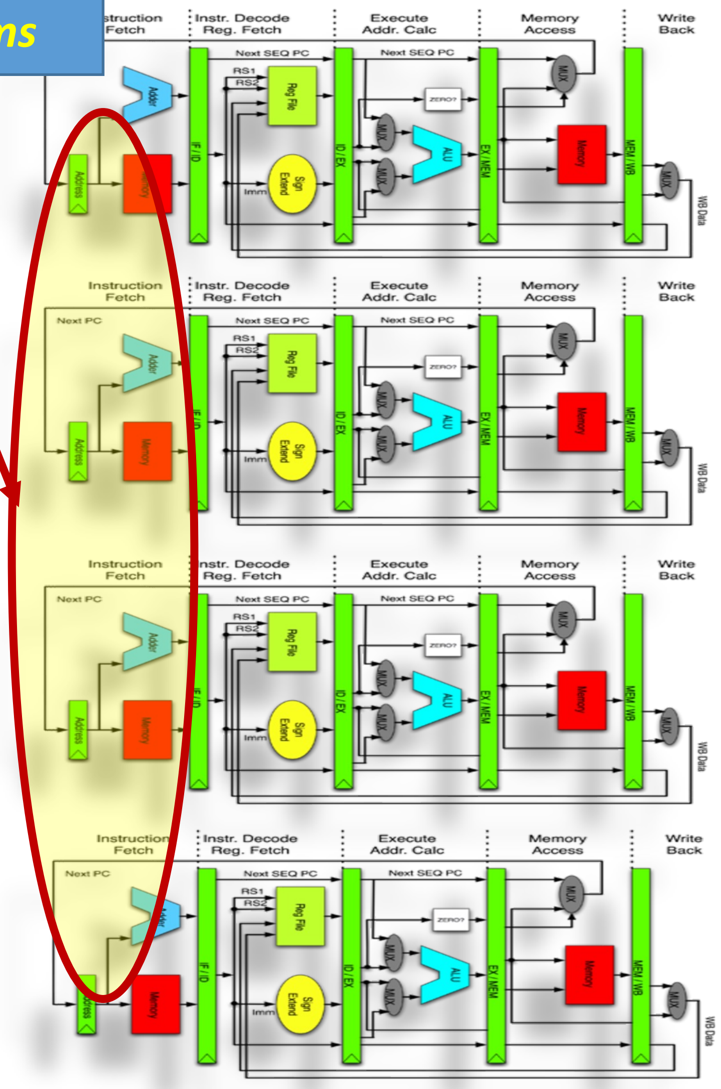
Other techniques extract parallelism here, try to let the machine find parallelism

Superscalar processors



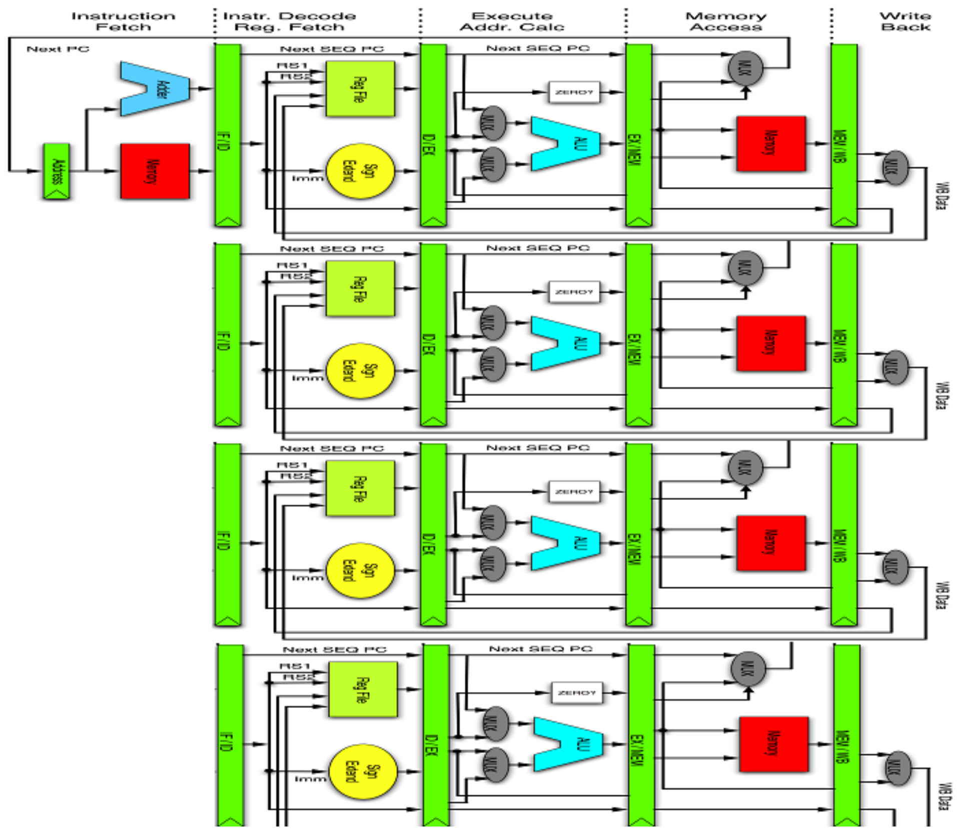
Superscalar processors

Remove extra instruction streams

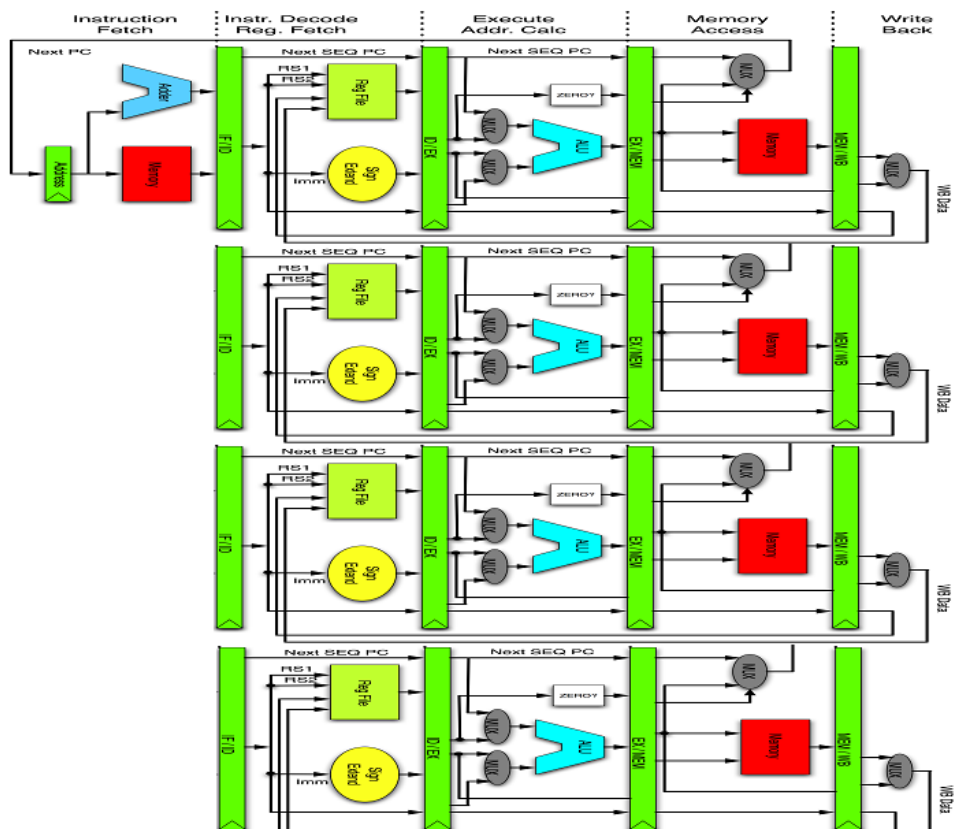


Superscalar processors

Superscalar processors



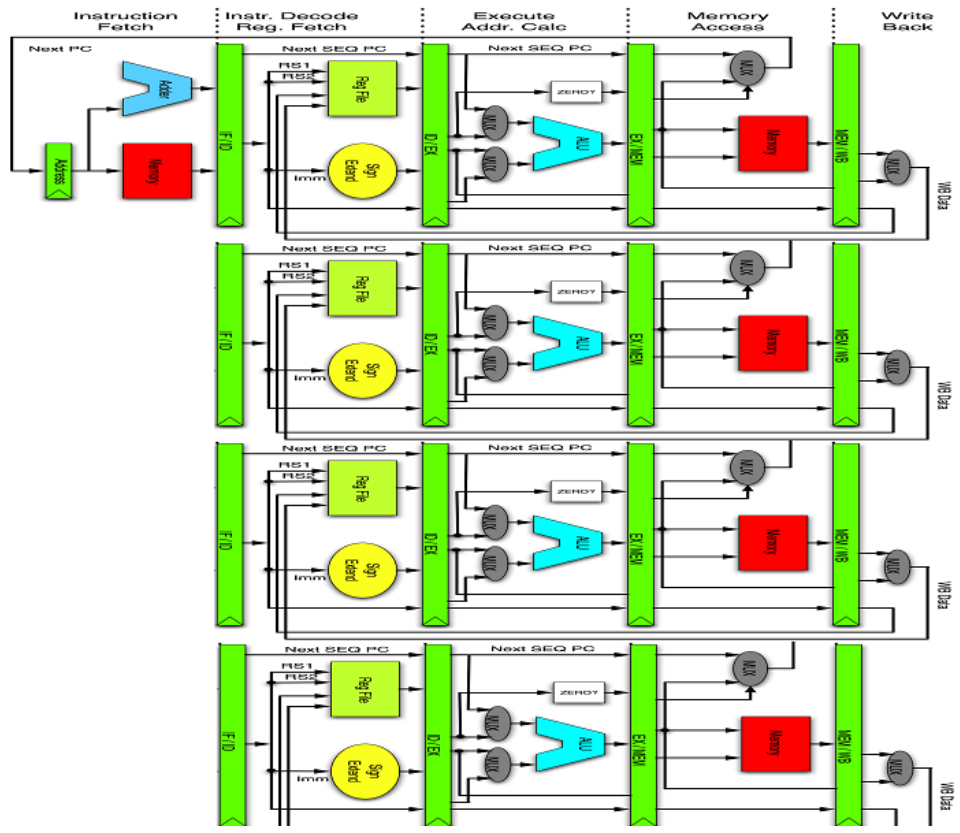
Superscalar processors



```
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);
}
```

Superscalar processors

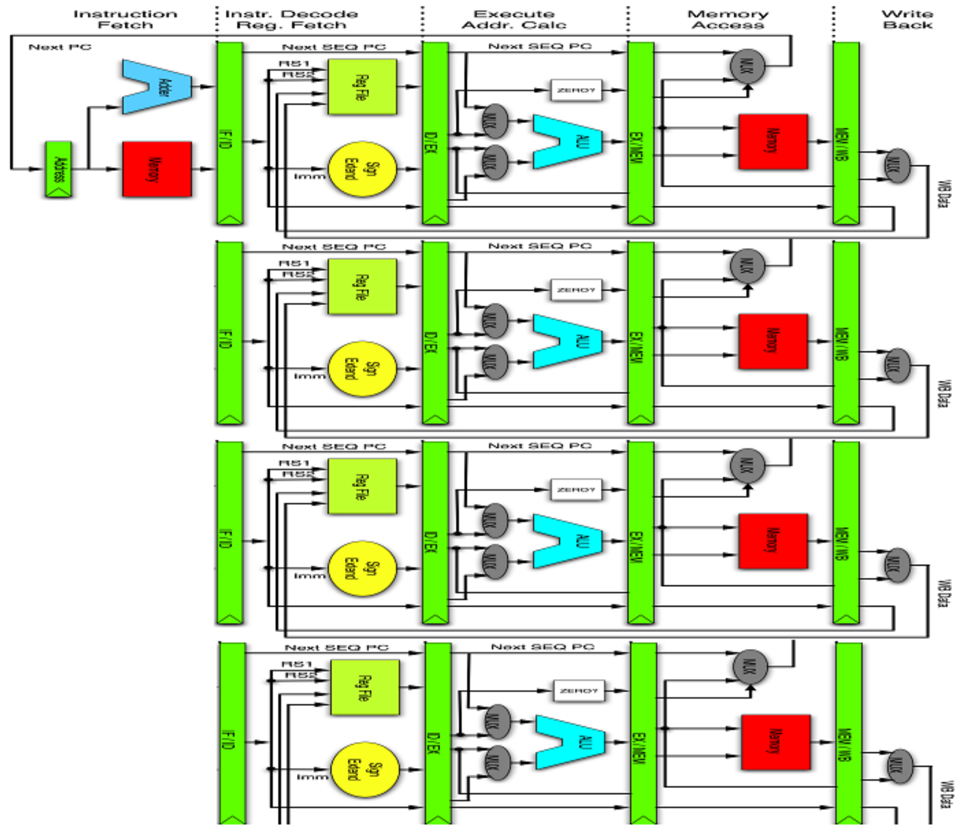


```
main() {
  for(i=0; i<CORES; i++)
    pthread_create(decode_exec);
    while(true) {
      instruction = fetch();
      enqueue(instruction);
    }
}


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?

Superscalar processors



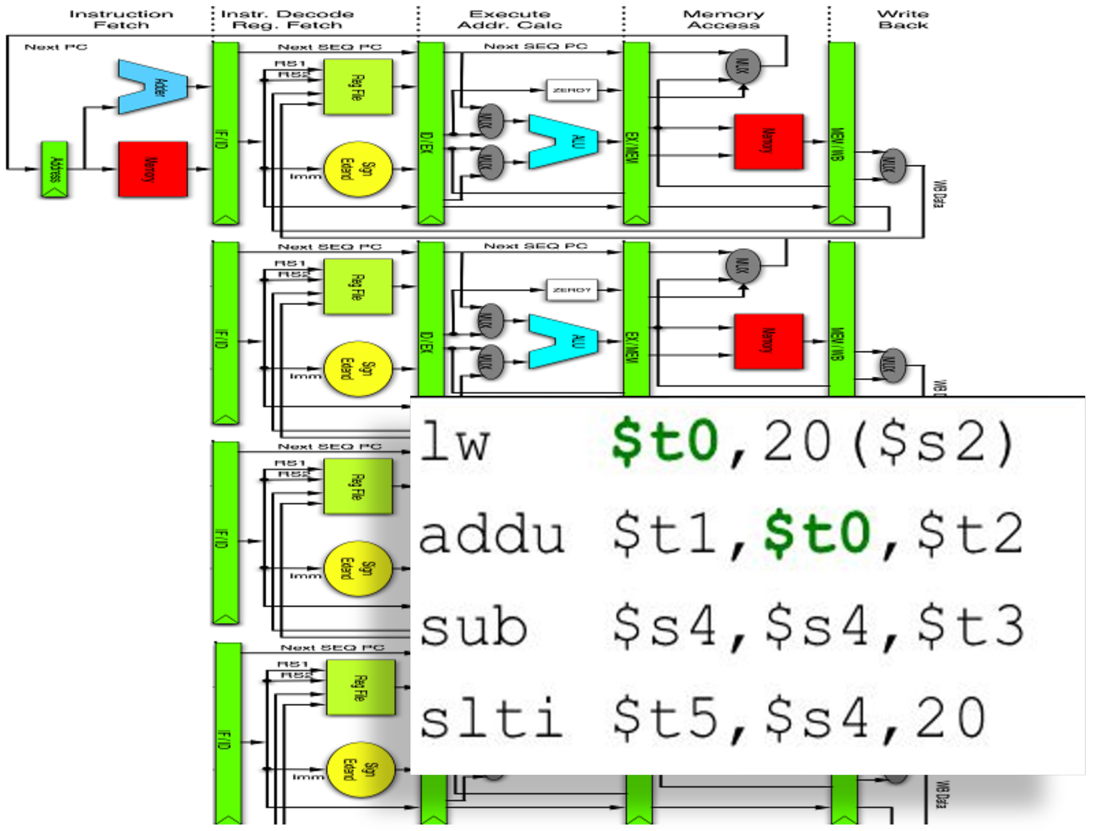
```
main() {
  for(i=0; i<CORES; i++)
    pthread_create(decode_exec);
    while(true) {
      instruction = fetch();
      enqueue(instruction);
    }
}


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.

Superscalar processors



```
lw    $t0,20($s2)
addu  $t1,$t0,$t2
sub   $s4,$s4,$t3
slti  $t5,$s4,20
```

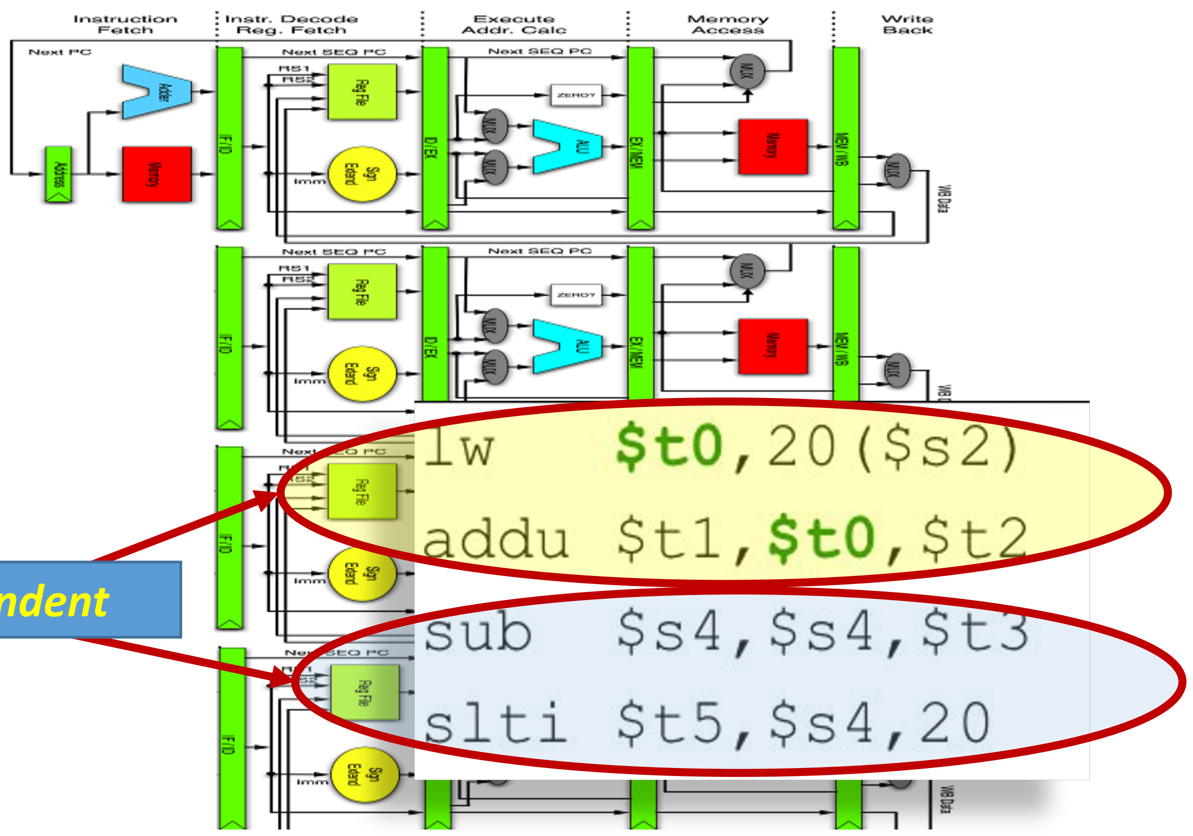
```
main() {
  for(i=0; i<CORES; i++)
    pthread_create(decode_exec);
    while(true) {
      instruction = fetch();
      enqueue(instruction);
    }
}

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.

Superscalar processors



```
lw    $t0,20($s2)
addu  $t1,$t0,$t2
sub   $s4,$s4,$t3
slti  $t5,$s4,20
```

independent

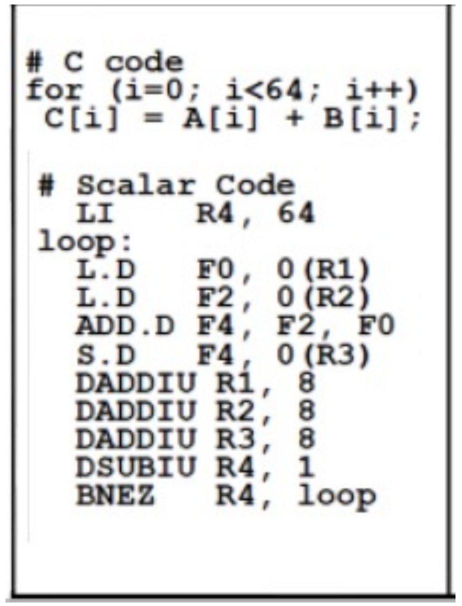
```
main() {
  for(i=0; i<CORES; i++)
    pthread_create(decode_exec);
    while(true) {
      instruction = fetch();
      enqueue(instruction);
    }
}

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.

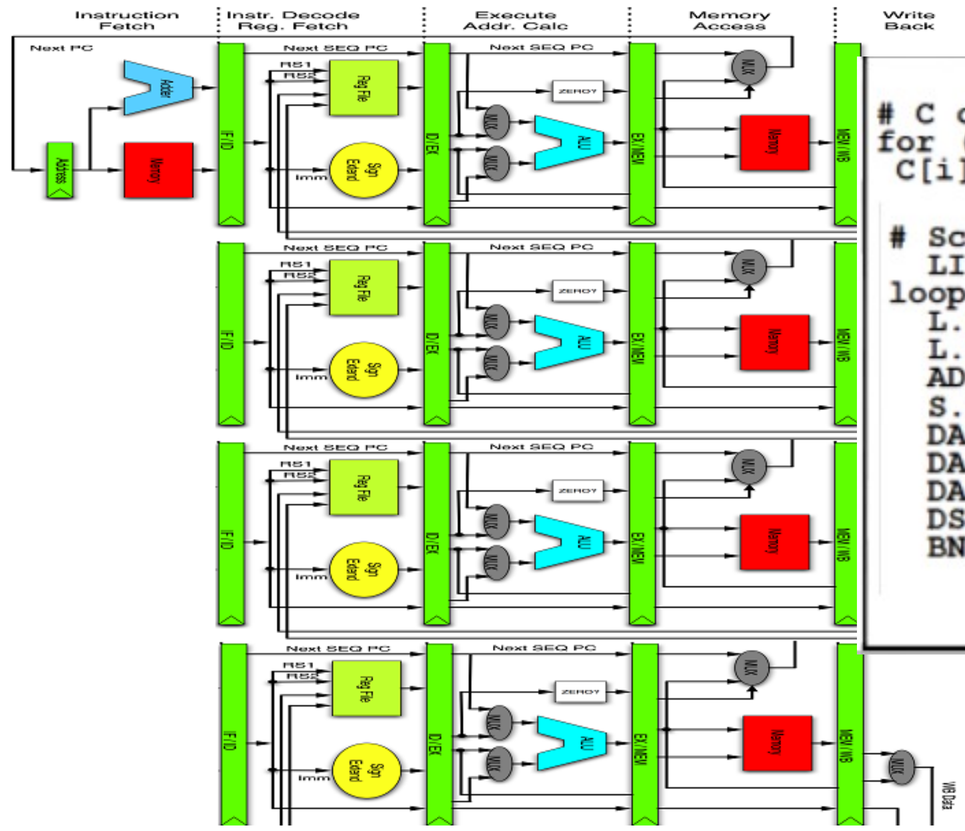
Vector/SIMD processors

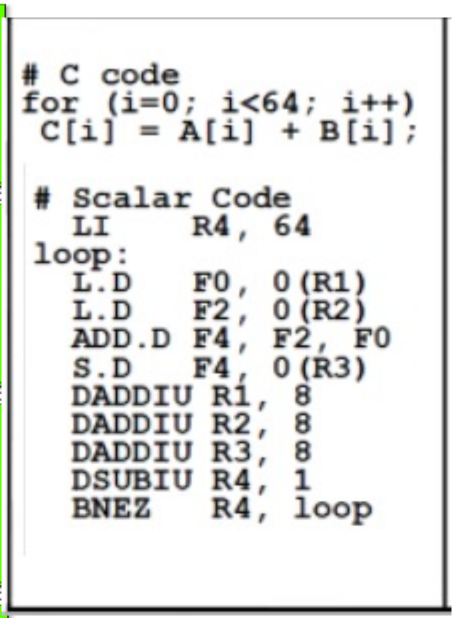


```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];

 # 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/SIMD processors



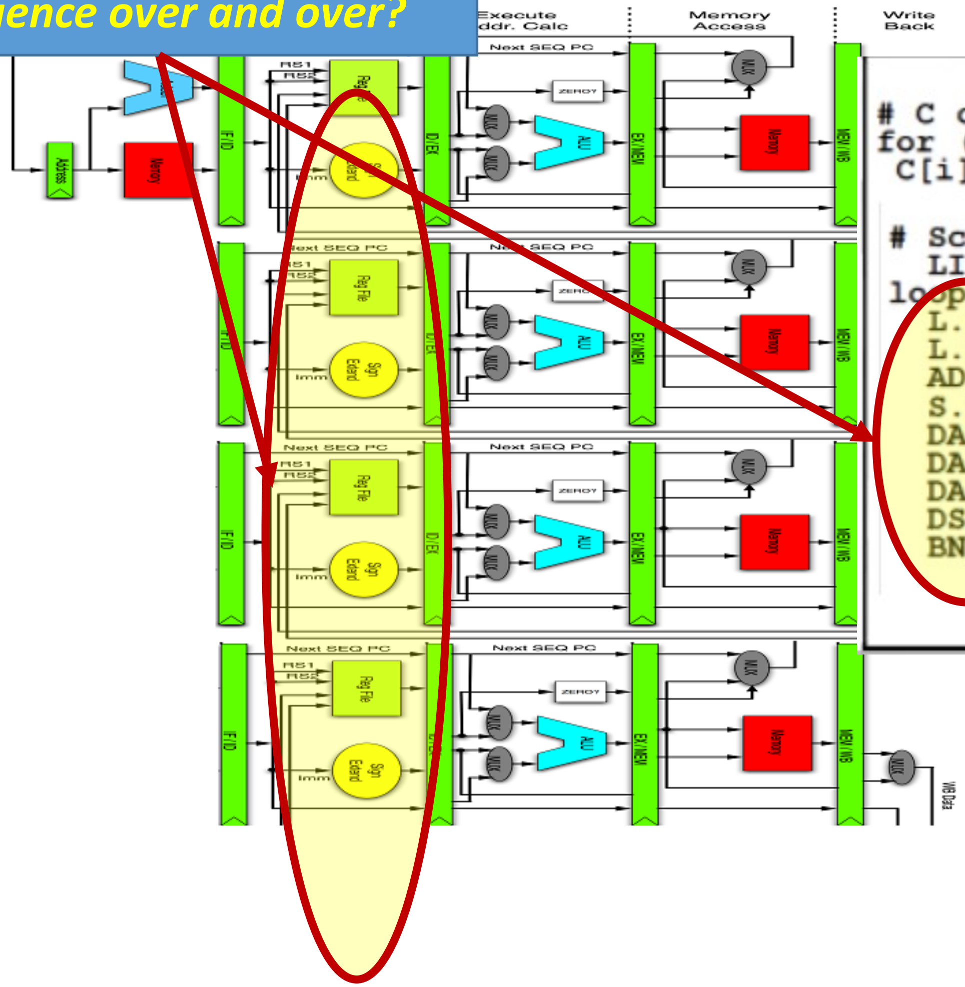


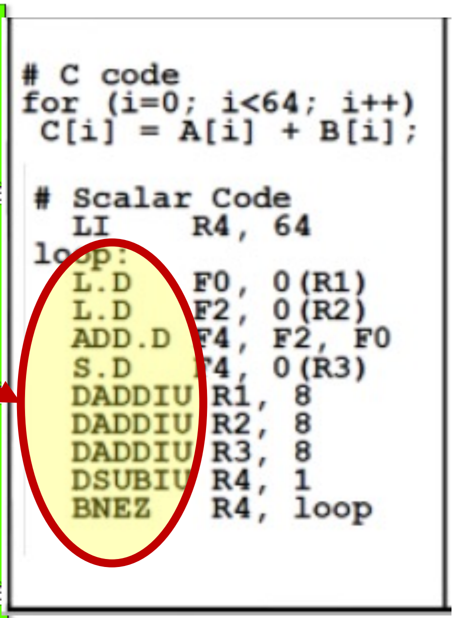
```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];

 # 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/SIMD processors

Why decode same instruction sequence over and over?

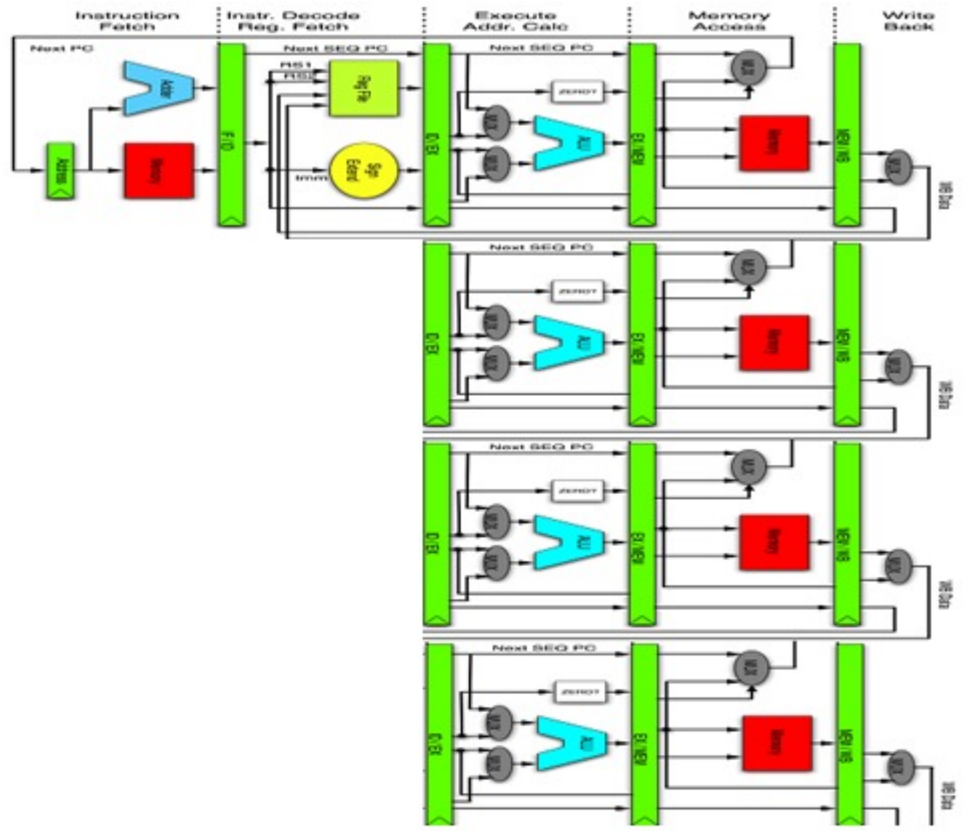




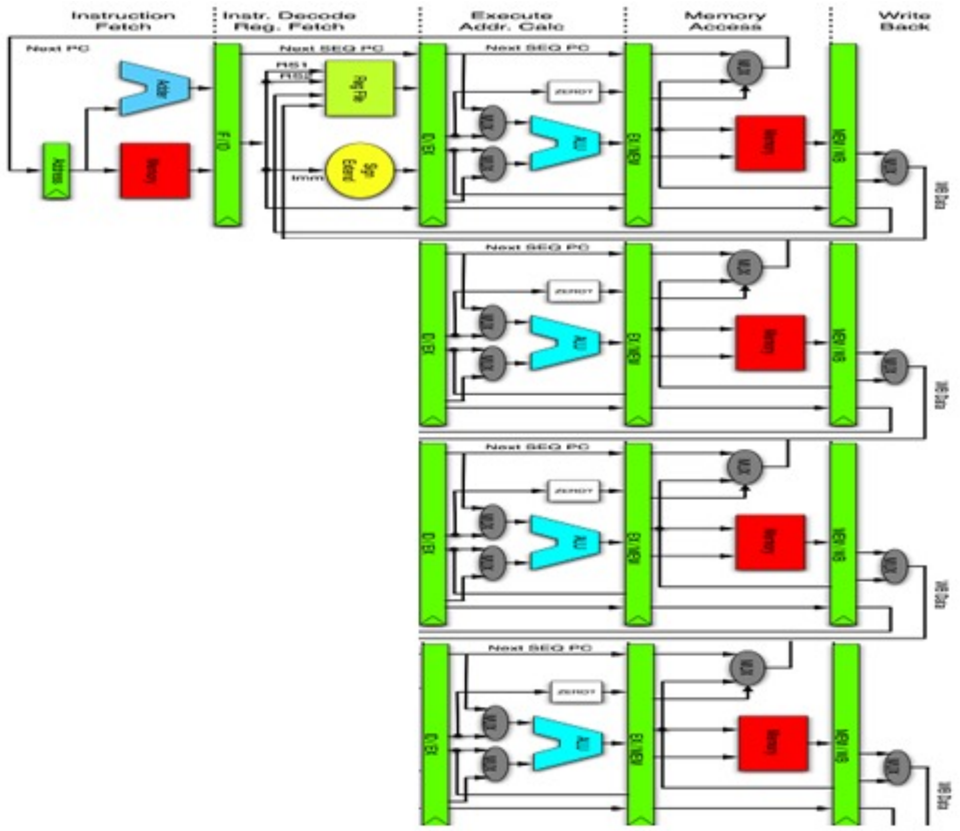
```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];

 # 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
```

Vector/SIMD processors



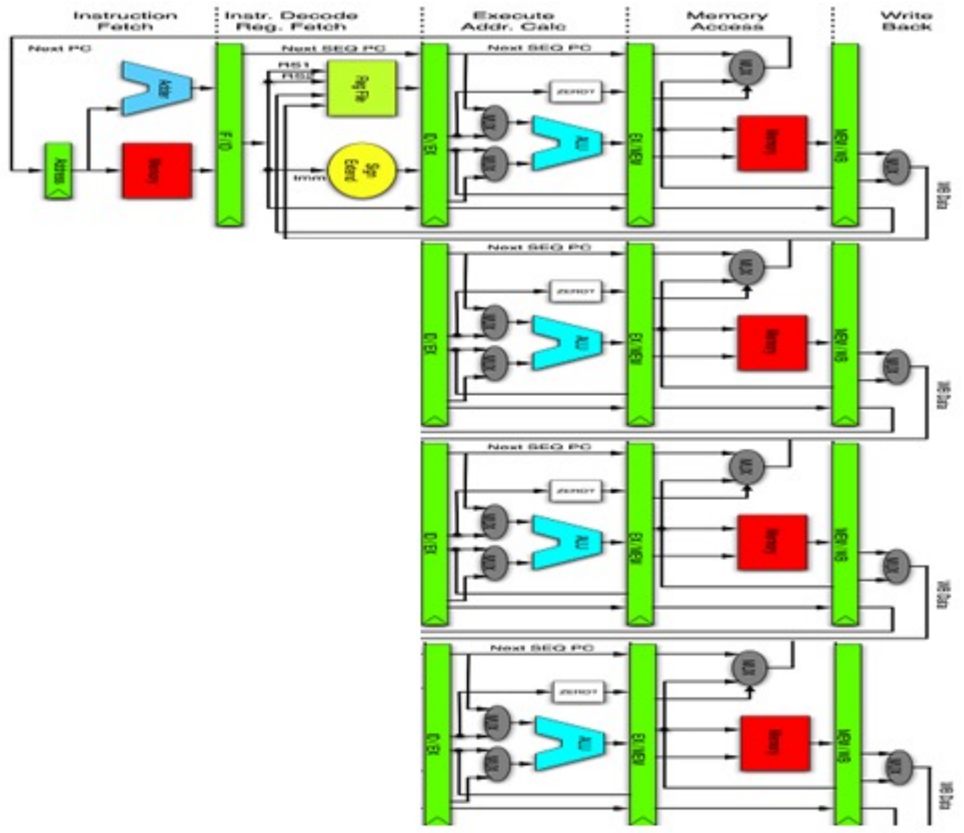
Vector/SIMD processors



```
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_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

Vector/SIMD processors

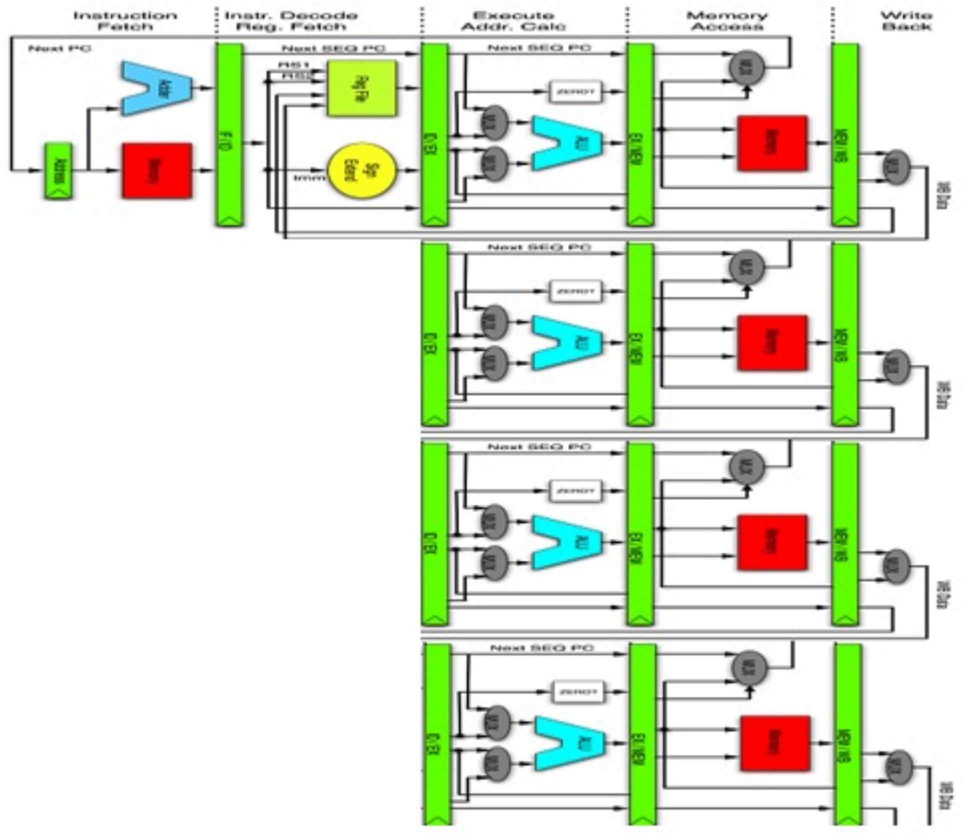


```
main() {
  for(i=0; i<CORES; i++)
    pthread_create(exec);
    while(true) {
       ops, regs = fetch_decode();
       enqueue(ops, regs);
    }
}


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

Single instruction stream, multiple computations

Vector/SIMD processors



```
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_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}
```

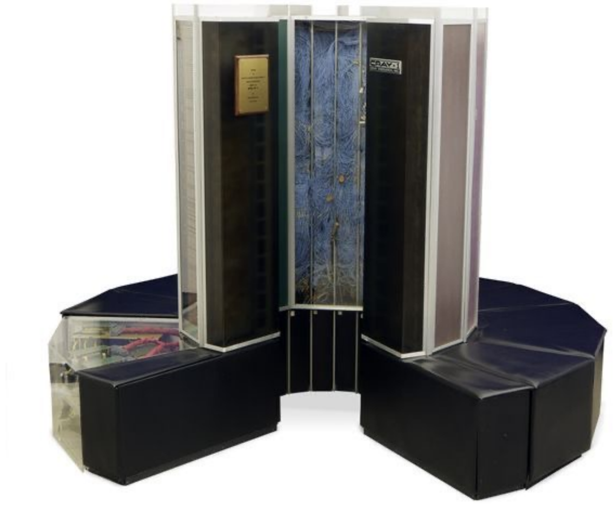
Single instruction stream, multiple computations

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

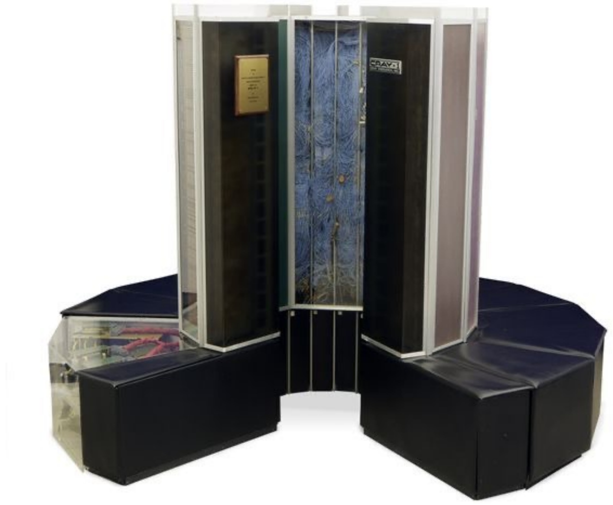
Vector Processors



- 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

Vector Processors

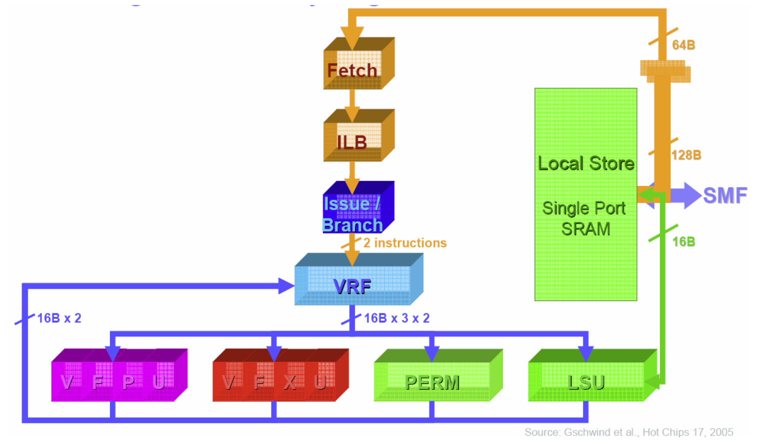


- Process multiple data elements simultaneously.
- Common in supercomputers of the 1970’s 80’s and 90’s.
- Modern CPUs support some vector processing instructions
  - Usually called SIMD
- Can operate on a few vectors elements per clock cycle in a pipeline or,
  - SIMD operate on all per clock cycle

- 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

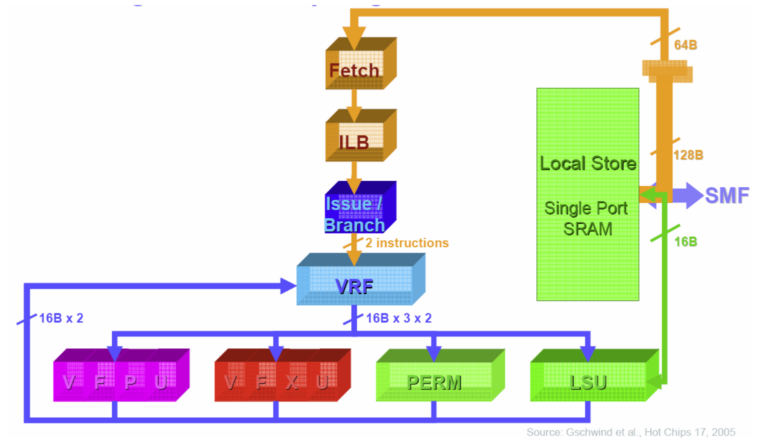
Vector Processors

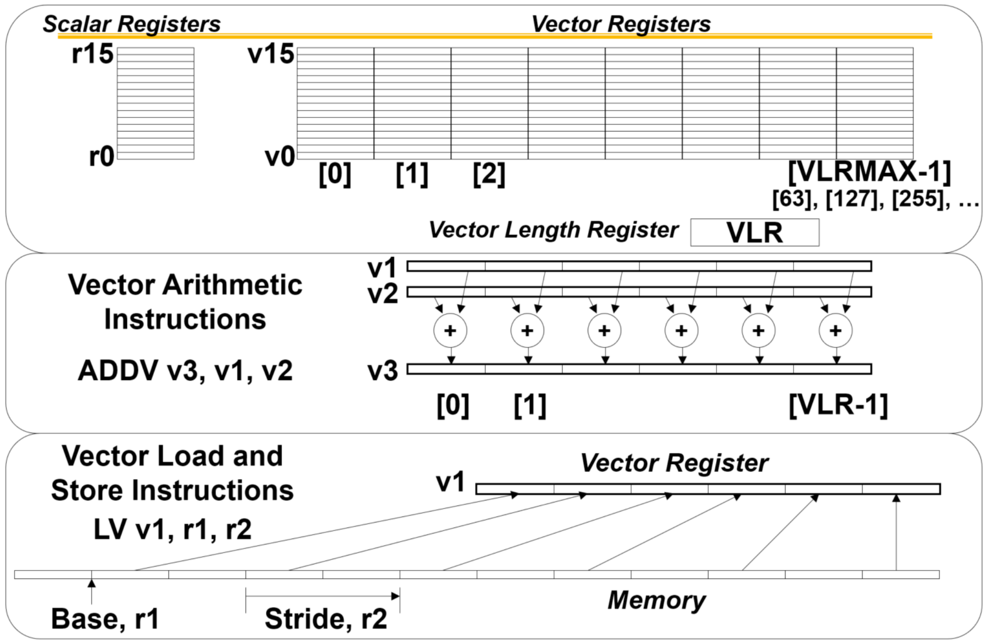


Implementation:

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

Vector Processors

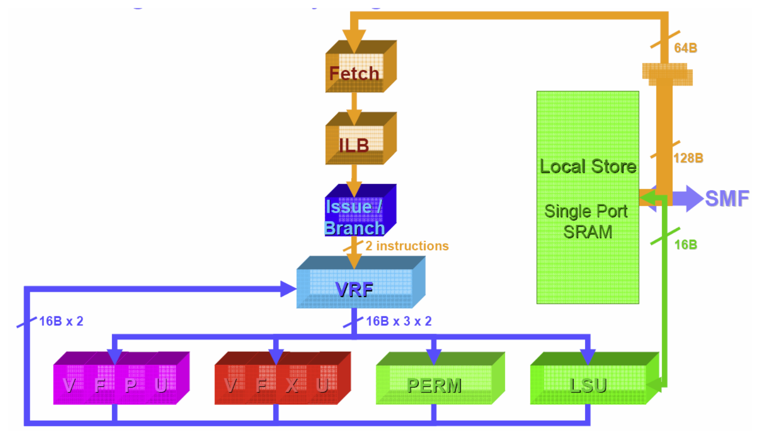




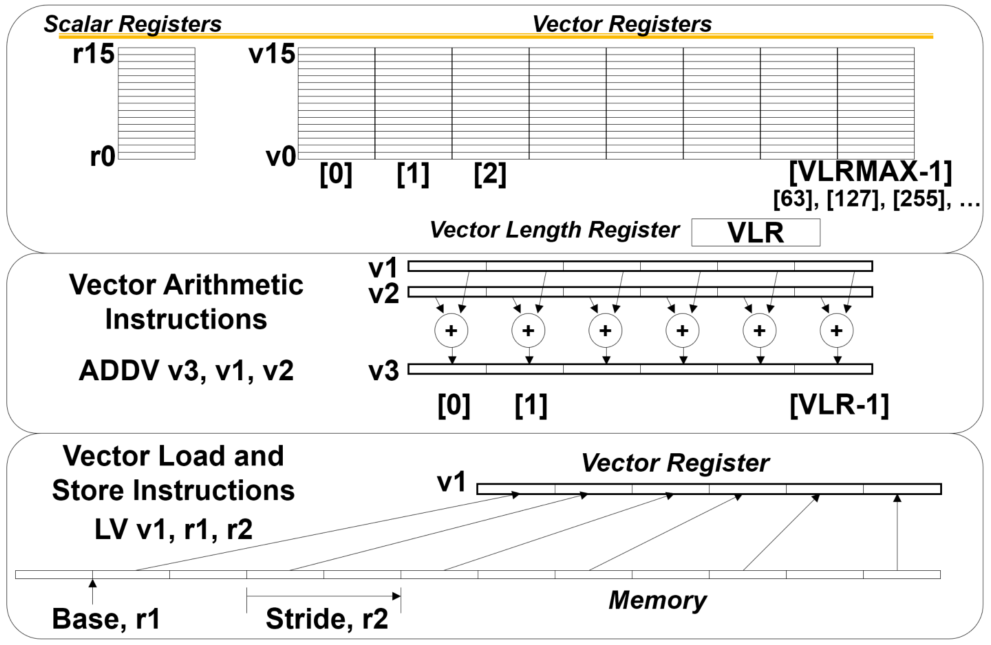
Implementation:

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

Vector Processors

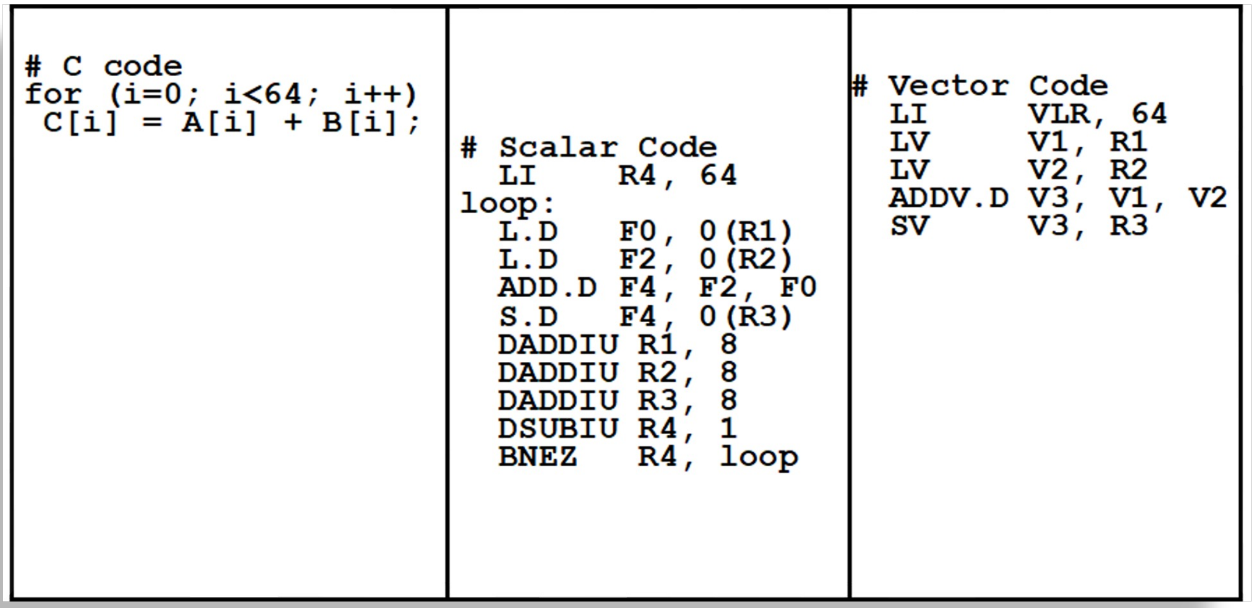


Source: Gschwind et al., Hot Chips 17, 2005



Implementation:

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



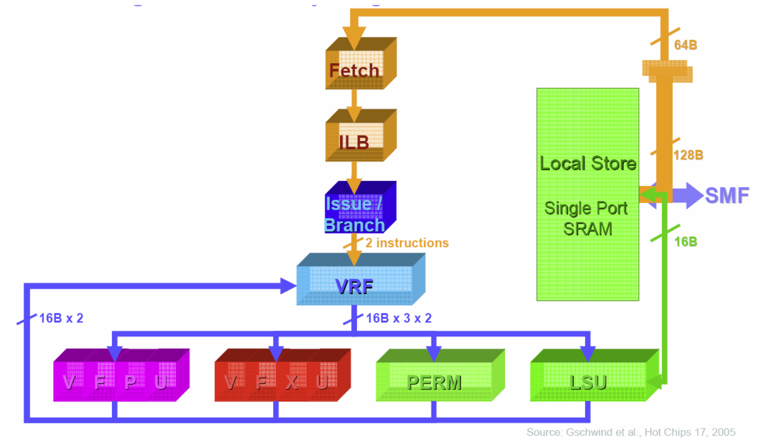
```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

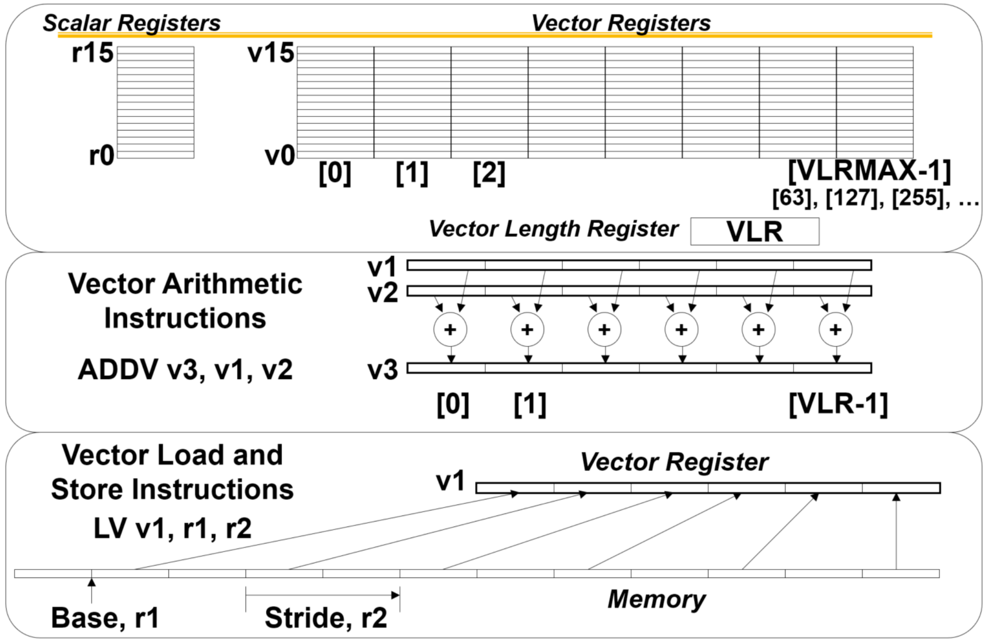
```
# Scalar Code
  LI     R4, 64
loop:
  L.D    F0, 0(R1)
  L.D    F2, 0(R2)
  ADD.D  F4, F2, F0
  S.D    F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ   R4, loop
```

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

Vector Processors

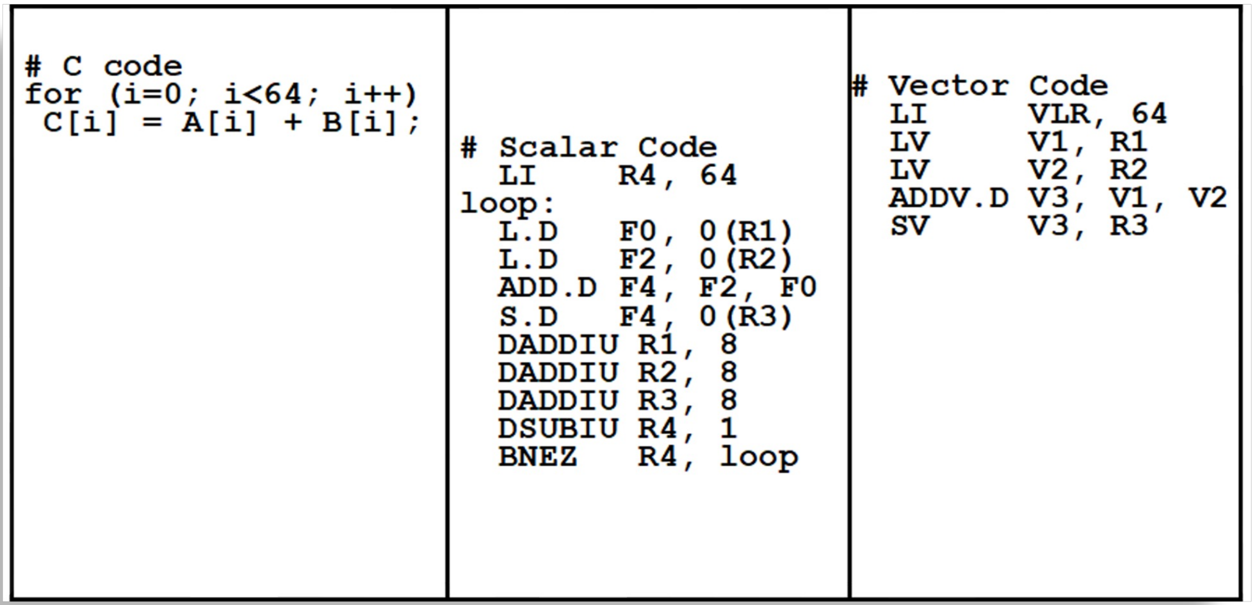
GPUs: same basic idea





Implementation:

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

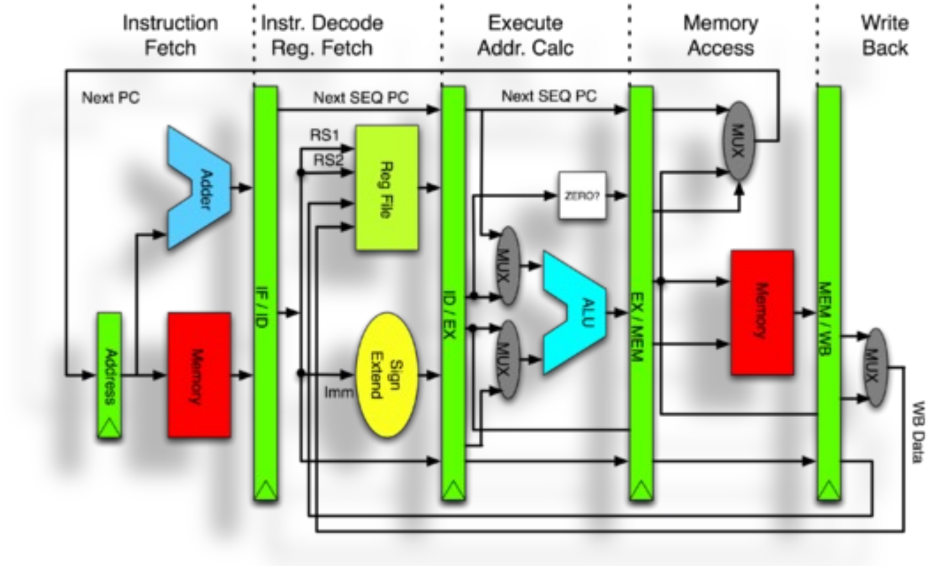


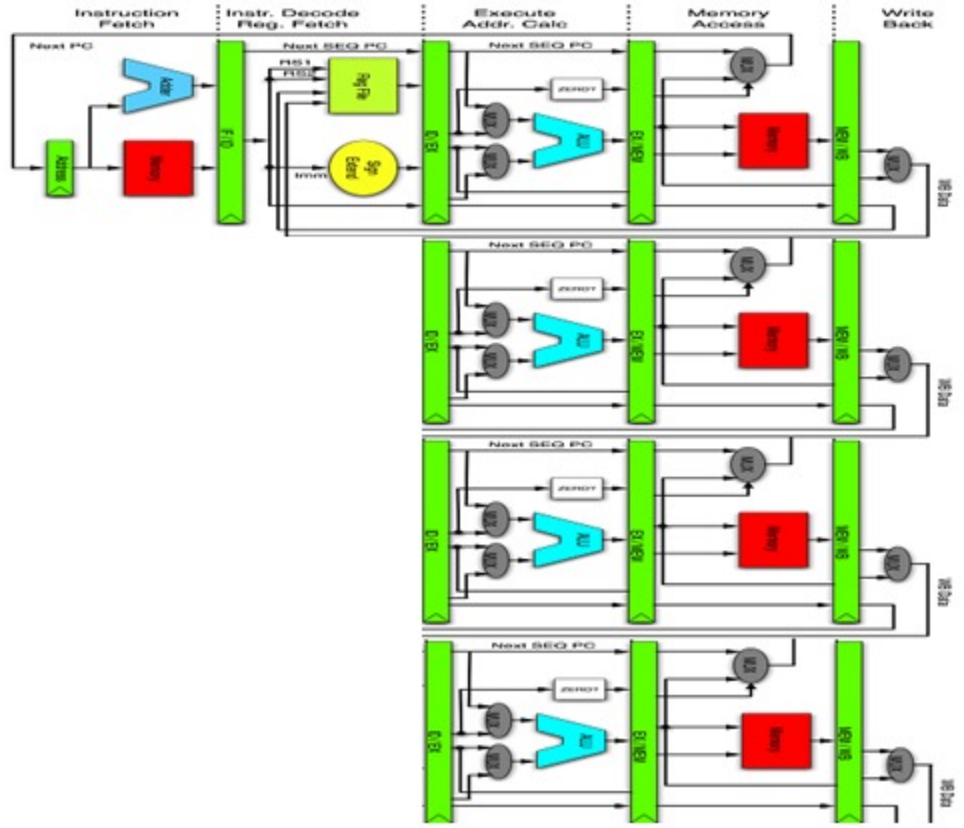
```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

```
# 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
```

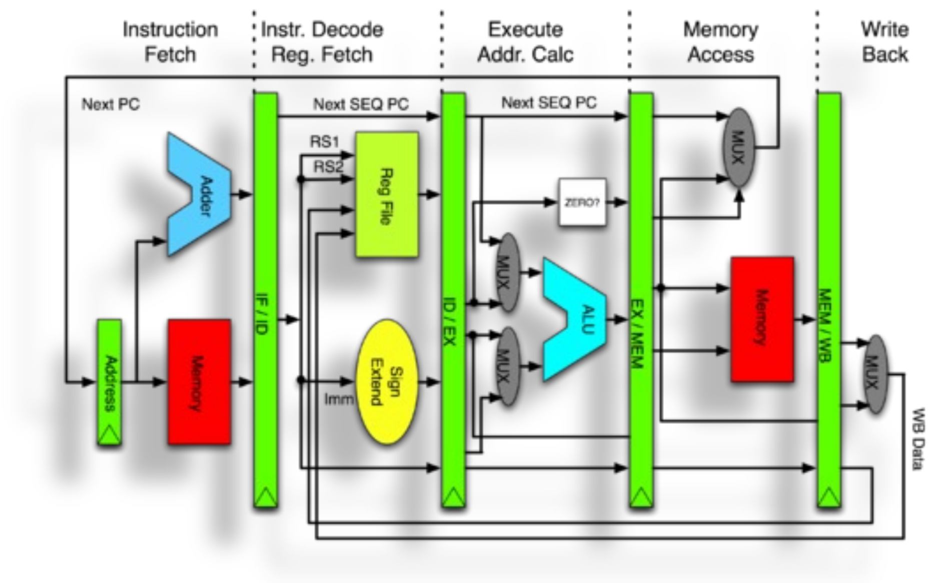
```
# Vector Code
  LI     VLR, 64
  LV     V1, R1
  LV     V2, R2
  ADDV.D V3, V1, V2
  SV     V3, R3
```

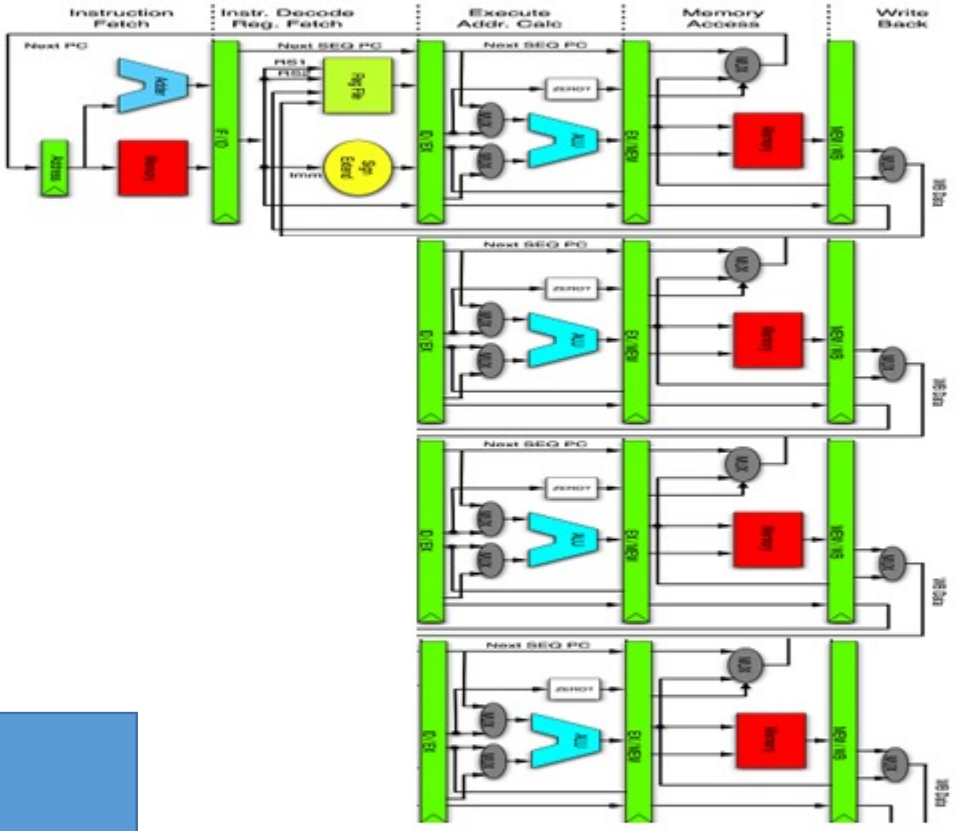
When does vector processing help?





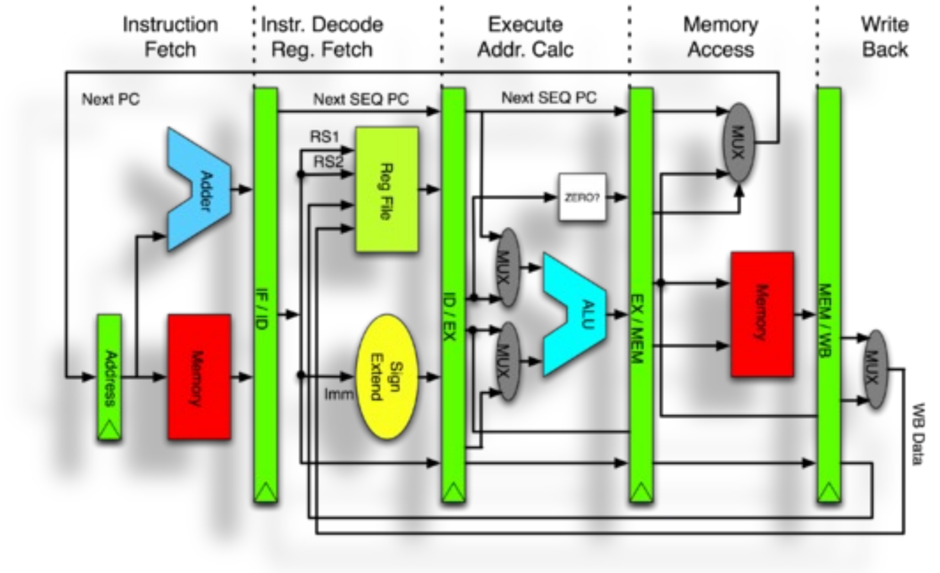
When does vector processing help?

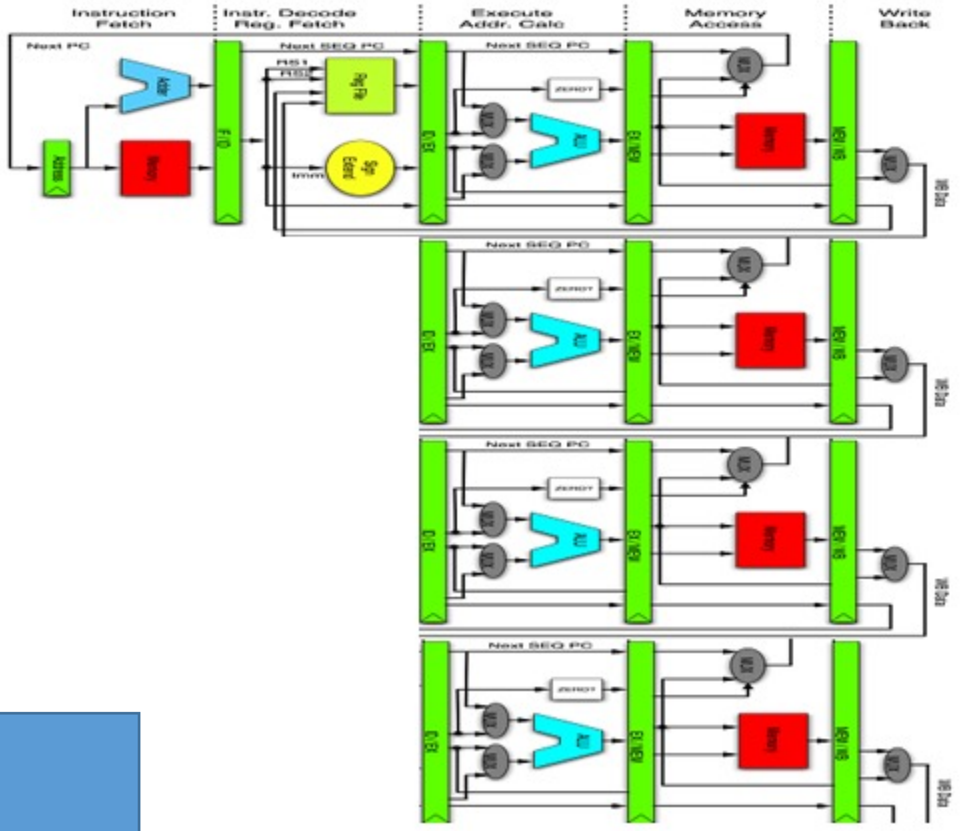




What are the potential bottlenecks here?
When can it improve throughput?

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

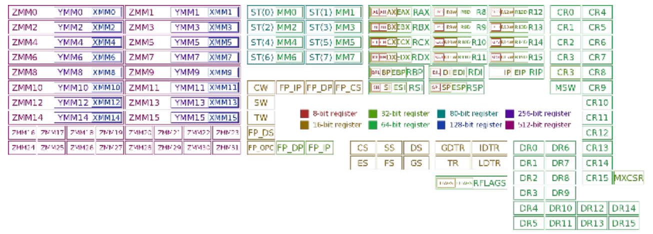
Hardware multi-threading

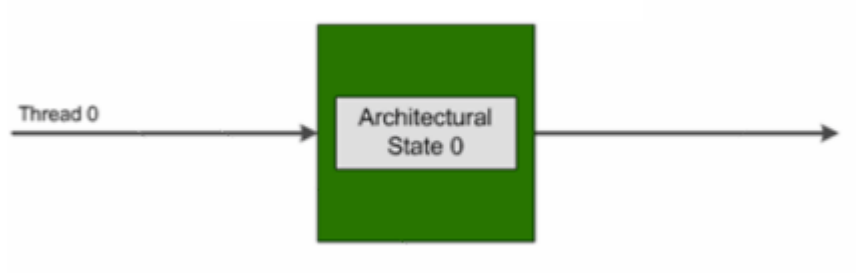
- Address memory bottleneck

Hardware multi-threading

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

Hardware multi-threading

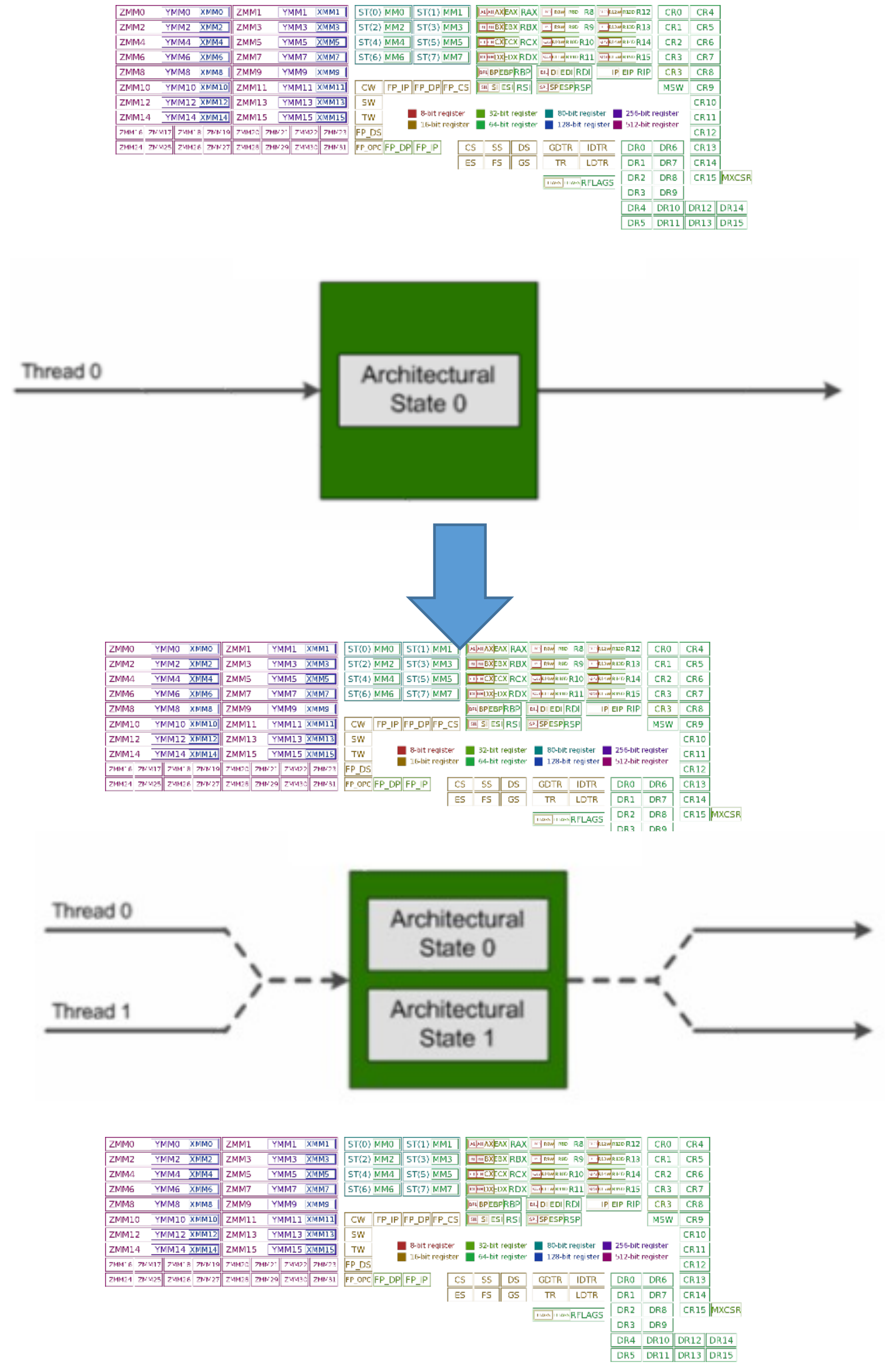




- Address memory bottleneck
- Share exec unit across
  - Instruction streams
  - Switch on stalls

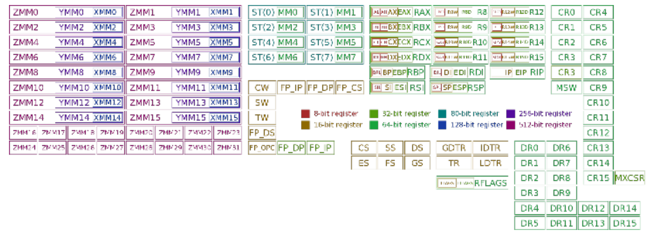
Hardware multi-threading

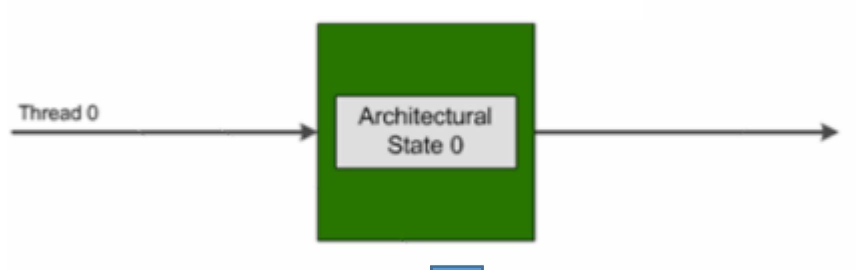
- Address memory bottleneck
- Share exec unit across
  - Instruction streams
  - Switch on stalls

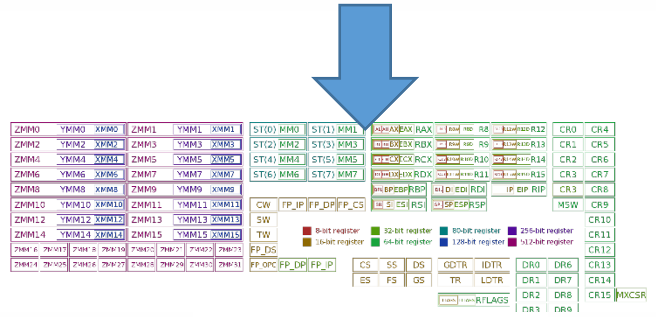


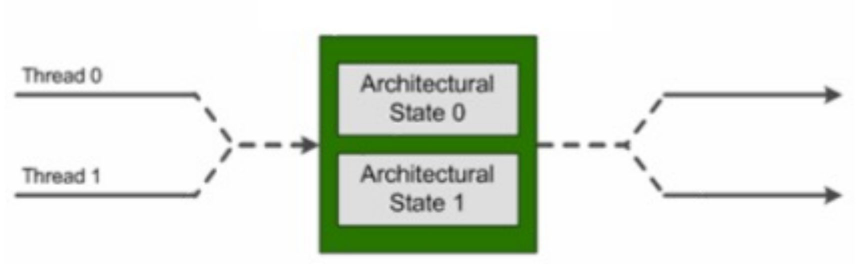
Hardware multi-threading

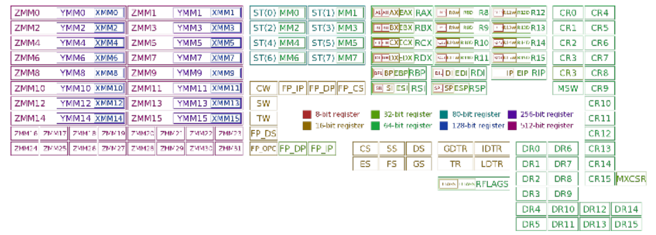
- Address memory bottleneck
- Share exec unit across
  - Instruction streams
  - Switch on stalls
- Looks like multiple cores to the OS





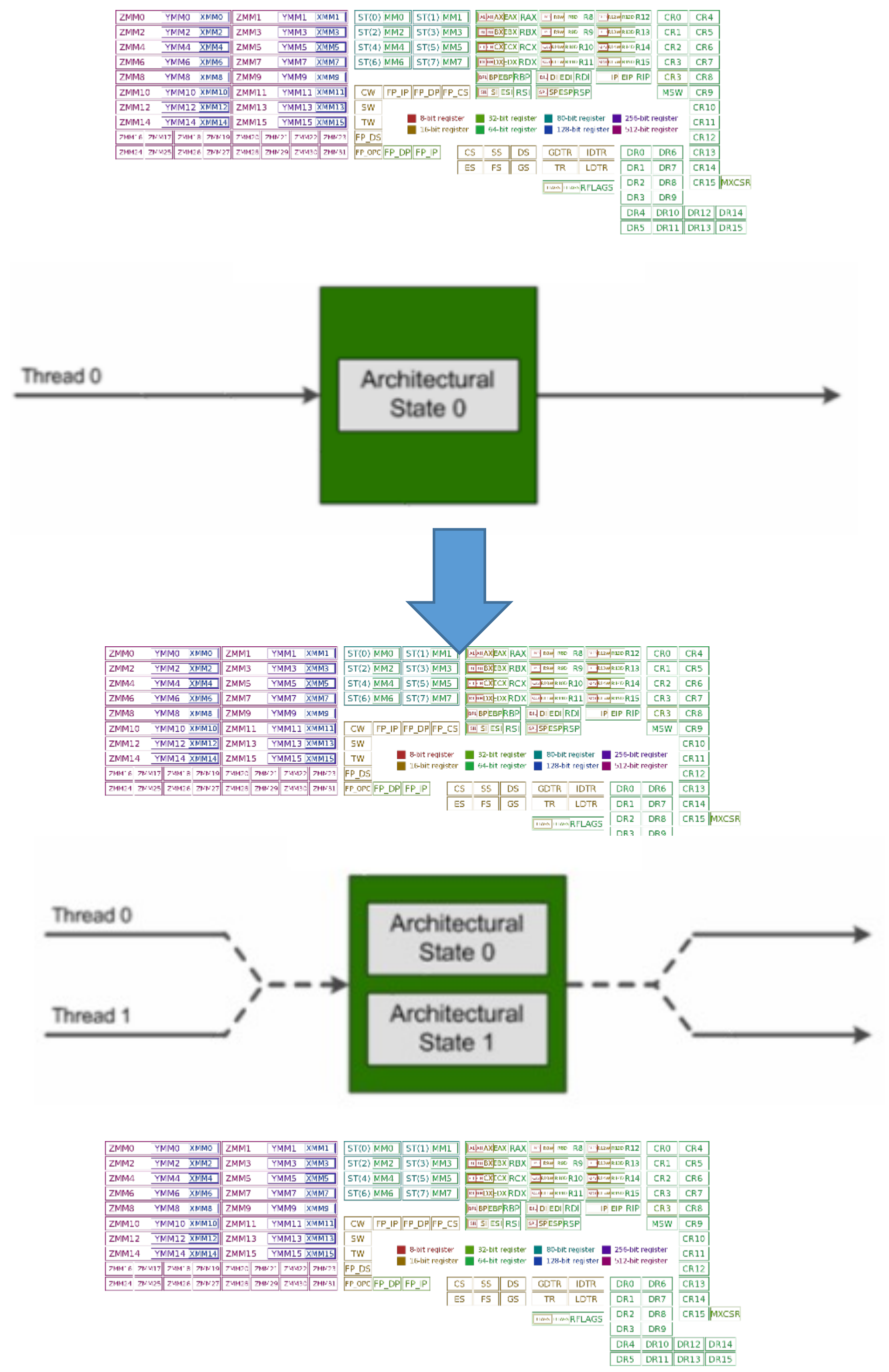




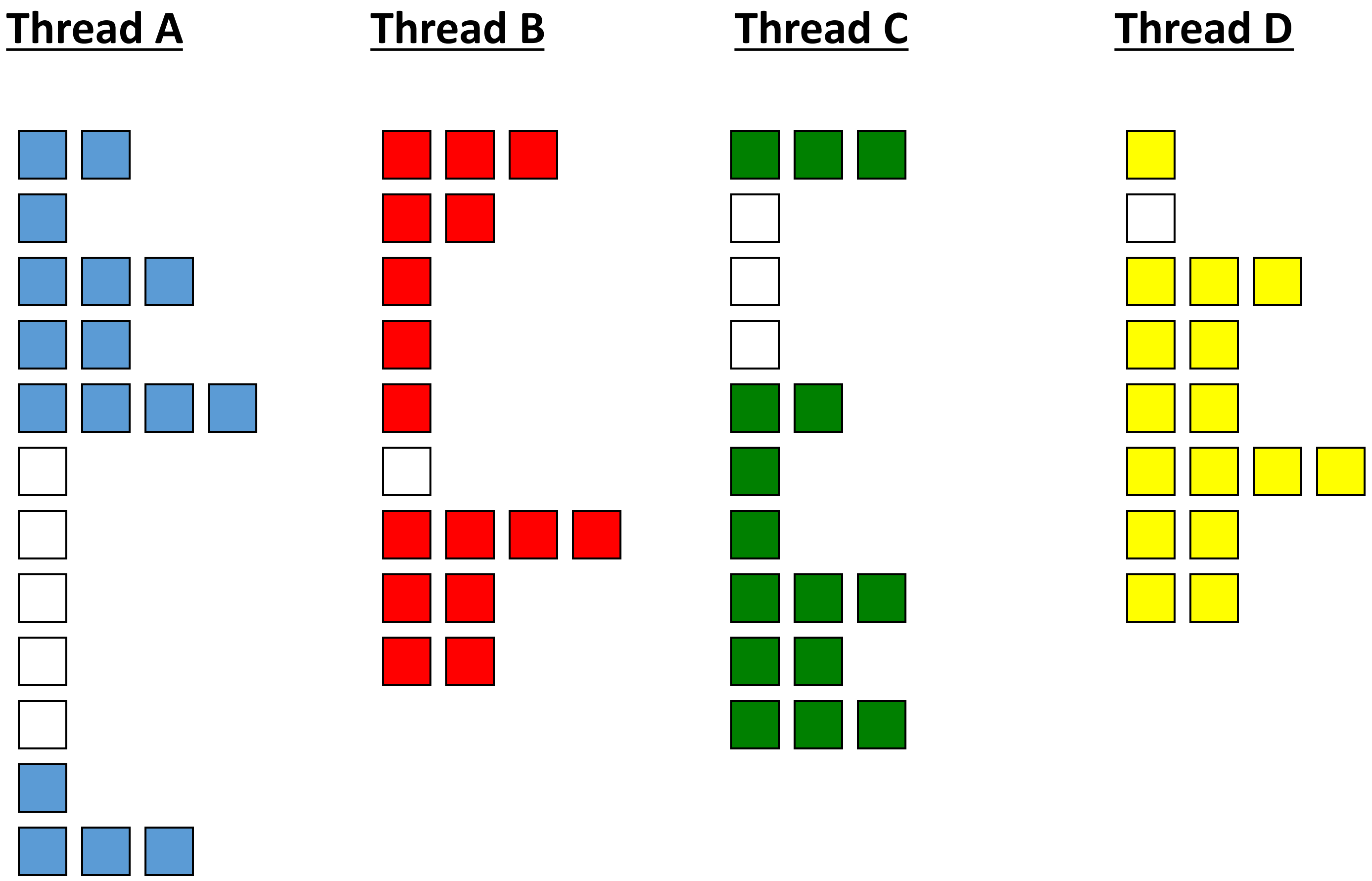


Hardware multi-threading

- 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

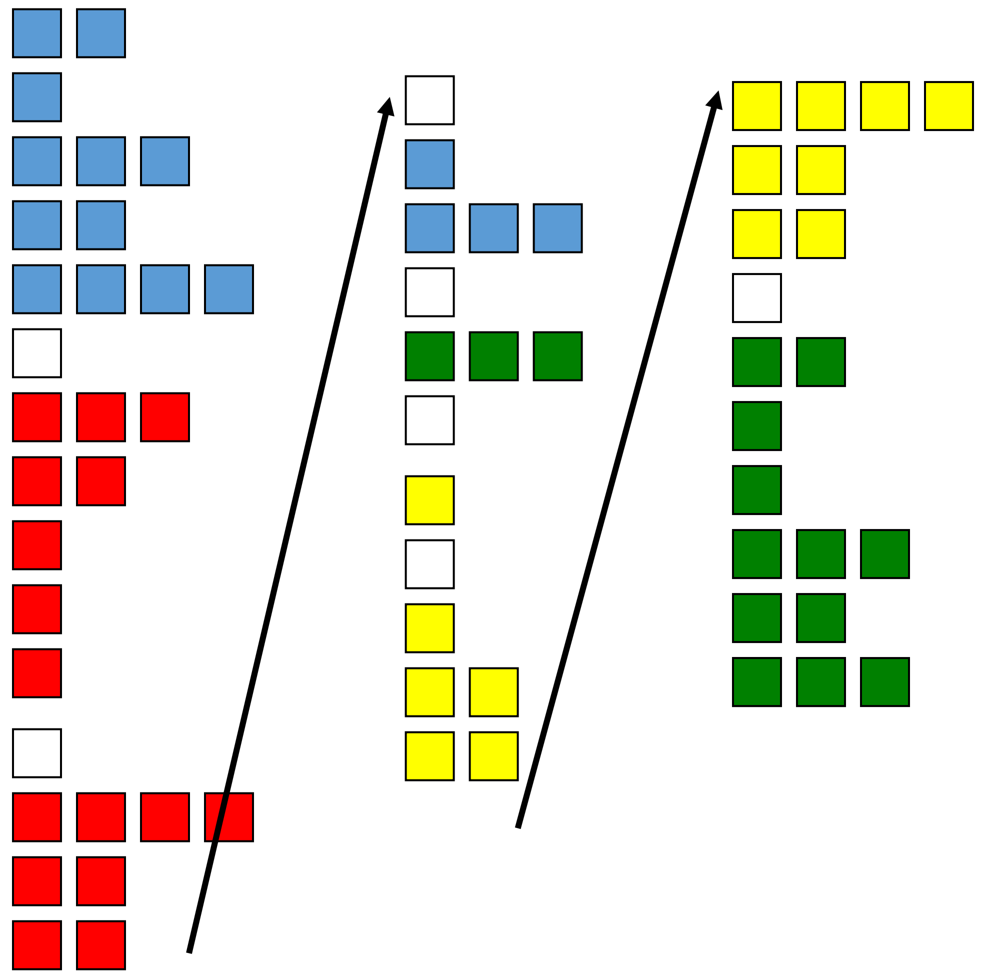
- Single thread runs until a costly stall
    - E.g. 2nd level cache miss

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!

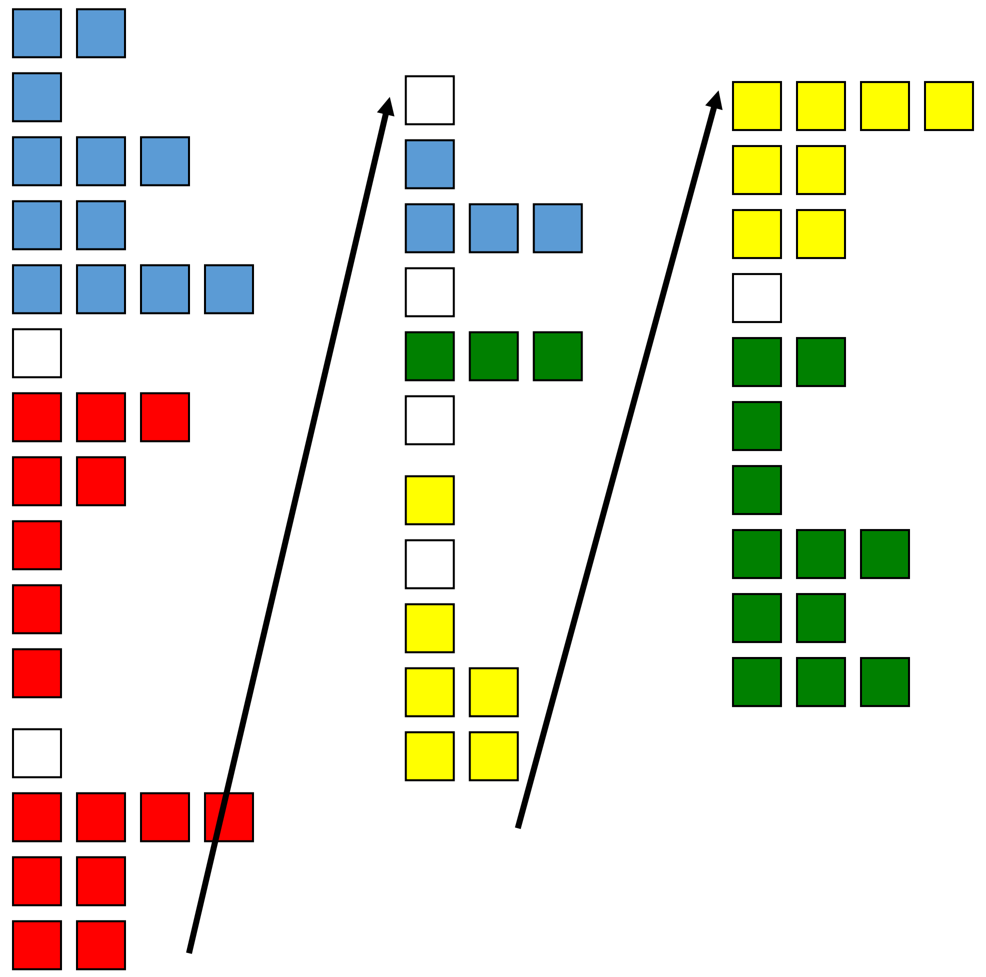
Coarse- grained multithreading

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



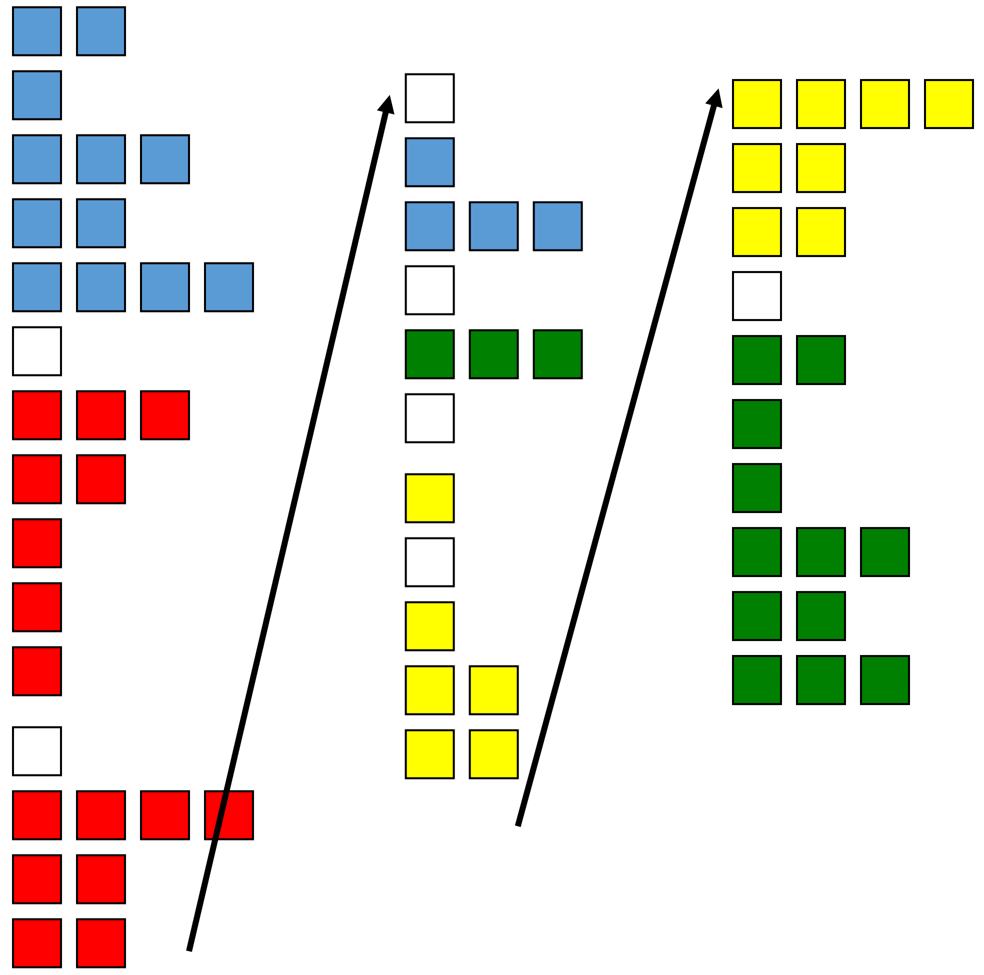
Coarse- grained multithreading

- Single thread runs until a costly stall
  - E.g. 2nd level cache miss
- Another thread starts during stall
  - Pipeline fill time requires several cycles!
- Does not cover short stalls



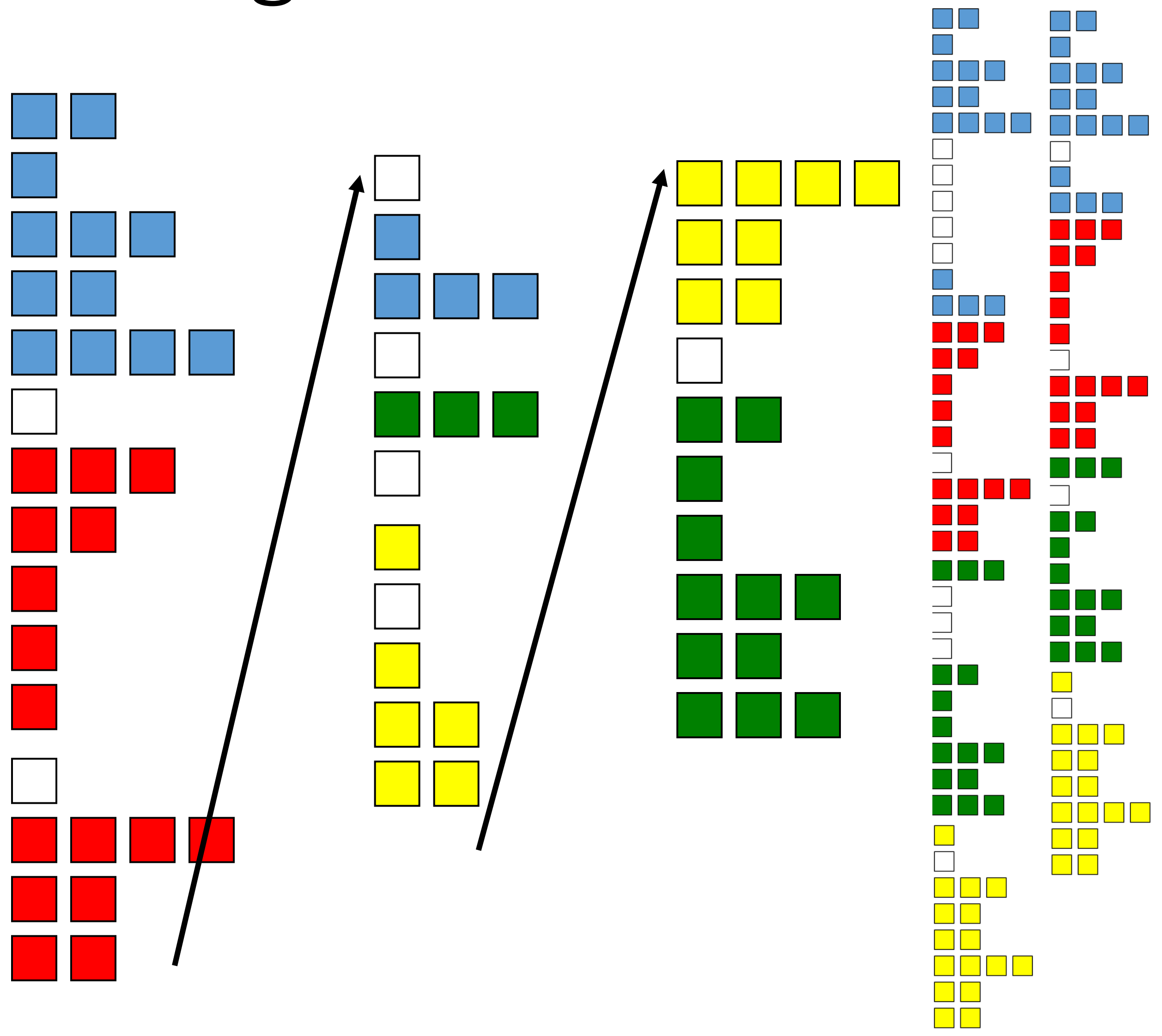
Coarse- grained multithreading

- 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



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



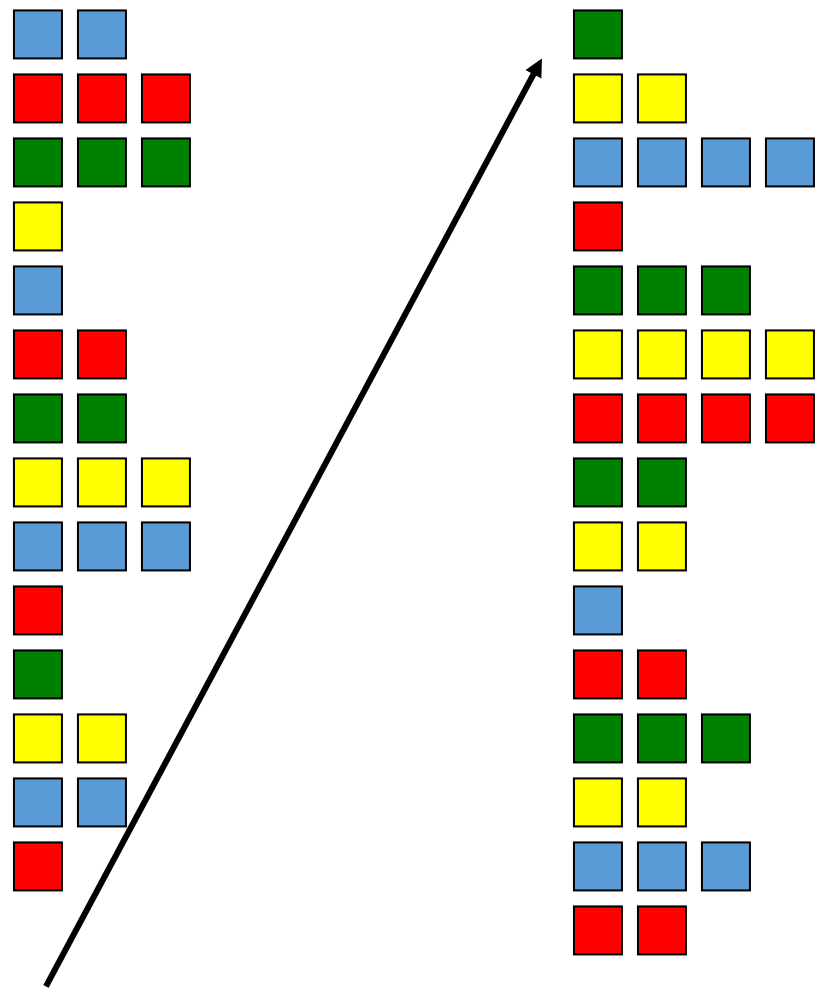
Fine-grained multithreading

Fine-grained multithreading

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

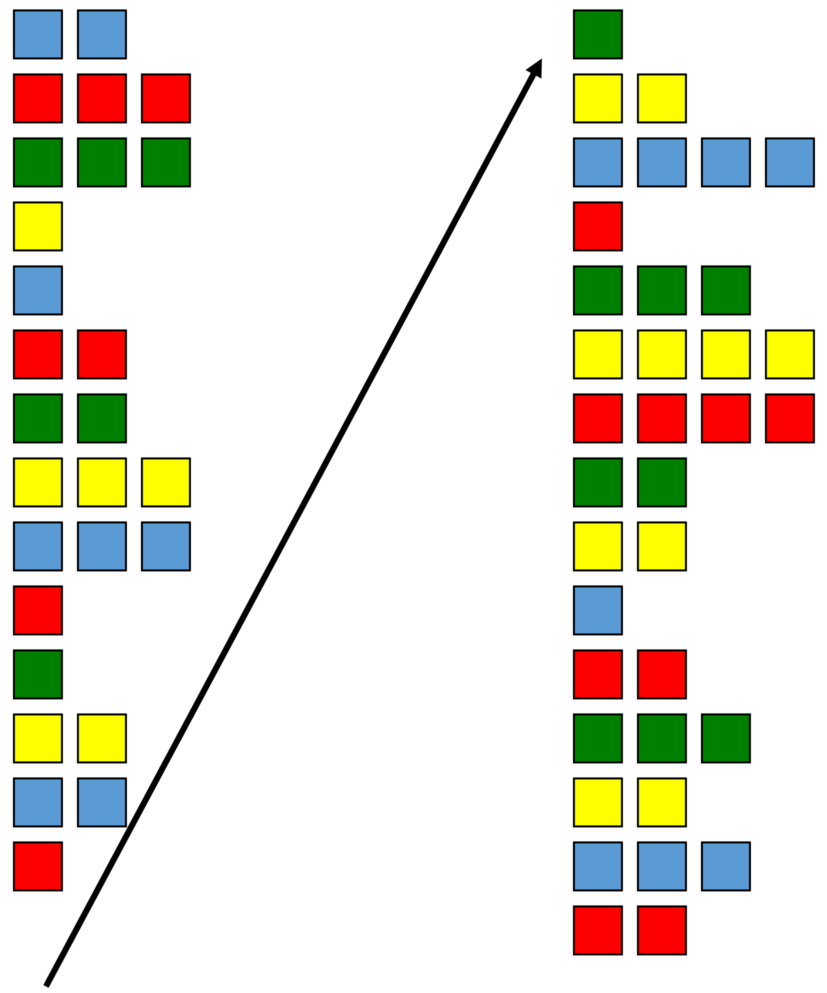
Fine-grained multithreading

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



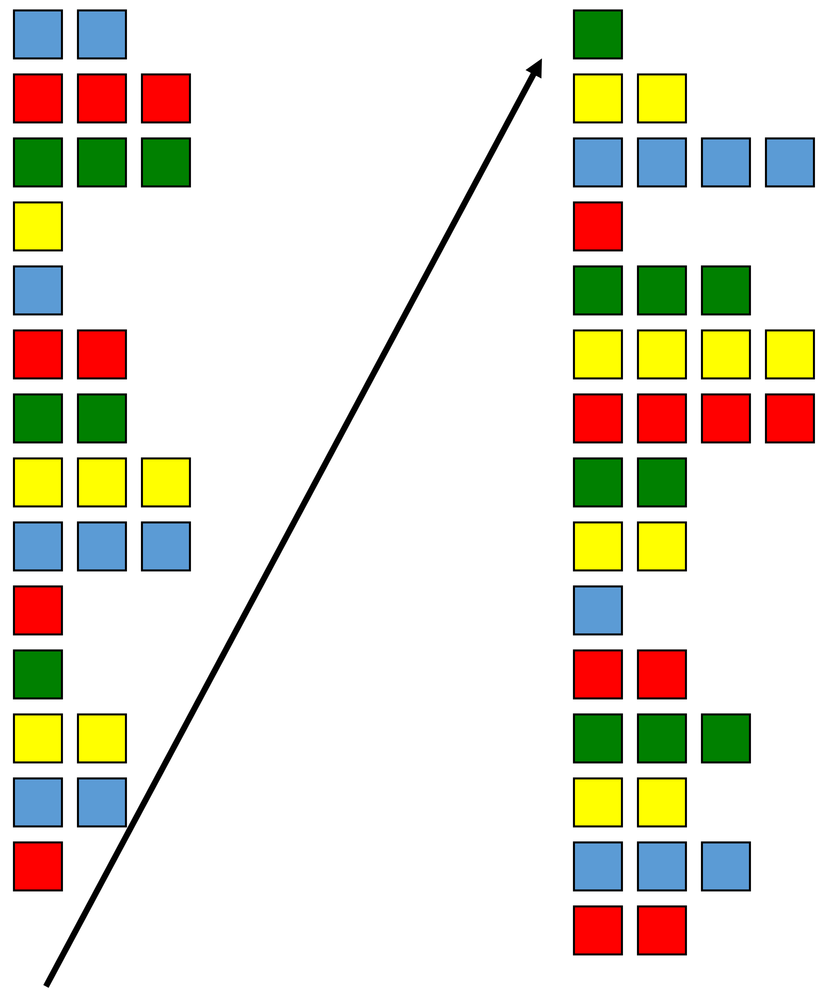
Fine-grained multithreading

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



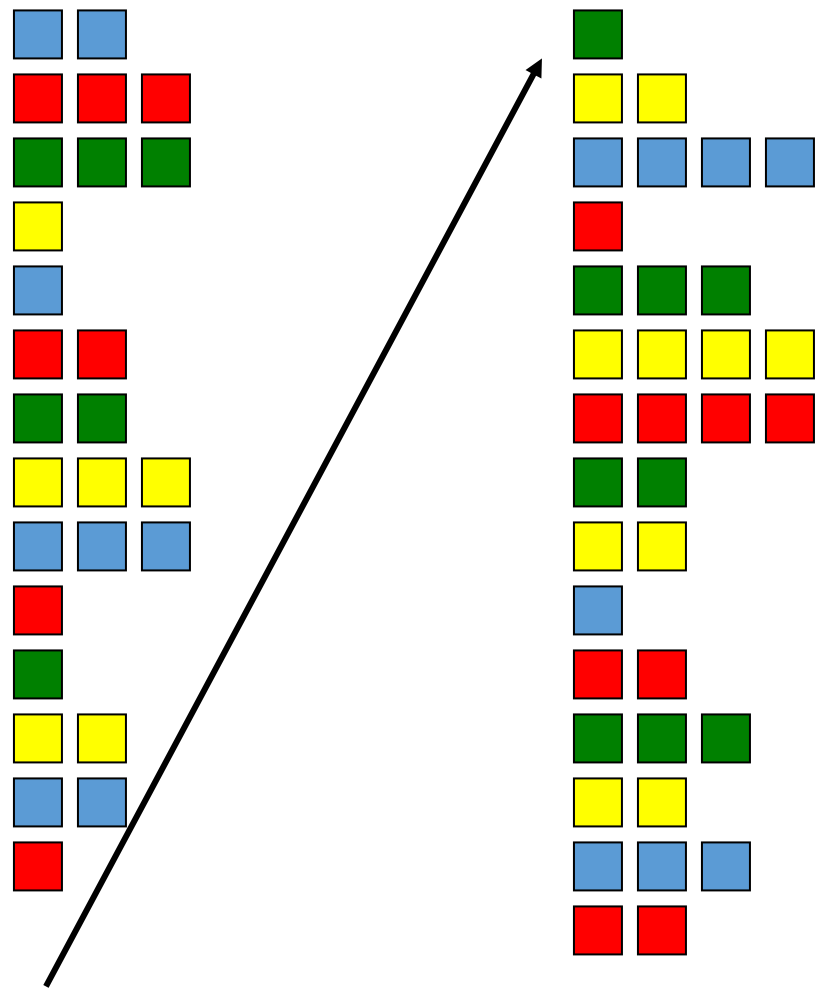
Fine-grained multithreading

- 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



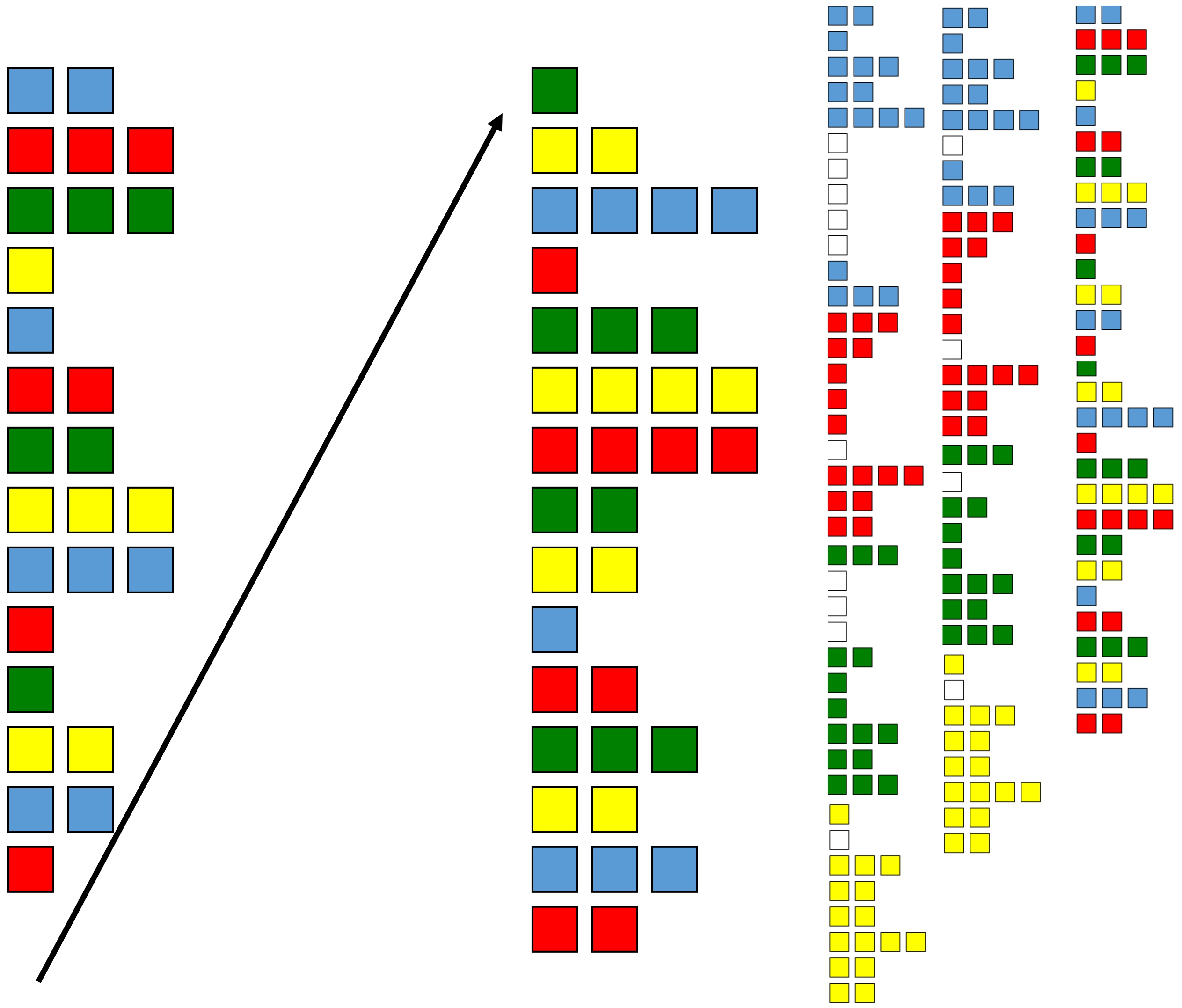
Fine-grained multithreading

- 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
- Doesn’t make full use of multi-issue



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



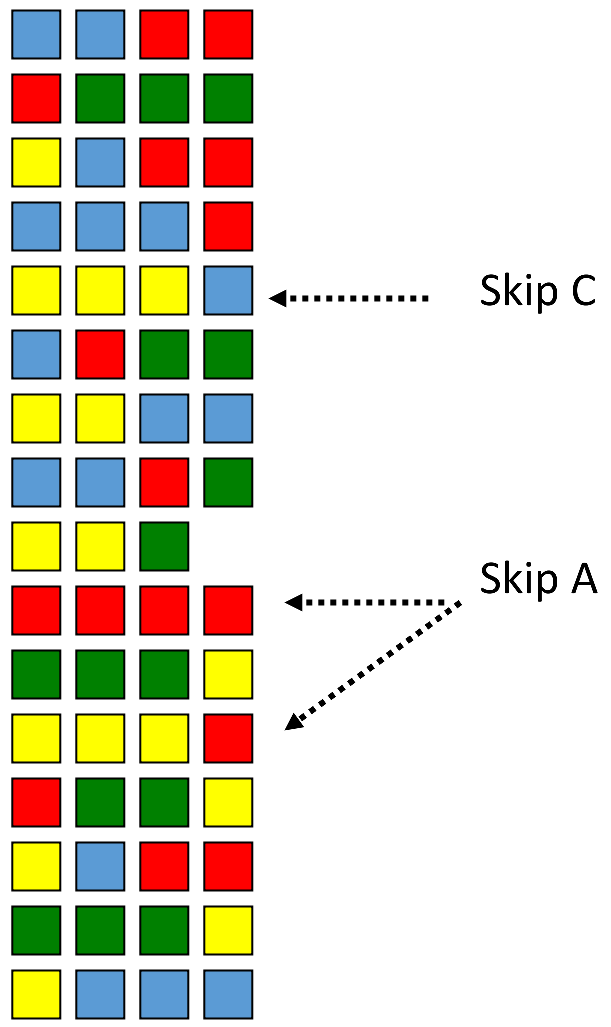
Simultaneous Multithreading (SMT)

Simultaneous Multithreading (SMT)

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

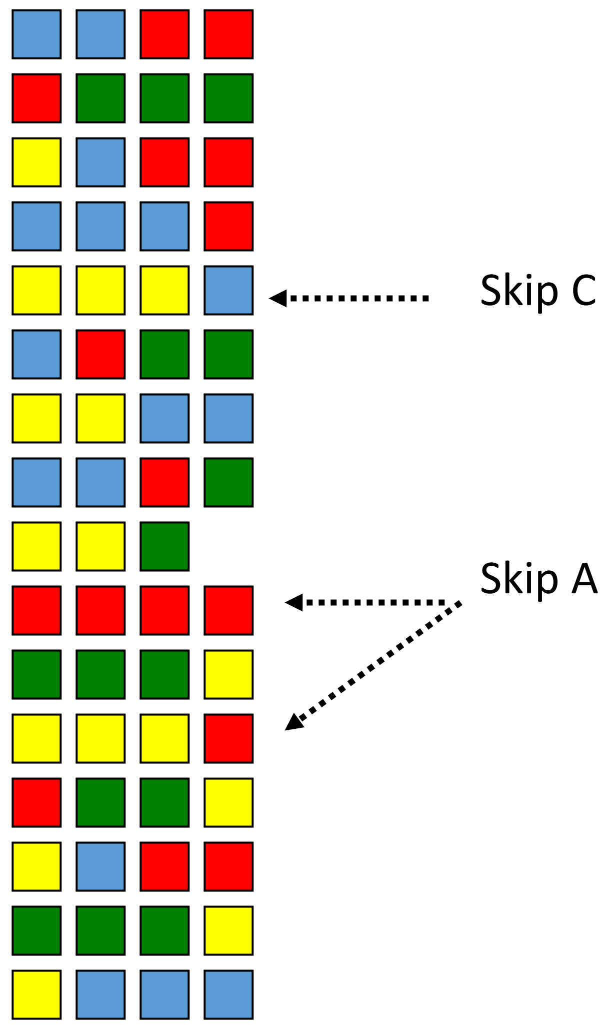
Simultaneous Multithreading (SMT)

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



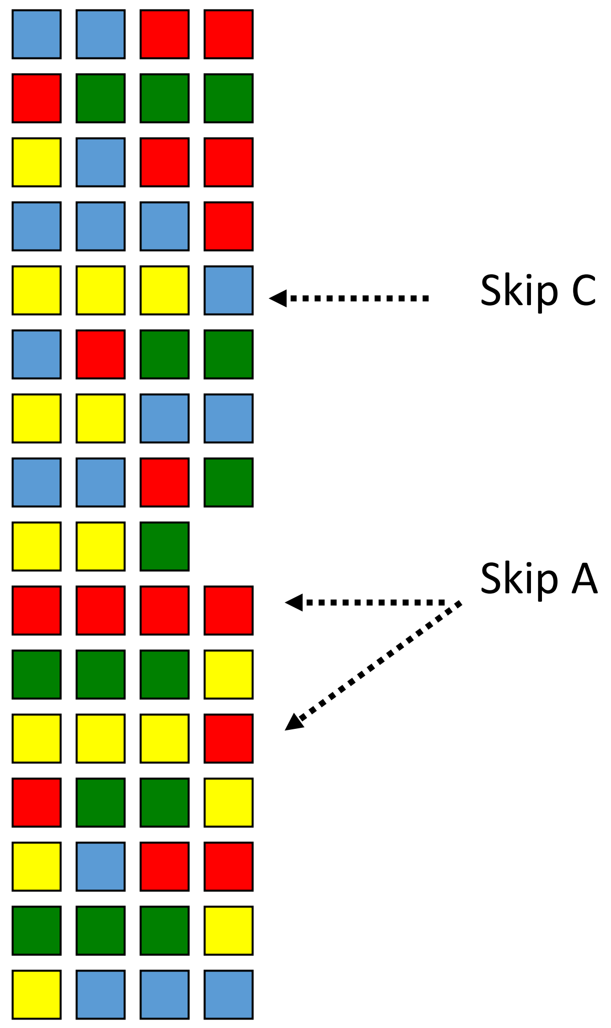
Simultaneous Multithreading (SMT)

- 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



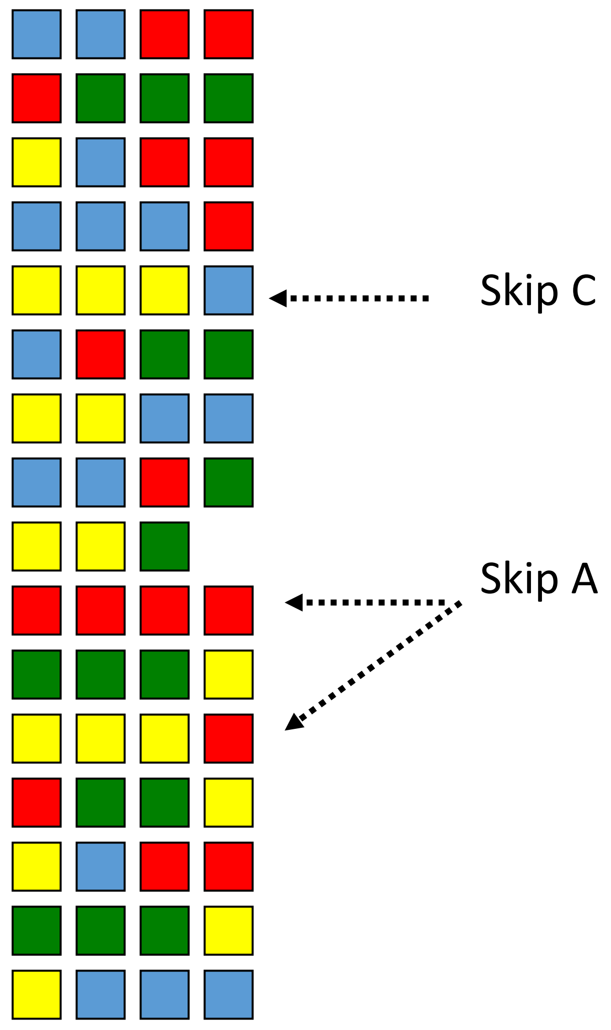
Simultaneous Multithreading (SMT)

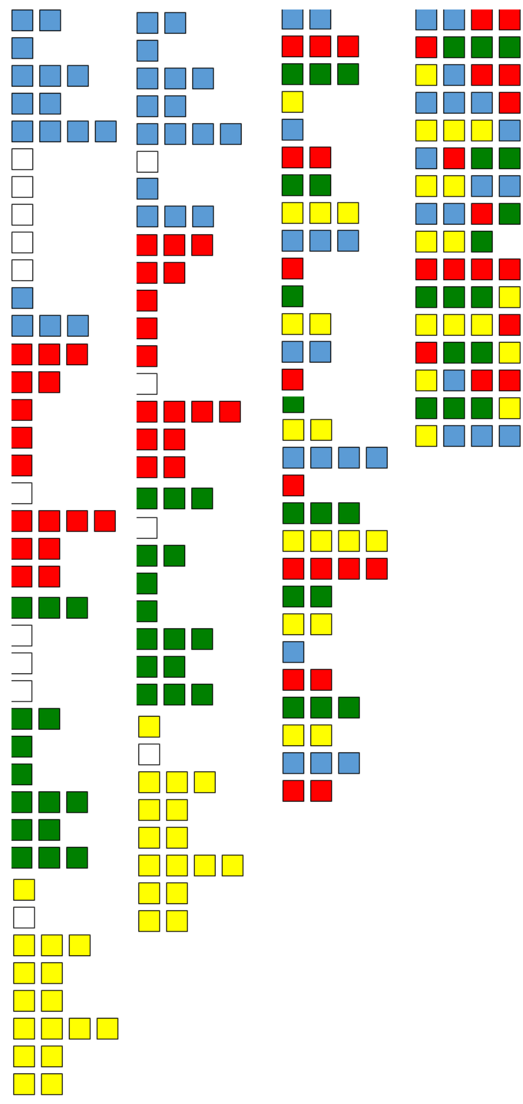
- Instructions from multiple threads issued on same cycle
  - Uses register renaming
  - dynamic scheduling facility of multi-issue architecture
- Hardware support:
  - Register files, PCs per thread
  - Temporary result registers pre commit
  - Support to sort out which threads get results from which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



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 ~= 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 $\rightarrow$ 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

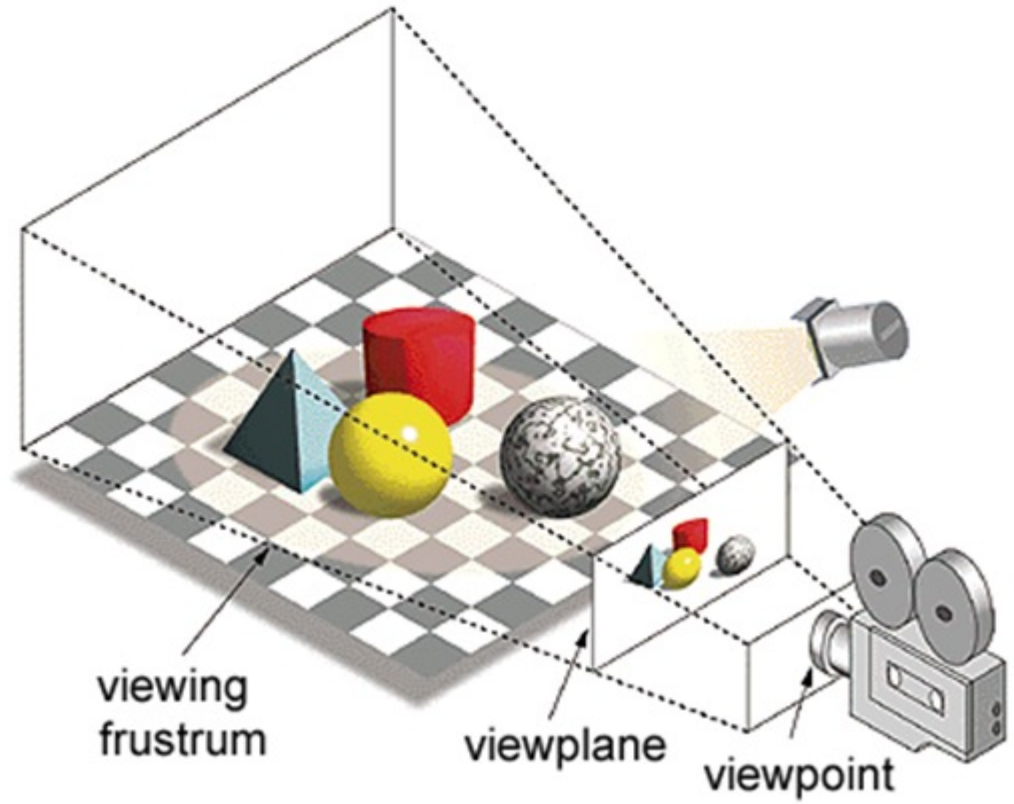
Graphics ~= Rendering

Inputs

- 3D world model(objects, materials)
  - Geometry modeled w triangle meshes, surface normals
  - GPUs subdivide triangles into “fragments” (rasterizat
  - 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



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

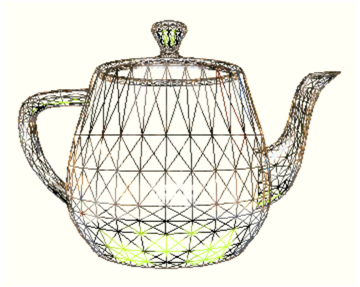
Grossly over-simplified rendering algorithm

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

Grossly over-simplified rendering algorithm

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

Grossly over-simplified rendering algorithm

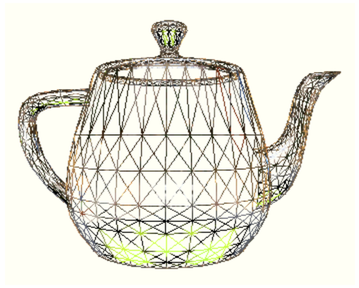


```
foreach(vertex v in model)
        map $v_{model} \rightarrow v_{view}$
fragment[] frags = {};
foreach triangle t ($v_{0,}$ $v_{1,}$ $v_2$)
```

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));
```

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
```

Grossly over-simplified rendering algorithm

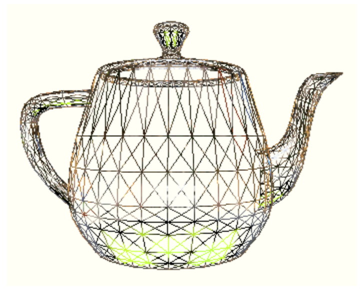
```
foreach(vertex v in model)
        map $v_{model} \rightarrow v_{view}$
fragment[] frags = {};
foreach triangle t ($v_{0,} v_{1,} v_2$)
        frags.add(rasterize(t));
foreach fragment f in frags
        choose_color(f);
```

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));
```

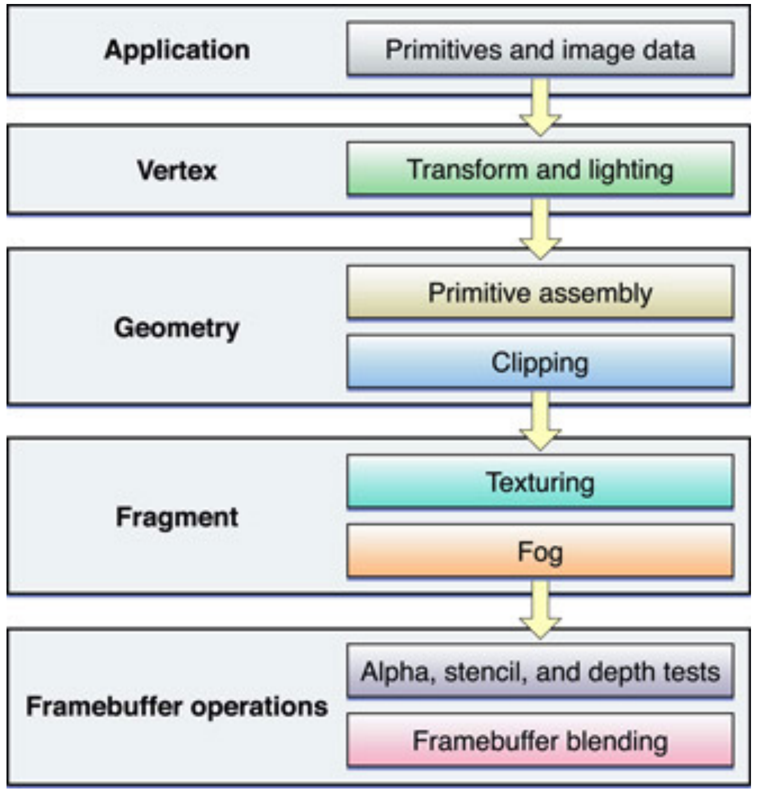
Grossly over-simplified rendering algorithm

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



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));
```

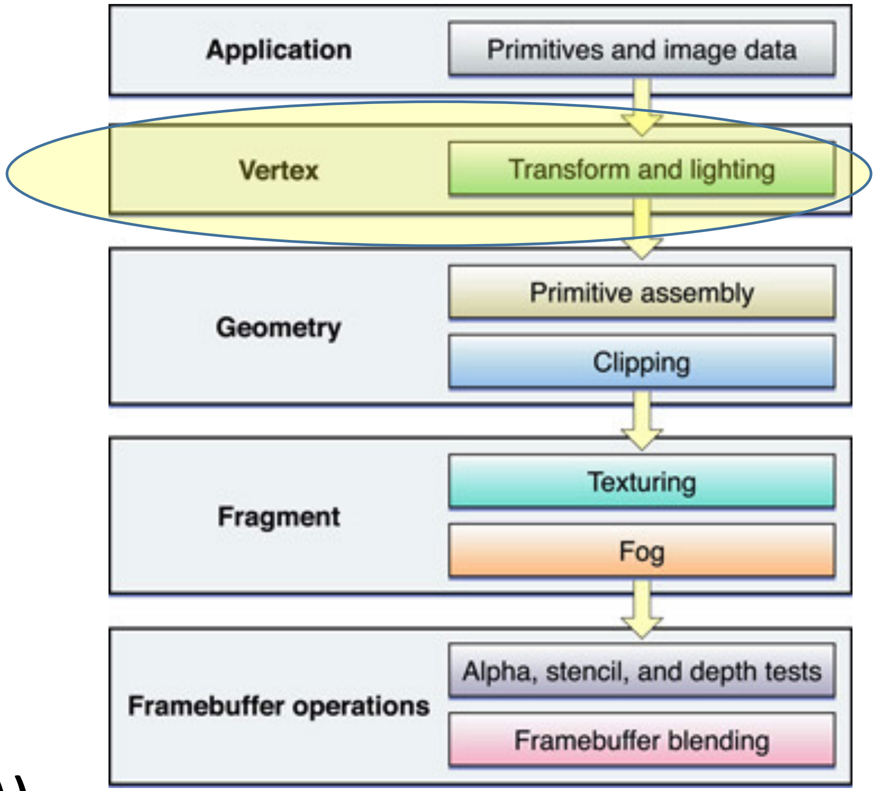


OpenGL pipeline

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));
```

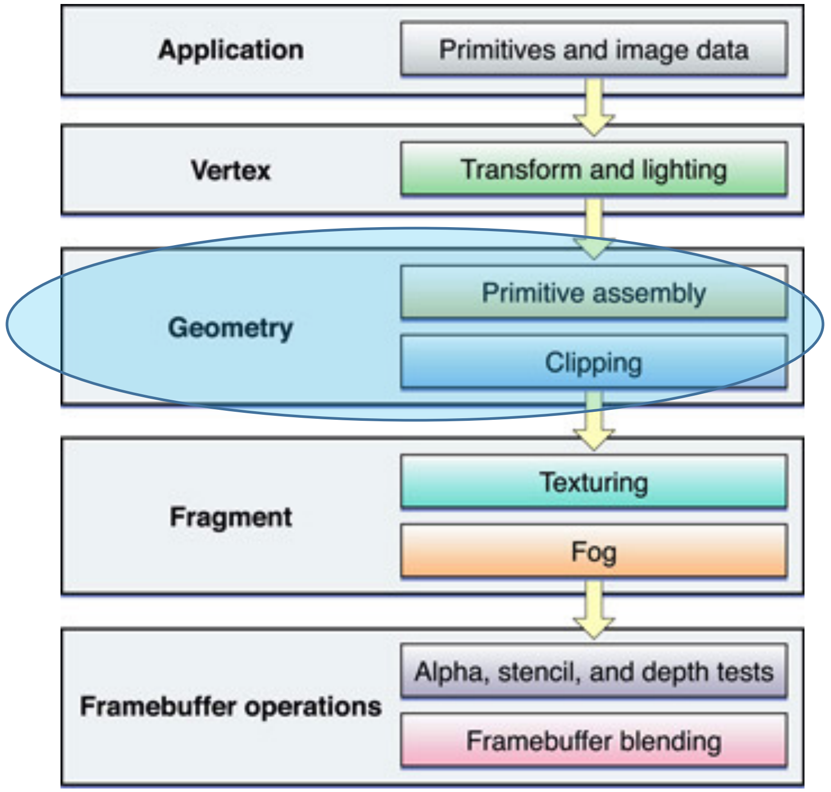


OpenGL pipeline

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));
```

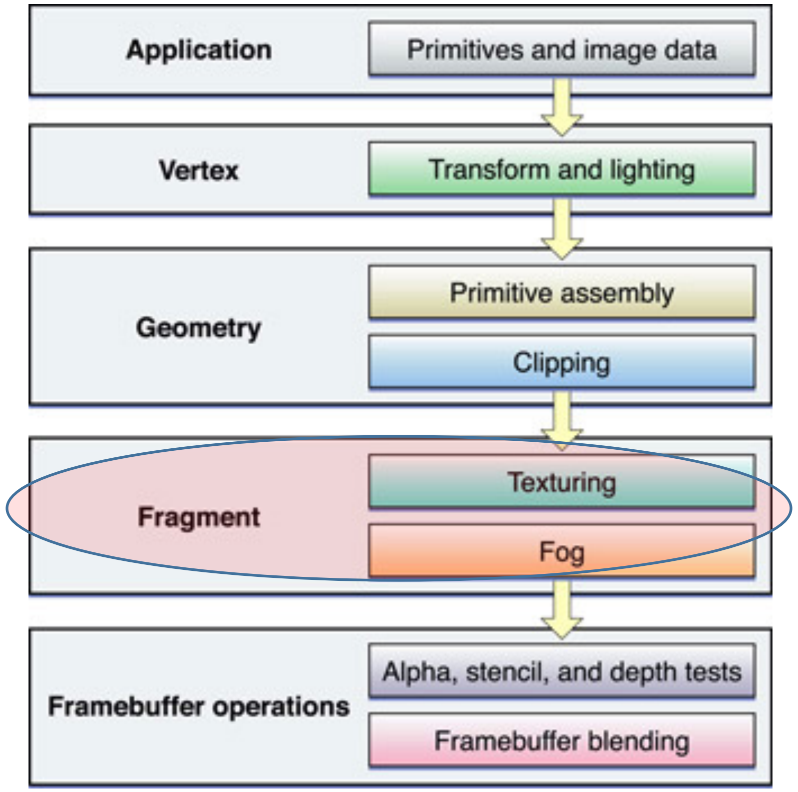


OpenGL pipeline

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));
```

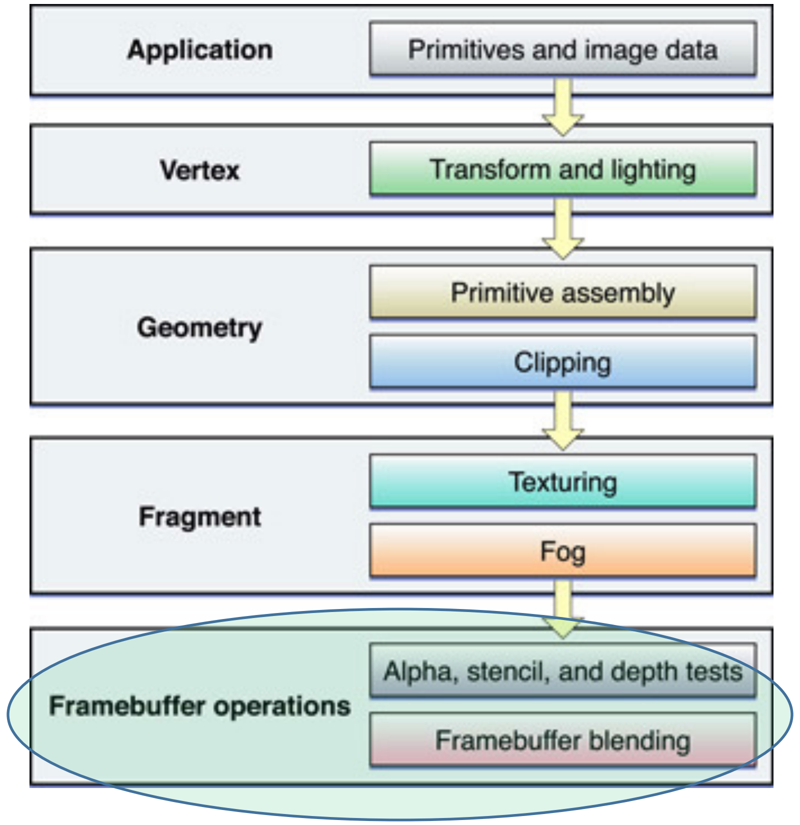


OpenGL pipeline

To first order, DirectX looks the same!

Algorithm → Graphics Pipeline

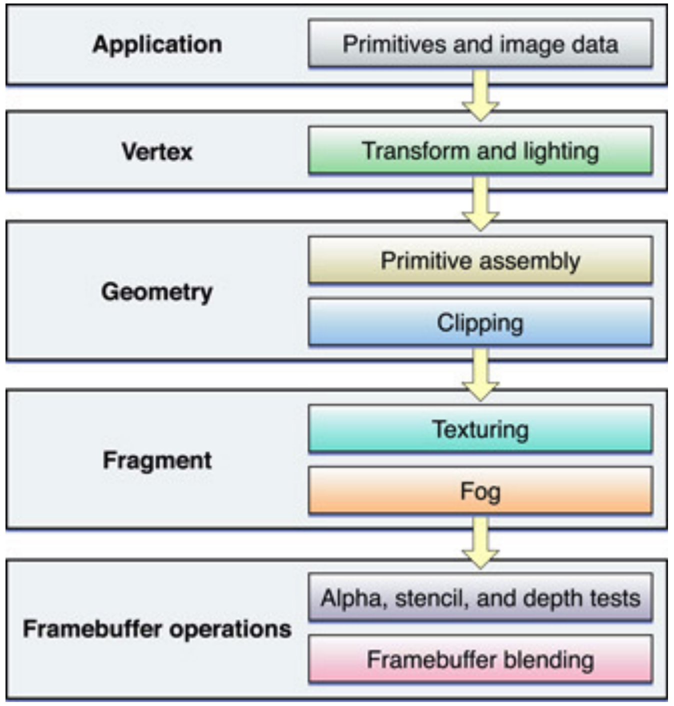
```
foreach(vertex v in model)
        map $v_{model} \rightarrow v_{view}$
fragment[] frags = {};
foreach triangle t ($v_0, v_1, v_2$)
        frags.add(rasterize(t));
foreach fragment f in frags
        choose_color(f);
display(visible_fragments(frags));
```

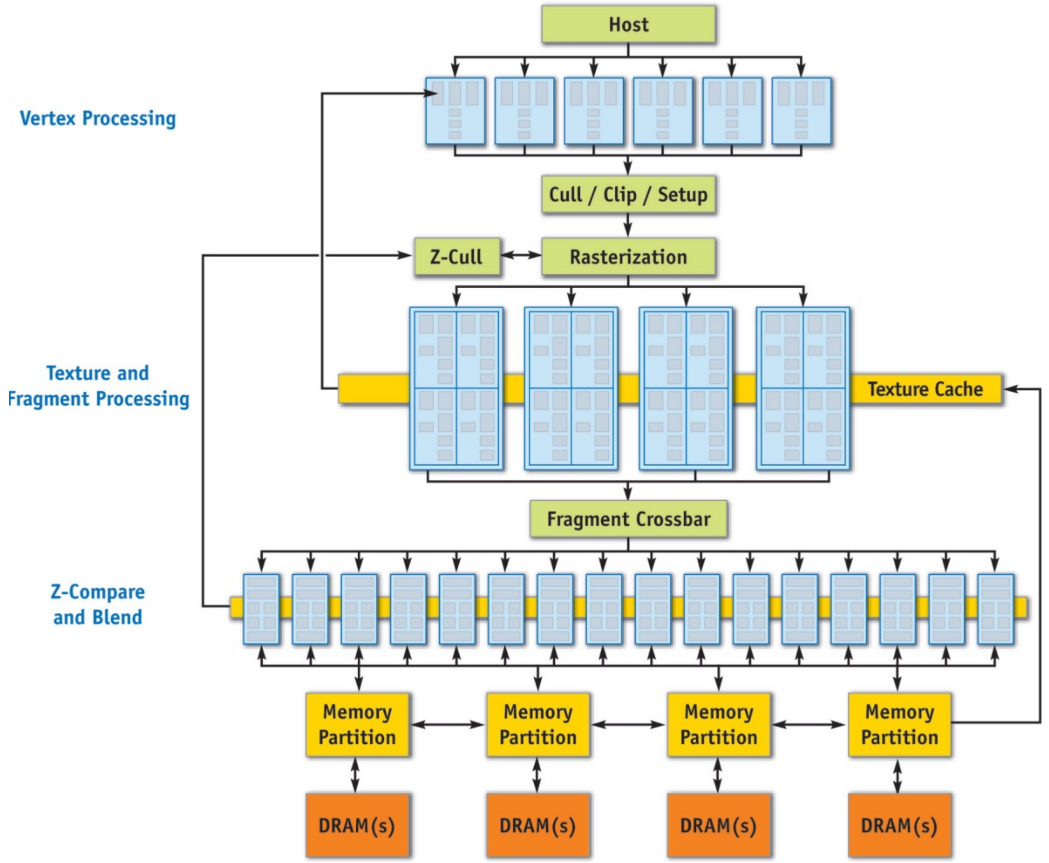


OpenGL pipeline

To first order, DirectX looks the same!

Graphics pipeline → GPU architecture

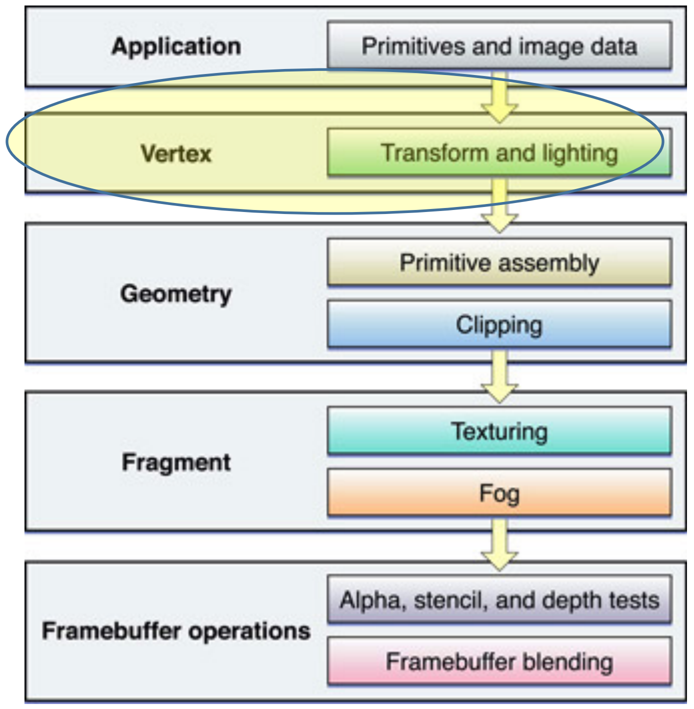


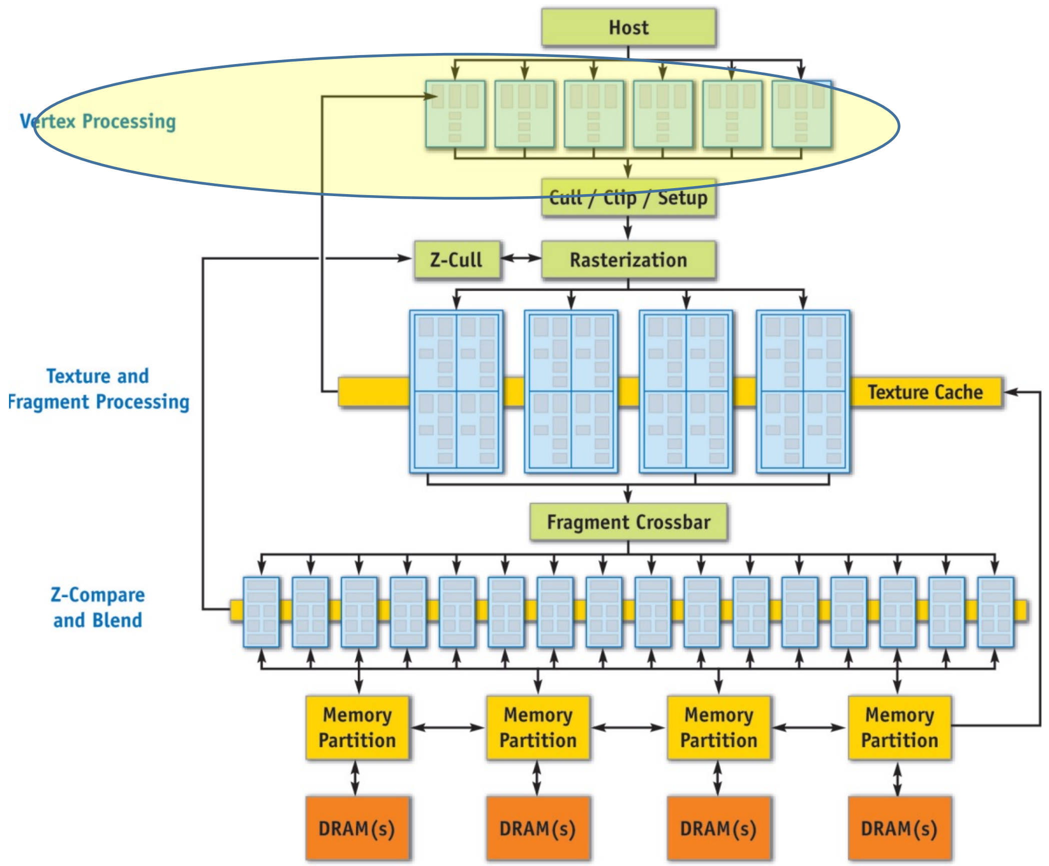


Limited "programmability" of shaders:
Minimal/no control flow
Maximum instruction count

GeForce 6 series

Graphics pipeline → GPU architecture

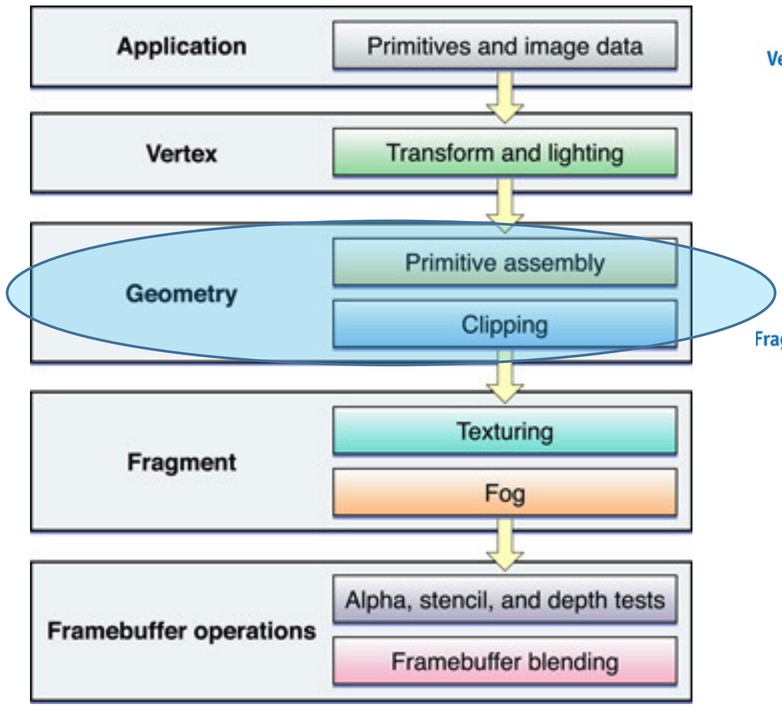


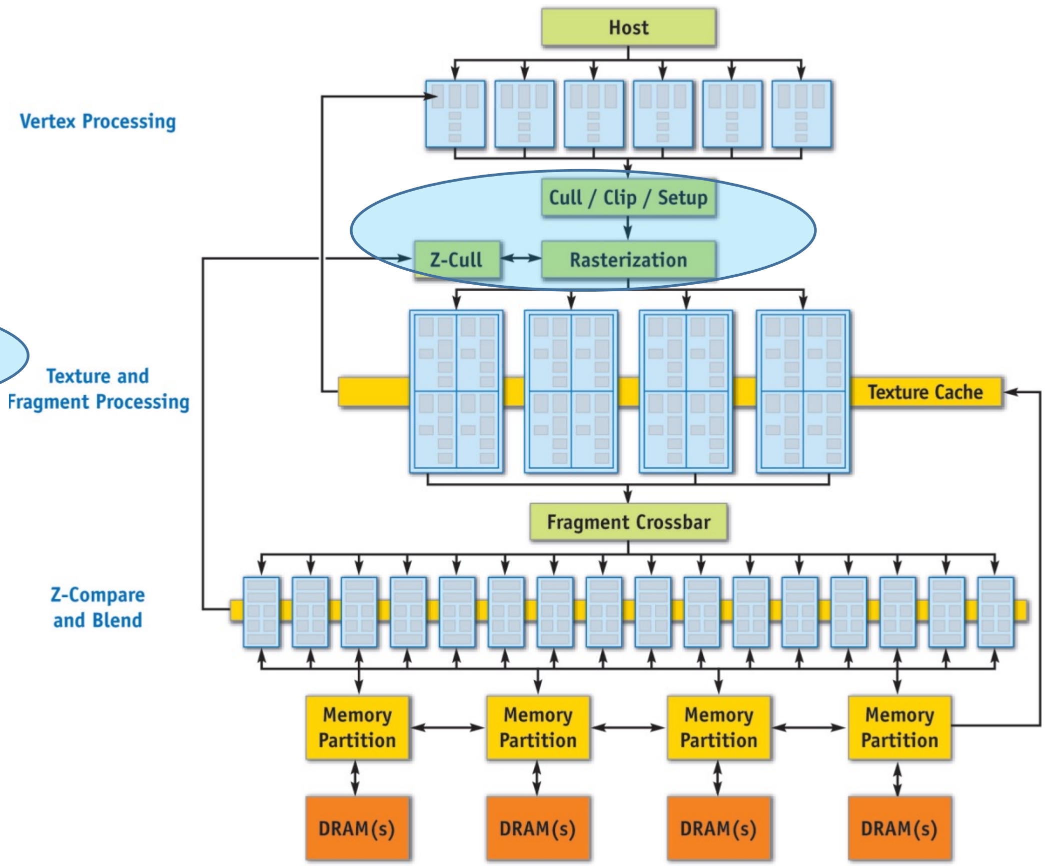


Limited "programmability" of shaders:
Minimal/no control flow
Maximum instruction count

GeForce 6 series

Graphics pipeline → GPU architecture

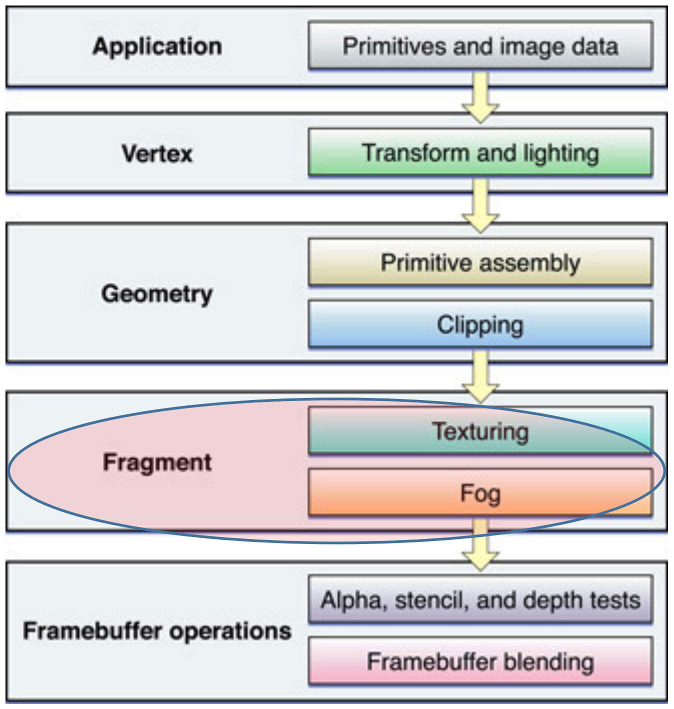


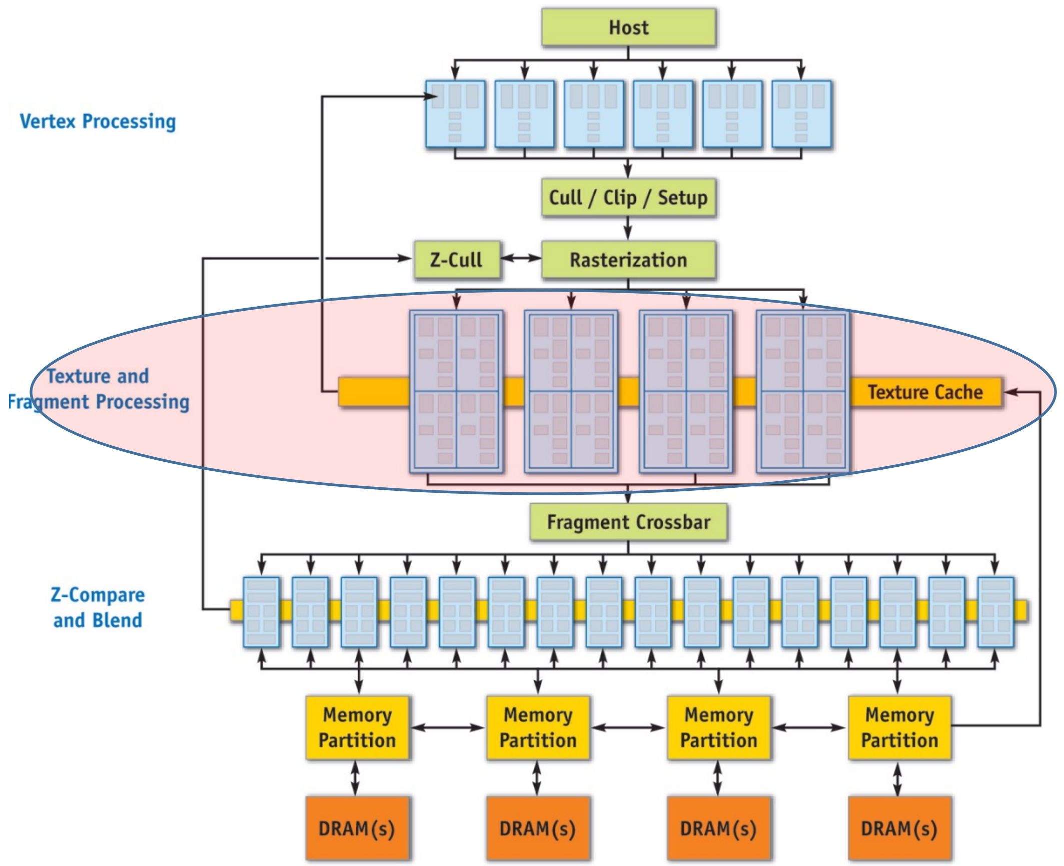


GeForce 6 series

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

Graphics pipeline → GPU architecture

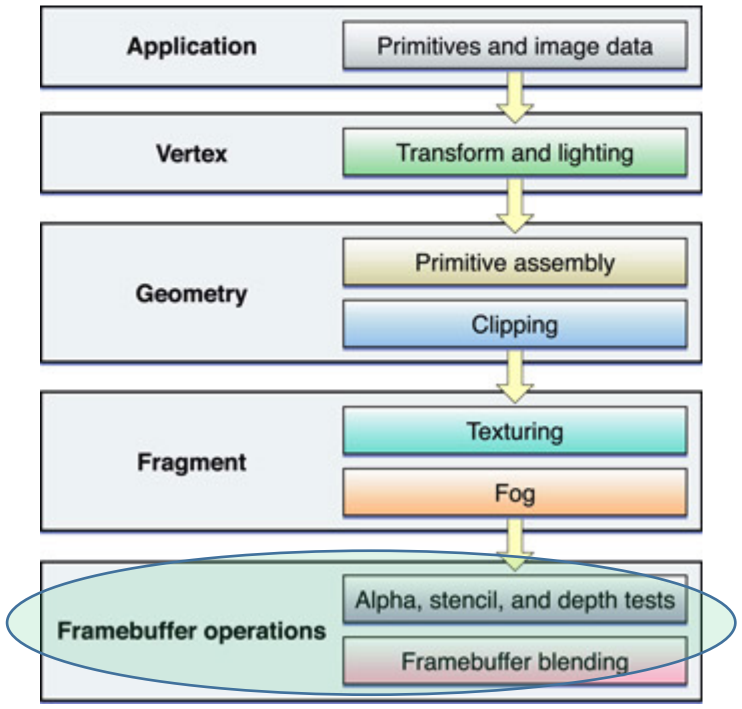


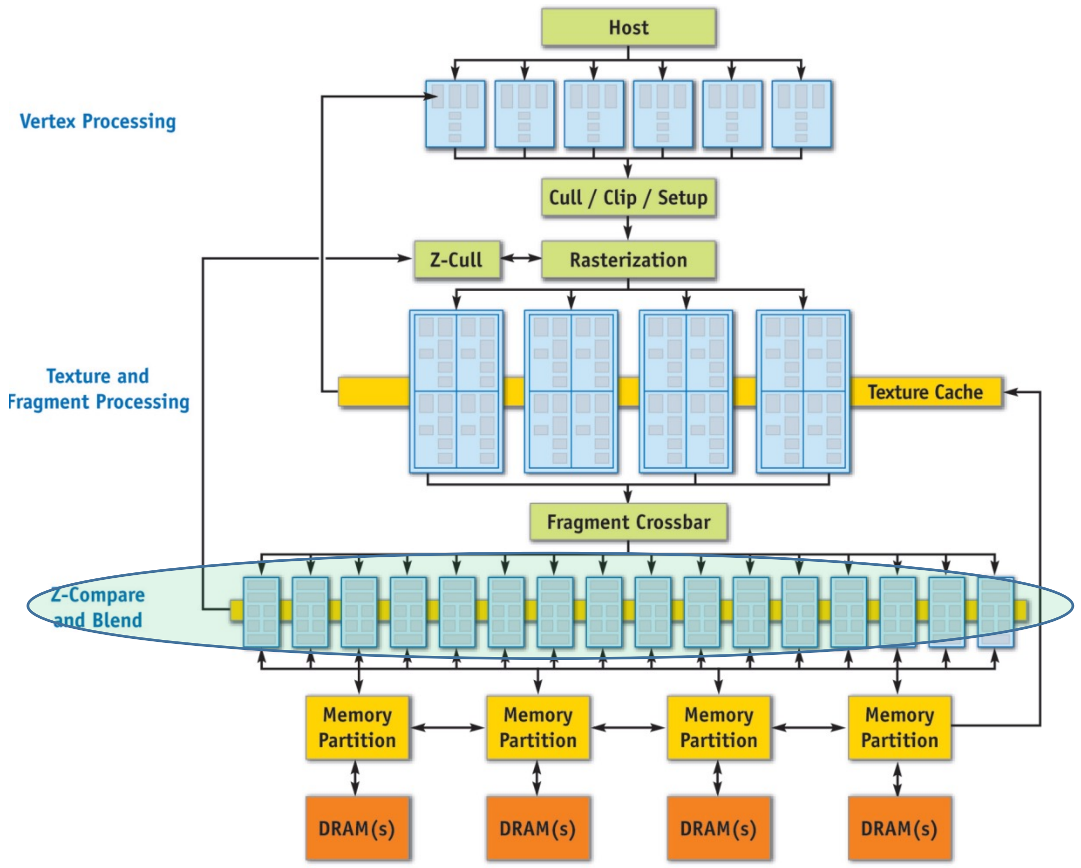


GeForce 6 series

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

Graphics pipeline → GPU architecture

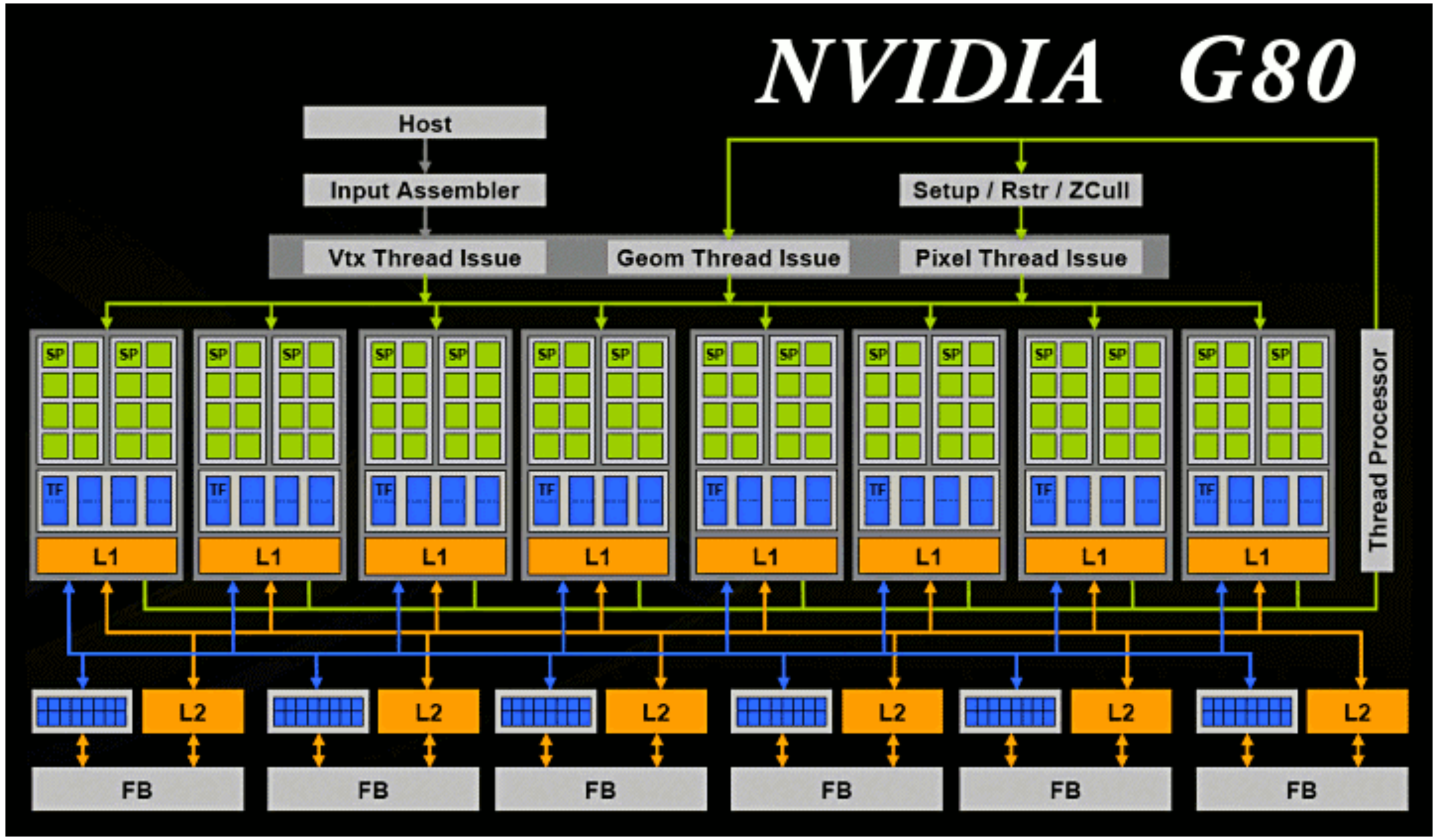




GeForce 6 series

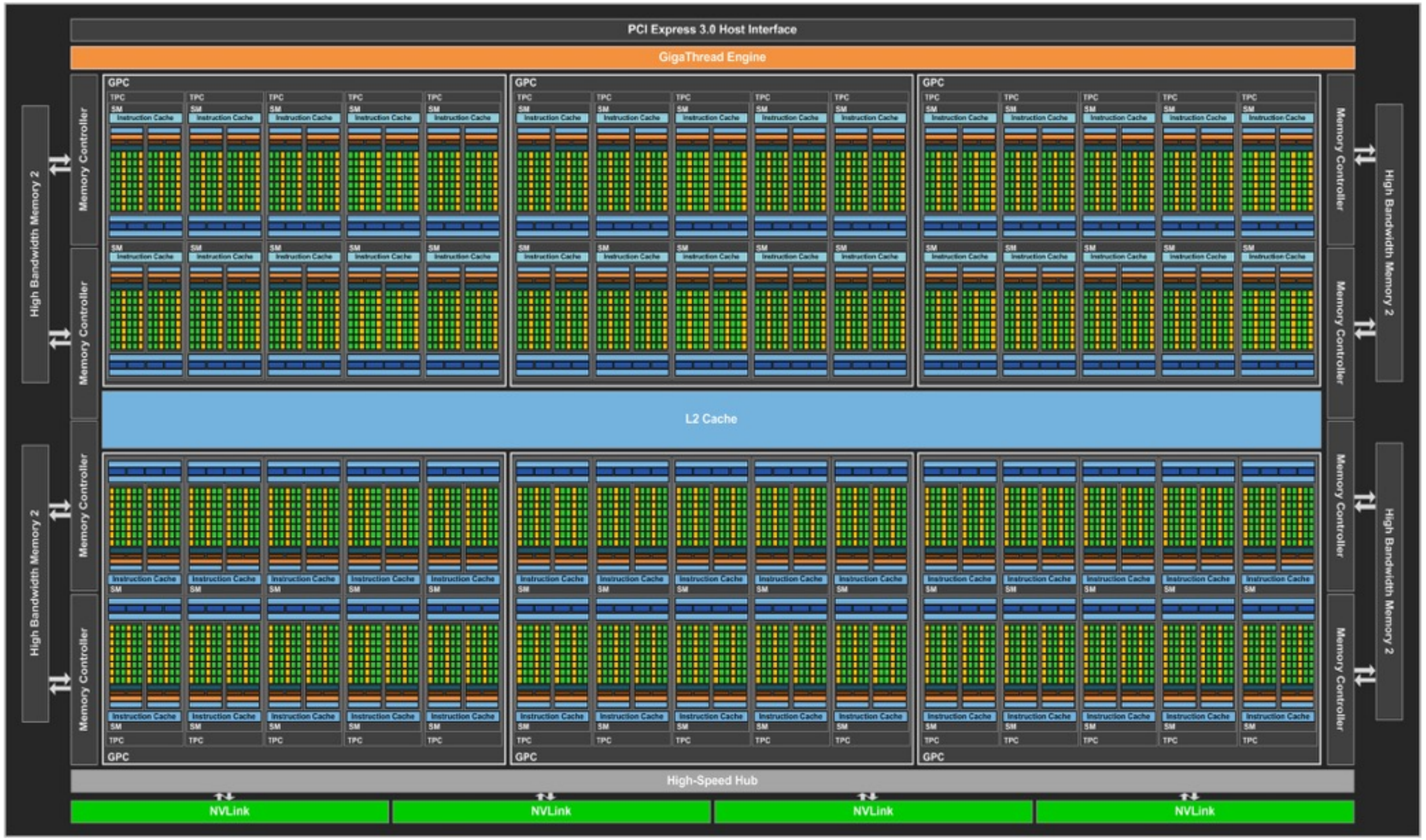
Limited "programmability" of shaders:
Minimal/no control flow
Maximum instruction count

Late Modernity: unified shaders

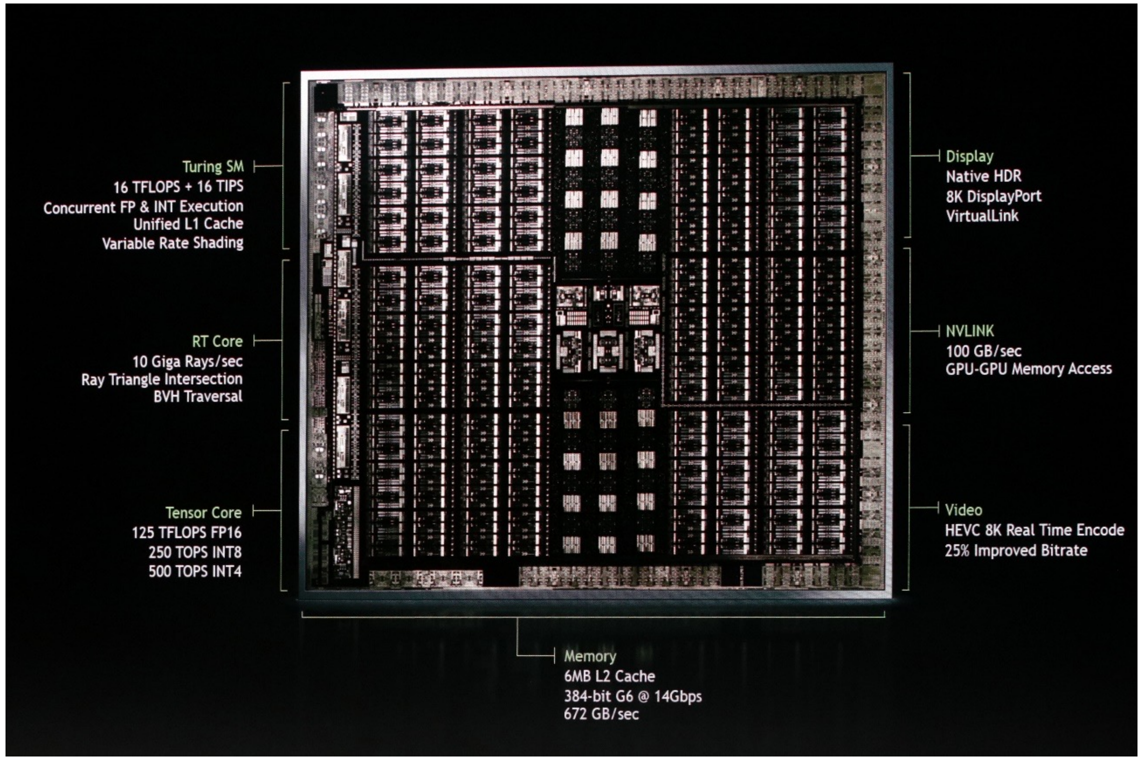


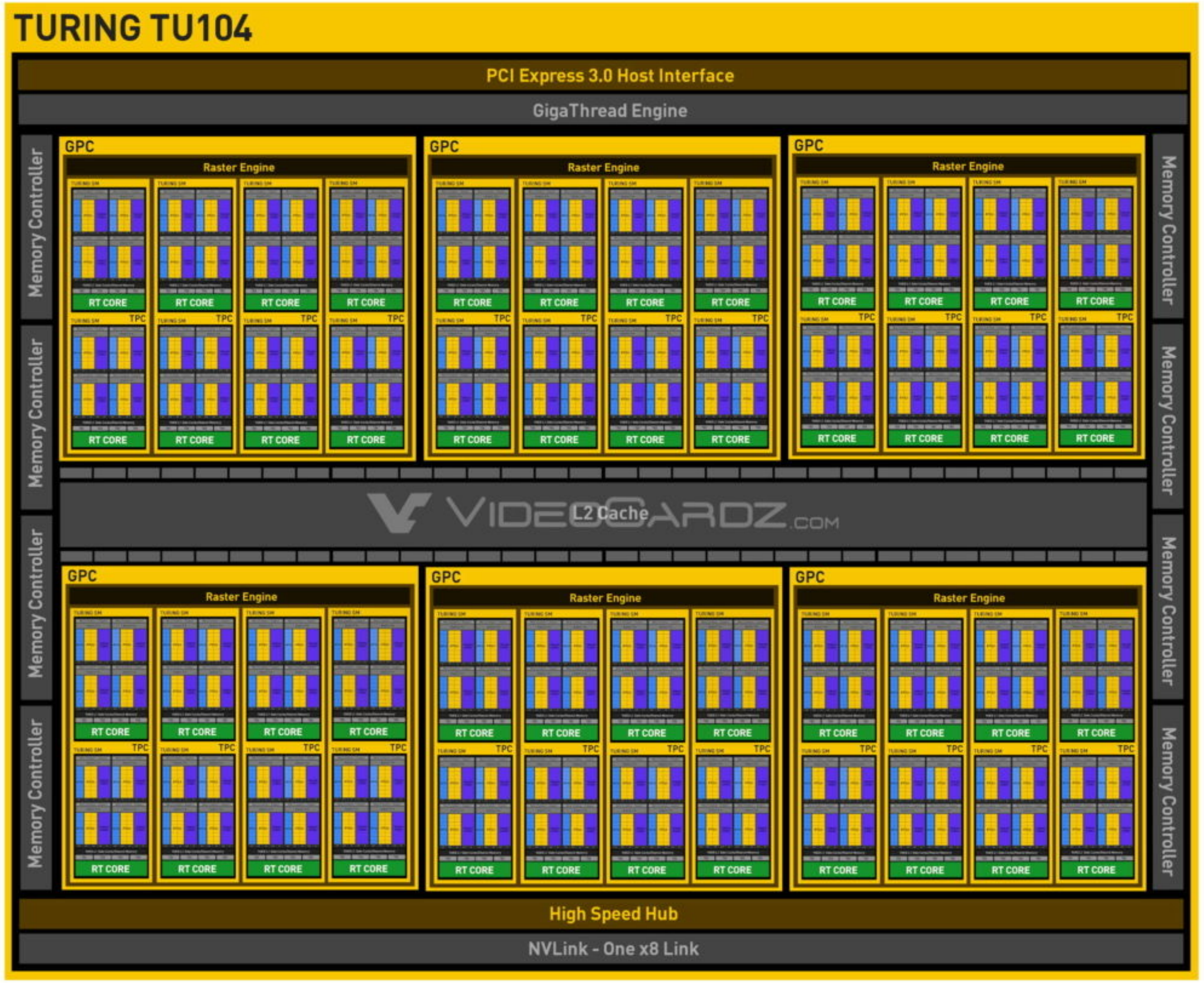
Mapping to Graphics pipeline no longer apparent
Processing elements no longer specialized to a particular role
Model supports *real* control flow, larger instr count

Mostly Modern: Pascal

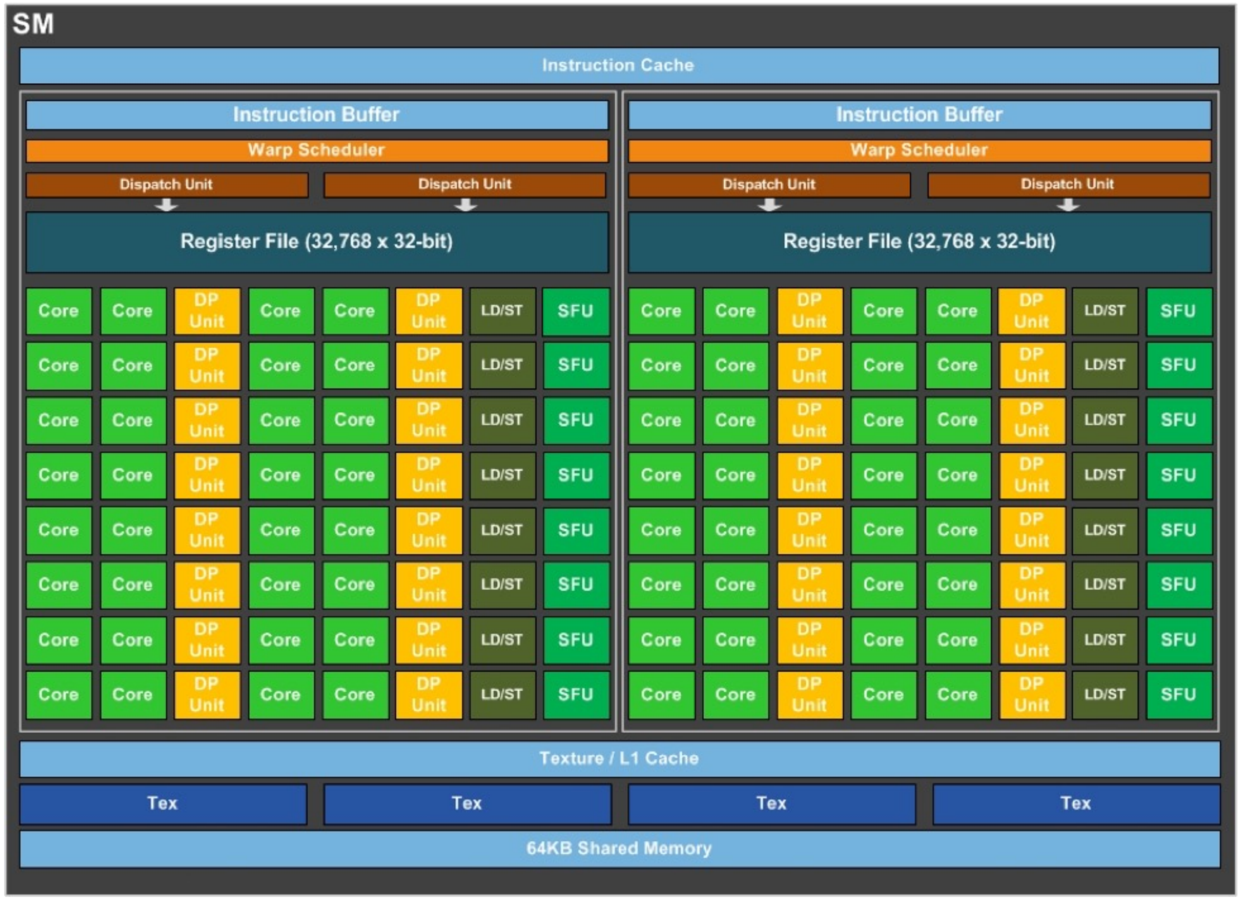


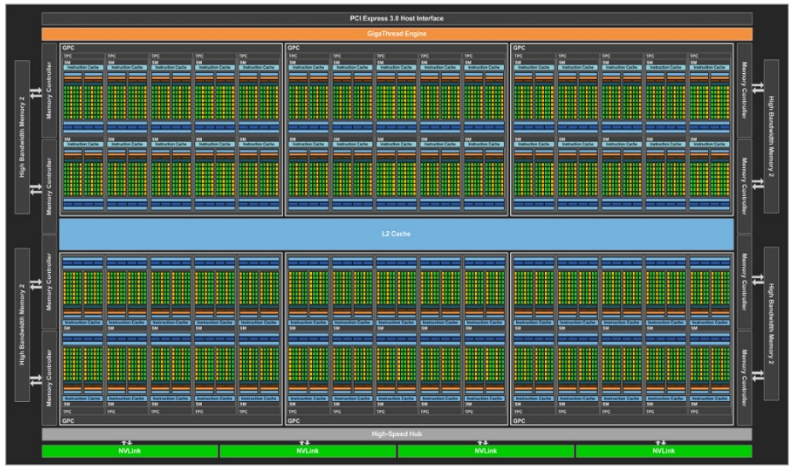
Definitely Modern: Turing





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

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

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

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

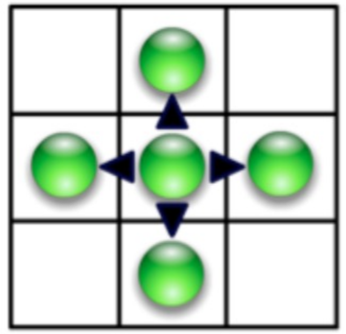
- Inputs
  - Array A
  - Function f(x)
- map(A, f) → apply f(x) on all elements in A
- Parallelism trivially exposed
  - f(x) can be applied in parallel to all elements, in principle

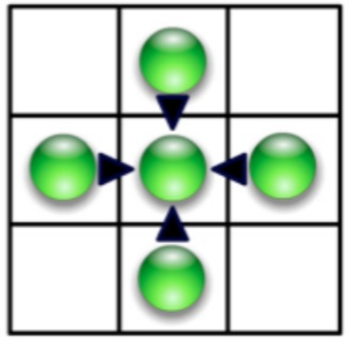
```
for(i=0; i<numPoints; i++) {
   labels[i] = findNearestCenter(points[i]);
}
```

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

Scatter and Gather

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


Scatter


Gather

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]];
```

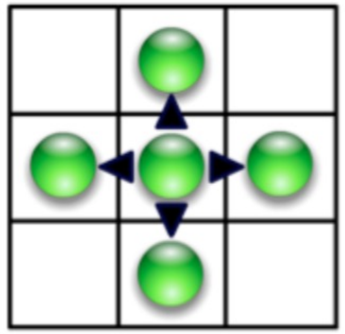
$\Rightarrow$

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

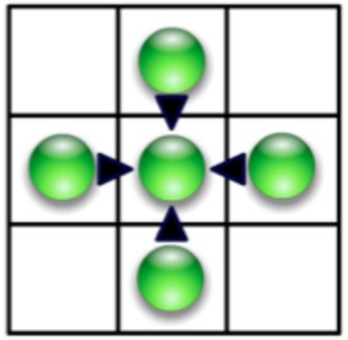
```
for (i=0; i<N; ++i)
  y[idx[i]] = x[i];
```

$\Rightarrow$

```
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 += (map(sqr, point[i]))
}
```

```
accum = reduce(+, map(sqr, point))
```

Reduce

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

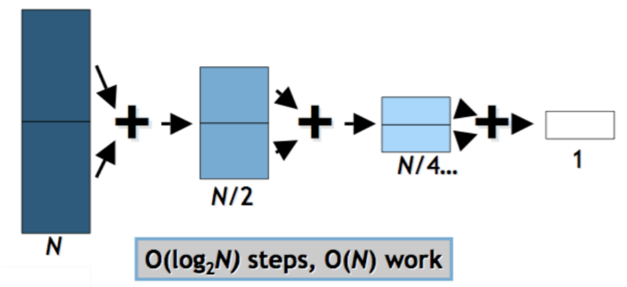
```
for(i=0; i<N; ++i) {
    accum += (map(sqr, point[i]))
}
```

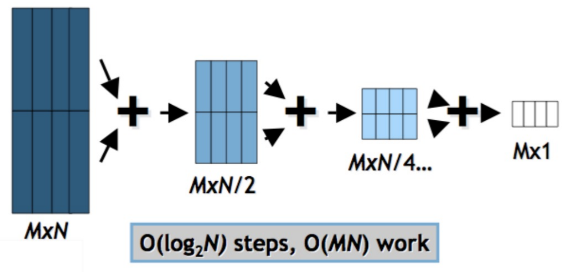
$\Rightarrow$

```
accum = reduce(+, map(sqr, 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 += (map(sqr, point[i]))
}
```

```
accum = reduce(+, map(sqr, point))
```

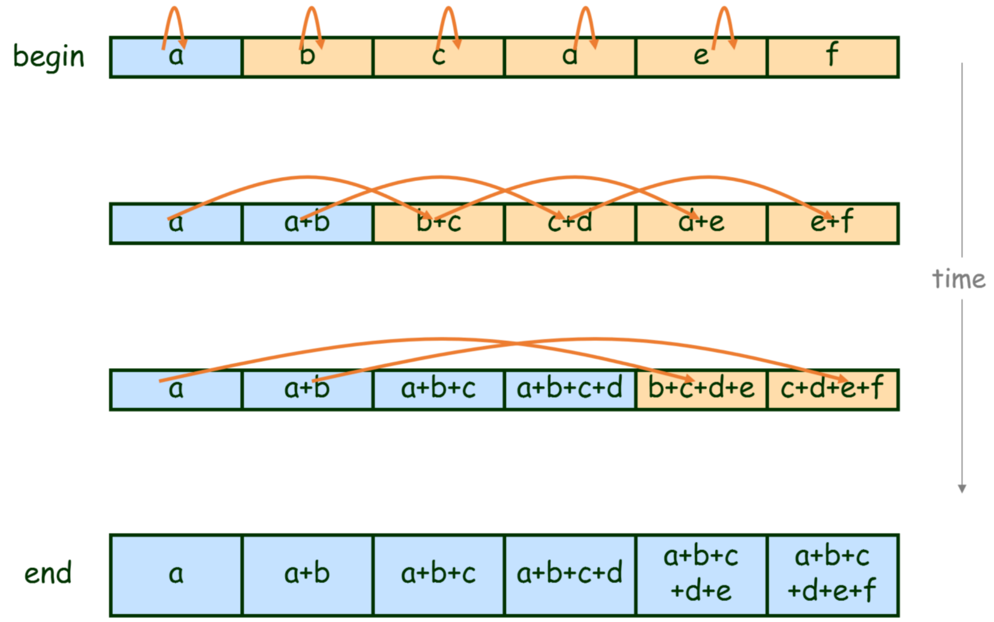
Why must op be associative?

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, …



Summary

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