Go Wrap up
Parallel Architectures

Chris Rossbach
cs378 Fall 2018
10/15/2018
Outline for Today

• Questions?
• Administrivia

• Agenda
  • Go
  • Parallel Architectures (GPU background)

• Rob Pike’s 2012 Go presentation is excellent, and I borrowed from it: https://talks.golang.org/2012/concurrency.slide
Faux Quiz questions

• How are promises and futures different or the same as goroutines
• What is the difference between a goroutine and a thread?
• What is the difference between a channel and a lock?
• How is a channel different from a concurrent FIFO?
• What is the CSP model?
• What are the tradeoffs between explicit vs implicit naming in message passing?
• What are the tradeoffs between blocking vs. non-blocking send/receive in a shared memory environment? In a distributed one?
• 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 programmer?
• List some ways that GPUs use concurrency but not necessarily parallelism.
Google Search

• Workload:
• Accept query
• Return page of results (with ugh, ads)
• Get search results by sending query to
  • Web Search
  • Image Search
  • YouTube
  • Maps
  • News, etc
• How to implement this?
Search 1.0

• Google function takes query and returns a slice of results (strings)
• Invokes Web, Image, Video search serially

```go
func Google(query string) (results []Result) {
    results = append(results, Web(query))
    results = append(results, Image(query))
    results = append(results, Video(query))
    return
}
```
Search 2.0

• Run Web, Image, Video searches concurrently, wait for results
• No locks, conditions, callbacks

```go
func Google(query string) (results []Result) {
    c := make(chan Result)
    go func() { c <- Web(query) } ()
    go func() { c <- Image(query) } ()
    go func() { c <- Video(query) } ()

    for i := 0; i < 3; i++ {
        result := <-c
        results = append(results, result)
    }
    return
}
```
Search 2.1

- Don’t wait for slow servers: No locks, conditions, callbacks!

```go
    c := make(chan Result)
    go func() { c <- Web(query) } ()
    go func() { c <- Image(query) } ()
    go func() { c <- Video(query) } ()

    timeout := time.After(80 * time.Millisecond)
    for i := 0; i < 3; i++ {
        select {
            case result := <-c:
                results = append(results, result)
            case <-timeout:
                fmt.Println("timed out")
                return
        }
    }
```

return
Search 3.0

• Reduce tail latency with replication. No locks, conditions, callbacks!

c := make(chan Result)
go func() { c <- First(query, Web1, Web2) } ()
go func() { c <- First(query, Image1, Image2) } ()
go func() { c <- First(query, Video1, Video2) } ()
timeout := time.After(80 * time.Millisecond)
for i := 0; i < 3; i++ {
    select {
        case result := <-c:
            results = append(results, result)
        case <-timeout:
            fmt.Println("timed out")
            return
    }
}
return

func First(query string, replicas ...Search) Result {
    c := make(chan Result)
    searchReplica := func(i int) { c <- replicas[i](query) }
    for i := range replicas {
        go searchReplica(i)
    }
    return <-c
}
Go: magic? ...or *threadpools and concurrent Qs*?

- We’ve seen several abstractions for
  - Control flow/execution
  - Communication
- Lots of discussion of pros and cons
- Ultimately still CPUs + instructions
- Go: just sweeping issues under the language interface?
  - Why is it OK to have 100,000s of goroutines?
  - Why isn’t composition an issue?
Go implementation details
Go implementation details

- \( M = \text{"machine"} \rightarrow \text{OS thread} \)
Go implementation details

- M = “machine” \(\rightarrow\) OS thread
- P = (processing) context
Go implementation details

- M = “machine” → OS thread
- P = (processing) context
- G = goroutines
Go implementation details

- **M** = “machine” → OS thread
- **P** = (processing) context
- **G** = goroutines
- Each ‘M’ has a queue of goroutines
Go implementation details

- M = “machine” → OS thread
- P = (processing) context
- G = goroutines
- Each ‘M’ has a queue of goroutines
- Goroutine scheduling is cooperative
  - Switch out on complete or block
  - Very light weight (fibers!)
  - Scheduler does work-stealing
Go implementation details

- **M** = “machine” → OS thread
- **P** = (processing) context
- **G** = goroutines

```go
struct G {
  byte* stackguard; // stack guard information
  byte* stackbase;  // base of stack
  byte* stack0;     // current stack pointer
  byte* entry;      // initial function
  void* param;     // passed parameter on wakeup
  int16 status;    // status
  int32 goid;      // unique id
  M* lockedm;      // used for locking M’s and G’s
}
```
Go implementation details

- M = “machine” → OS thread
- P = (processing) context
- G = goroutines
- Each ‘M’ has a queue of goroutines
- Goroutine scheduling is cooperative
  - Switch out on complete or block
  - Very light weight (fibers!)
  - Scheduler does work-stealing
Go implementation details

- \( M \) = “machine” → OS thread
- \( P \) = (processing) context
- \( G \) = goroutines

Each “machine” has a queue of goroutines.

Goroutine scheduling is cooperative - switch out on complete or block.

Very lightweight (fibers!)

Scheduler does work - stealing

```go
struct M {
    G* curg;          // current running goroutine
    int32 id;         // unique id
    int32 locks;      // locks held by this M
    MCache *mcache;  // cache for this thread
    G* lockedg;       // used for locking M’s and G’s
    uintptr createsstack [32]; // Stack that created this thread
    M* nextwaitm;     // next M waiting for lock
...
};
```
Go implementation details

- $M =$ “machine” $\rightarrow$ OS thread
- $P =$ (processing) context
- $G =$ goroutines
- Each $M$ has a queue of goroutines
- Goroutine scheduling is cooperative
  - Switch out on complete or block
  - Very light weight (fibers!)
- Scheduler does work - stealing

```go
def struct Sched {
    Lock; // global sched lock.
    // must be held to edit G or M queues
    G *gfree; // available g’s (status == Gdead)
    G *ghead; // g’s waiting to run queue
    G *gtail; // tail of g’s waiting to run queue
    int32 gwait; // number of g’s waiting to run
    int32 gcount; // number of g’s that are alive
    int32 grunning; // number of g’s running on cpu
    // or in syscall
    M *mhead; // m’s waiting for work
    int32 mwait; // number of m’s waiting for work
    int32 mcount; // number of m’s that have been created
}
```
Go implementation details

- M = “machine” → OS thread
- P = (processing) context
- G = goroutines

Each ‘M’ has a queue of goroutines
Goroutine scheduling is cooperative
  - Switch out on complete or block
  - Very light weight (fibers!)
  - Scheduler does work
    - stealing

```go
struct Sched {
  Lock;  // global sched lock.
    // must be held to edit G or M queues
  G *gfree;  // available g’s (status == Gdead)
  G *ghead;  // g’s waiting to run queue
  G *gtail;   // tail of g’s waiting to run queue
  int32 gwait;  // number of g’s waiting to run
  int32 gcount;  // number of g’s that are alive
  int32 grunning;  // number of g’s running on cpu
    // or in syscall
  M *mhead;  // m’s waiting for work
  int32 mwait;  // number of m’s waiting for work
  int32 mcount;  // number of m’s that have been created
...
};
```
Go implementation details

- \( M = \text{“machine”} \rightarrow \text{OS thread} \)
- \( P = \text{(processing) context} \)
- \( G = \text{goroutines} \)
- Each 'M' has a queue of goroutines
- Goroutine scheduling is cooperative
  - Switch out on complete or block
  - Very lightweight (fibers!)
  - Scheduler does work - stealing

```
struct Sched {
    Lock;    // global sched lock.
             // must be held to edit G or M queues
    G *gfree;    // available g's (status == Gdead)
    G *ghead;    // g's waiting to run queue
    G *gtail;    // tail of g's waiting to run queue
    int32 gwait;    // number of g's waiting to run
    int32 gcount;    // number of g's that are alive
    int32 grunning;    // number of g's running on cpu
                       // or in syscall
    M *mhead;    // m's waiting for work
    int32 mwait;    // number of m's waiting for work
    int32 mcount;    // number of m's that have been created
};
```
func testQ(consumers int) {
    startTimes["testQ"] = time.Now()
    var wg sync.WaitGroup
    wg.Add(consumers)
    ch := make(chan int)
    for i:=0; i<consumers; i++ {
        go func(id int) {
            aval, amore := <- ch
            if(amore) {
                info("reader #%d got %d value\n", id, aval)
            } else {
                info("channel reader #%d terminated with nothing.\n", id)
            }
            wg.Done()
        }(i)
    }
    time.Sleep(1000 * time.Millisecond)
    close(ch)
    wg.Wait()
    stopTimes["testQ"] = time.Now()
}
func testQ(consumers int) {
    startTimes["testQ"] = time.Now()
    var wg sync.WaitGroup
    wg.Add(consumers)
    ch := make(chan int)
    for i := 0; i < consumers; i++ {
        go func(id int) {
            aval, amore := <- ch
            if amore {
                info("reader #%d got %d value\n", id, aval)
            } else {
                info("channel reader #%d terminated with nothing.\n", id)
            }
            wg.Done()
        }(i)
    }
    time.Sleep(1000 * time.Millisecond)
    close(ch)
    wg.Wait()
    stopTimes["testQ"] = time.Now()
}
```
func testQ(consumers int) {
    startTimes["testQ"] = time.Now()
    var wg sync.WaitGroup
    wg.Add(consumers)
    ch := make(chan int)
    for i := 0; i < consumers; i++ {
        go func(id int) {
            aval, amore := <- ch
            if amore {
                info("reader #%d got %d value\n", id, aval)
            } else {
                info("channel reader #%d terminated with nothing.\n", id)
            }
            wg.Done()
        }(i)
    }
    time.Sleep(1000 * time.Millisecond)
    close(ch)
    wg.Wait()
    stopTimes["testQ"] = time.Now()
}
```
Channel implementation

• You can just read it:
  • https://golang.org/src/runtime/chan.go

• Some highlights
Channel implementation

```go
func chansend(c *chan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
    if c == nil {
        if !block {
            return false
        }
        gopark(nil, nil, "channel send (nil chan)", traceEvGoStop, 2)
        throw("unreachable")
    }

    if debugChan {
        print("chansend: chan=", c, "\n")
    }

    if raceenabled {
        racereadpc(unsafe.Pointer(c), callerpc, funcPC(chansend))
    }
}
```

• You can just read it:

• https://golang.org/src/runtime/chan.go

• Some highlights
Channel implementation

- You can just read it:
  https://golang.org/src/runtime/chan.go

- Some highlights
  Race detection! Cool!

```go
func chansend(c *chan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
    if c == nil {
        if !block {
            return false
        }

        gopark(nil, nil, "chan send (nil chan)", traceEvGoStop, 2)
        throw("unreachable")
    }

    if debugChan {
        print("chansend: chan=", c, "\n")
    }

    if raceenabled {
        racereadpc(unsafe.Pointer(c), callerpc, funcPC(chansend))
    }
}
```
Channel implementation

- You can just read it:
  - [https://golang.org/src/runtime/chan.go](https://golang.org/src/runtime/chan.go)
- Some highlights
Channel implementation

• You can just read it:
  • https://golang.org/src/runtime/chan.go

• Some highlights

```go
if sg := c.recvq.dequeue(); sg != nil {
    // Found a waiting receiver. We pass the value we want to send
    // directly to the receiver, bypassing the channel buffer (if any).
    send(c, sg, ep, func() { unlock(&c.lock }, 3)
    return true
}
```
Channel implementation

- You can just read it:
  - [https://golang.org/src/runtime/chan.go](https://golang.org/src/runtime/chan.go)
- Some highlights
Channel implementation

- You can just read it:
  - [https://golang.org/src/runtime/chan.go](https://golang.org/src/runtime/chan.go)
- Some highlights

```go
// Sends and receives on unbuffered or empty-buffered channels are the
// only operations where one running goroutine writes to the stack of
// another running goroutine. The GC assumes that stack writes only
// happen when the goroutine is running and are only done by that
// goroutine. Using a write barrier is sufficient to make up for
// violating that assumption, but the write barrier has to work.
// typedmemmove will call bulkBarrierPreWrite, but the target bytes
// are not in the heap, so that will not help. We arrange to call
// memmove and typeBitsBulkBarrier instead.

func sendDirect(t *type, sg *sudog, src unsafe.Pointer) {
    // src is on our stack, dst is a slot on another stack.

    // Once we read sg.elem out of sg, it will no longer
    // be updated if the destination's stack gets copied (shrunk).
    // So make sure that no preemption points can happen between read & use.
    dst := sg.elem
    typeBitsBulkBarrier(t, uintptr(dst), uintptr(src), t.size)
    memmove(dst, src, t.size)

    // The lock.
    emptyslicePreRemove("send on closed channel")
    if sz == nil {
        // Read a waiting channel - we pass the value we want to send
        // directly to the receiver, bypassing the channel buffer (if any).
    }
}
```
Channel implementation

- You can just read it:
  - [https://golang.org/src/runtime/chan.go](https://golang.org/src/runtime/chan.go)

- Some highlights

```go
295 // Sends and receives on unbuffered or empty-buffered channels are the
296 // only operations where one running goroutine writes to the stack of
297 // another running goroutine. The GC assumes that stack writes only
298 // happen when the goroutine is running and are only done by that
299 // goroutine. Using a write barrier is sufficient to make up for
300 // violating that assumption, but the write barrier has to work.
301 // typedmemcpy will call bulkBarrierPreWrite, but the target bytes
302 // are not in the heap, so that will not help. We arrange to call
303 // memmove and typeBitsBulkBarrier instead.
304
305 func sendDirect(t *type, sg *sudog, src unsafe.Pointer) {
306     // src is on our stack, dst is a slot on another stack.
307
308     // Once we read sg.elem out of sg, it will no longer
309     // be updated if the destination's stack gets copied (shrunk).
310     // So make sure that no preemption points can happen between read & use.
311     dst := sg.elem
typeBitsBulkBarrier(t, uintptr(dst), uintptr(src), t.size)
313     memmove(dst, src, t.size)
314 }
```
Channel implementation

- You can just read it:
  - https://golang.org/src/runtime/chan.go

- Some highlights

```go
func sendDirect(t *type, sg *sudog, src unsafe.Pointer) {
    // src is on our stack, dst is a slot on another stack.
    // Once we read sg.elem out of sg, it will no longer
    // be updated if the destination's stack gets copied (shrunk).
    // So make sure that no preemption points can happen between read & use.
    dst := sg.elem
typeBitsBulkBarrier(t, uintptr(dst), uintptr(src), t.size)
    memmove(dst, src, t.size)
}
```
Channel implementation

• You can just read it:
  • [https://golang.org/src/runtime/chan.go](https://golang.org/src/runtime/chan.go)

• Some highlights

Transputers did this in hardware in the 90s btw.
Channel implementation

• You can just read it:
  • https://golang.org/src/runtime/chan.go

• Some highlights:
  • Race detection built in
  • Fast path just write to receiver stack
  • Often has no capacity → scheduler hint!
  • Buffered channel implementation fairly standard
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 GPU:

- 80 SMs
- Streaming Multiprocessor
- 64 cores/SM
- 5210 threads!
- 15.7 TFLOPS
A modern GPU: Volta V100

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

Roughly: all of k-means 1,000s X/sec
A modern GPU:

- 80 SMs
- Streaming Multiprocessor
- 64 cores/SM
- 5210 threads!
- 15.7 TFLOPS
A modern GPU:

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

- 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

- 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: Volta V100

- 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: Volta V100

- 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: Volta V100

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

- 640 Tensor cores
- HBM2 memory
  - 4096-bit bus
  - No cache coherence!

- 16 GB memory
- PCIe-attached

How do you program a machine like this? pthread_create()?
GPUs: Outline

• Background from many areas
  • Architecture
    • Vector processors
    • Hardware multi-threading
  • Graphics
    • Graphics pipeline
    • Graphics programming models
  • Algorithms
    • parallel architectures → parallel algorithms

• Programming GPUs
  • CUDA
  • Basics: getting something working
  • Advanced: making it perform
Architecture Review: Pipelines

Processor algorithm:

```java
main() {
    while(true)
        do_next_instruction();
}
```
Architecture Review: Pipelines

Processor algorithm:

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

```cpp
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:

```c
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);
}

main() {
    pthread_create(do_instructions);
    pthread_create(do_decode);
    pthread_create(do_execute);
    ... 
    pthread_join(...);
    ... 
}
```
Architecture Review: Pipelines

Processor algorithm:

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

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

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

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

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

```c
main() {
    pthread_create(do_instructions);
    pthread_create(do_decode);
    pthread_create(do_execute);
    ...
    pthread_join(...);
}
```
Architecture Review: Pipelines

Processor algorithm:

```c
main() {
    while(true) {
        do_next_instruction();
    }
}
```
Architecture Review: Pipelines

Processor algorithm:

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

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

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

```cpp
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:

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

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

![Pipeline Diagram]

**Pipeline Stage**

<table>
<thead>
<tr>
<th>Instr No.</th>
<th>Pipeline Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>2</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>3</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>4</td>
<td>IF ID EX MEM</td>
</tr>
<tr>
<td>5</td>
<td>IF ID EX</td>
</tr>
</tbody>
</table>

Clock Cycle: 1 2 3 4 5 6 7
Architecture Review: Pipelines

Processor algorithm:

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

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

What is the name of this kind of parallelism?
Architecture Review: Pipelines

Processor algorithm:

```c
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);
}
```

What is the name of this kind of parallelism?

Works well if pipeline is kept full

What kinds of things cause “bubbles”/stalls?
Architecture Review: Pipelines

Processor algorithm:
```java
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);
}

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?
Processor algorithm:

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

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

What is the name of this kind of parallelism?

What kinds of things cause “bubbles”/stalls?
Architecture Review: Pipelines

Processor algorithm:

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

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

What is the name of this kind of parallelism?

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
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);
    }
}
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);
    }
}
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);
    }
}
Superscalar processors
Superscalar processors

Remove extra instruction streams
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_addr(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?
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?

Enables independent instruction parallelism.
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.
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
Vector/SIMD processors

Why decode same instruction sequence over and over?
Vector/SIMD processors
Vector/SIMD processors

```c
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);
}
Vector/SIMD processors

```c
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 \( \rightarrow \) 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

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

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

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

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

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

GPUs: same basic idea

```
# 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  
    DADDIU 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 $\approx =$ Rendering
Graphics $\sim= $ Rendering

Inputs
Graphics $\sim=$ 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
Graphics \sim 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 surface/light interactions with modeled objects/materials
Graphics \sim 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 surface/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 surface/light interactions with modeled objects/materials

• View point

Output
Graphics \approx \text{Rendering}

**Inputs**

- 3D world model (objects, materials)
  - Geometry modeled with 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 surface/light interactions with modeled objects/materials

- View point

**Output**

- 2D projection seen from the view-point
Graphics \(\sim\) Rendering

**Inputs**

- **3D world model (objects, materials)**
  - Geometry modeled with 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 surface/light interactions with modeled objects/materials

- **View point**

**Output**

- 2D projection seen from the view-point

10/30/2018
Grossly over-simplified rendering algorithm
Grossly over-simplified rendering algorithm

foreach(vertex v in model)
Grossly over-simplified rendering algorithm

```java
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

```java
foreach (vertex v in model)
    map v_{model} \rightarrow v_{view}
fragment[] frags = {};
```
Grossly over-simplified rendering algorithm

```plaintext
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

\[
\begin{align*}
\text{foreach}(\text{vertex } v \text{ in model}) & \\
& \quad \text{map } v_{\text{model}} \rightarrow v_{\text{view}} \\
\text{fragment}[] \text{ frags} = \{} & \\
\text{foreach triangle } t (v_0, v_1, v_2) & \\
& \quad \text{frags.add(rasterize}(t)) ;
\end{align*}
\]
Grossly over-simplified rendering algorithm

```java
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
```

10/30/2018  Dandelion
Grossly over-simplified rendering algorithm

```java
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));

tforeach fragment f in frags
    choose_color(f);
```
Grossly over-simplified rendering algorithm

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

OpenGL pipeline

To first order, DirectX looks the same!
Algorithm ➔ Graphics Pipeline

```plaintext
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

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

GeForce 6 series
Limited “programmability” of shaders:
Minimal/no control flow
Maximum instruction count
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

GeForce 6 series
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
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)
- \( \text{map}(A, f) \rightarrow \) apply \( f(x) \) on all elements in \( A \)
- Parallelism trivially exposed
  - \( f(x) \) can be applied in parallel to all elements, in principle
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

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

• Gather:
  • Read multiple items to single location

• Scatter:
  • Write single data item to multiple locations
Scatter and Gather

• Gather:
  • Read multiple items to single location

• Scatter:
  • Write single data item to multiple locations

\[
\text{for } (i=0; i<N; ++i) \\
x[i] = y[idx[i]];
\]
\[
\text{gather}(x, y, \text{idx})
\]

\[
\text{for } (i=0; i<N; ++i) \\
y[idx[i]] = x[i];
\]
\[
\text{scatter}(x, y, \text{idx})
\]
Reduce

- Input
  - Associative operator \( \text{op} \)
  - Ordered set \( s = [a, b, c, \ldots z] \)

- \( \text{Reduce}(\text{op}, s) \) returns \( a \ \text{op} \ b \ \text{op} \ c \ldots \ \text{op} \ z \)
Reduce

• Input
  • Associative operator \( \text{op} \)
  • Ordered set \( s = [a, b, c, \ldots z] \)
• \( \text{Reduce}(\text{op}, s) \) returns \( a \ \text{op} \ b \ \text{op} \ c \ldots \ \text{op} \ z \)

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

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

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

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

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

- Input
  - Associative operator $\text{op}$
  - Ordered set $s = [a, b, c, \ldots, z]$
  - Identity $I$

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

- 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