

Parallel Architectures

Parallel Algorithms

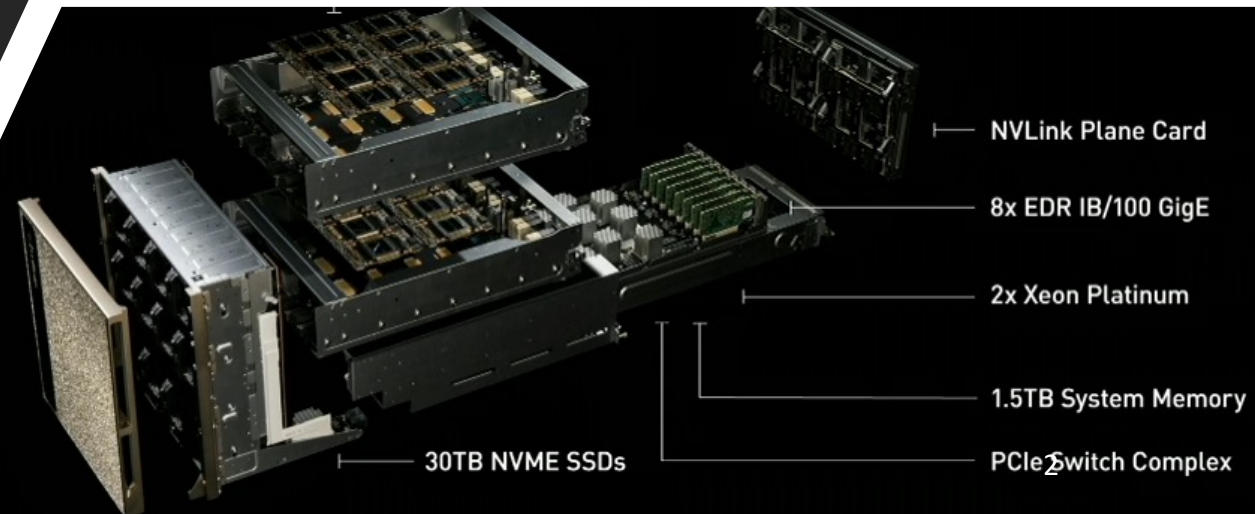
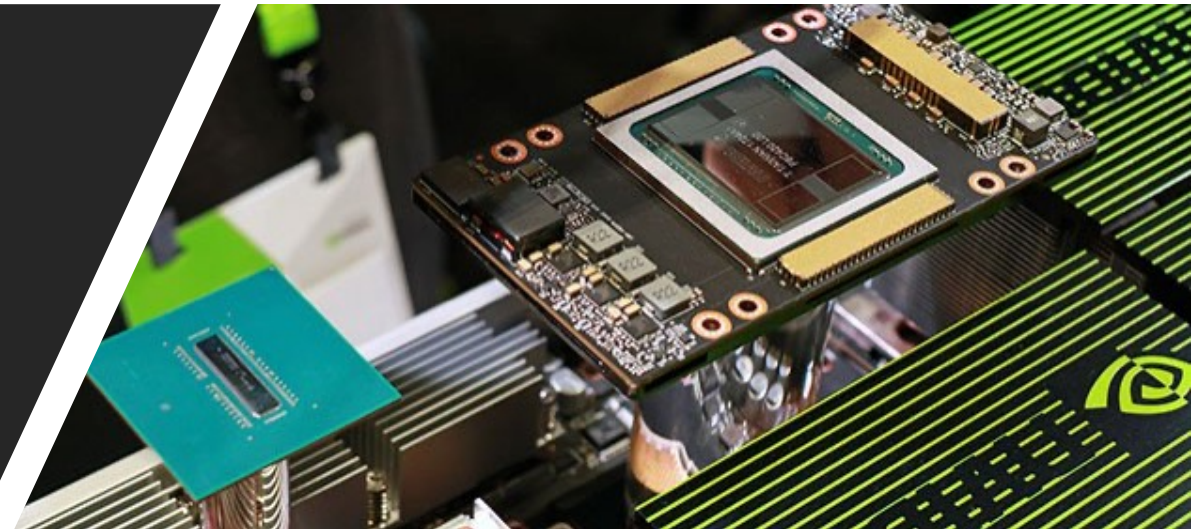
CUDA

Chris Rossbach

cs378h

Outline for Today

- Questions?
- Administrivia
 - pedagogical-* machines should be available
- Agenda
 - Parallel Algorithms
 - CUDA
- Acknowledgements:
http://developer.download.nvidia.com/compute/developertrainingmaterials/presentations/cuda_language/Introduction_to_CUDA_C.pptx



Faux Quiz Questions

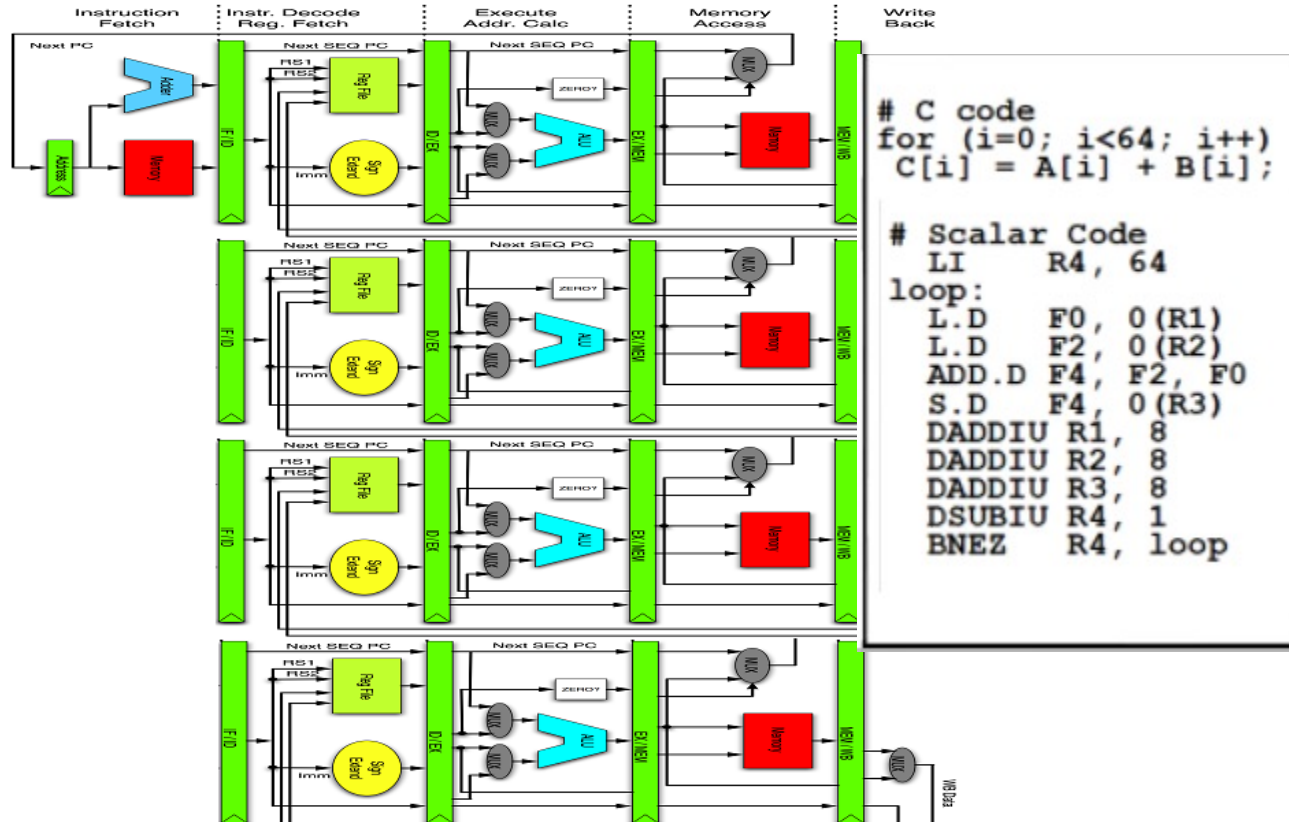
- What is a reduction? A prefix sum? Why are they hard to parallelize and what basic techniques can be used to parallelize them?
- Define flow dependence, output dependence, and anti-dependence: give an example of each. Why/how do compilers use them to detect loop-independent vs loop-carried dependences?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

Review: what is a vector processor?

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

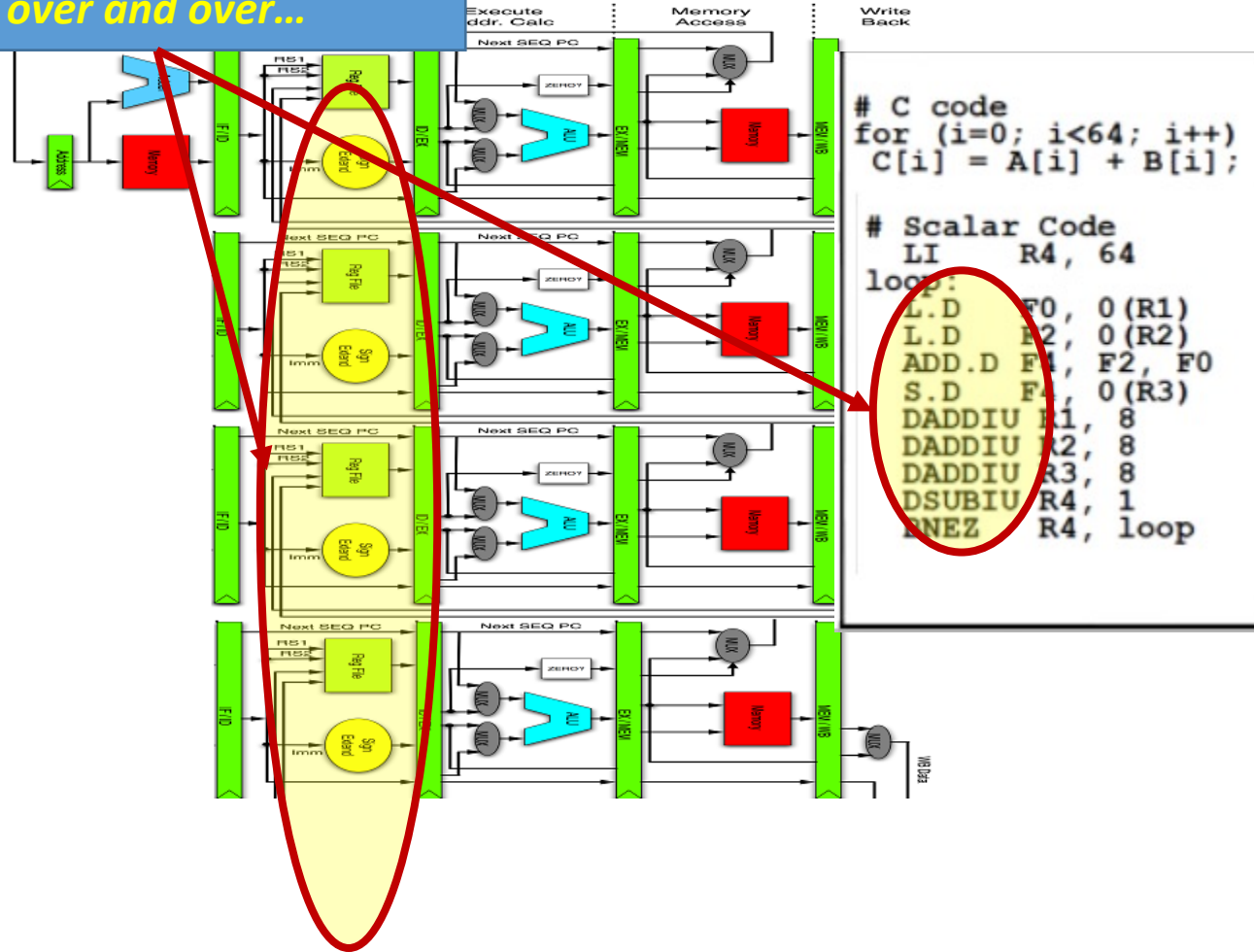
# Scalar Code
LI      R4, 64
loop:
  L.D   F0, 0(R1)
  L.D   F2, 0(R2)
  ADD.D F4, F2, F0
  S.D   F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ  R4, loop
```

Review: what is a vector processor?

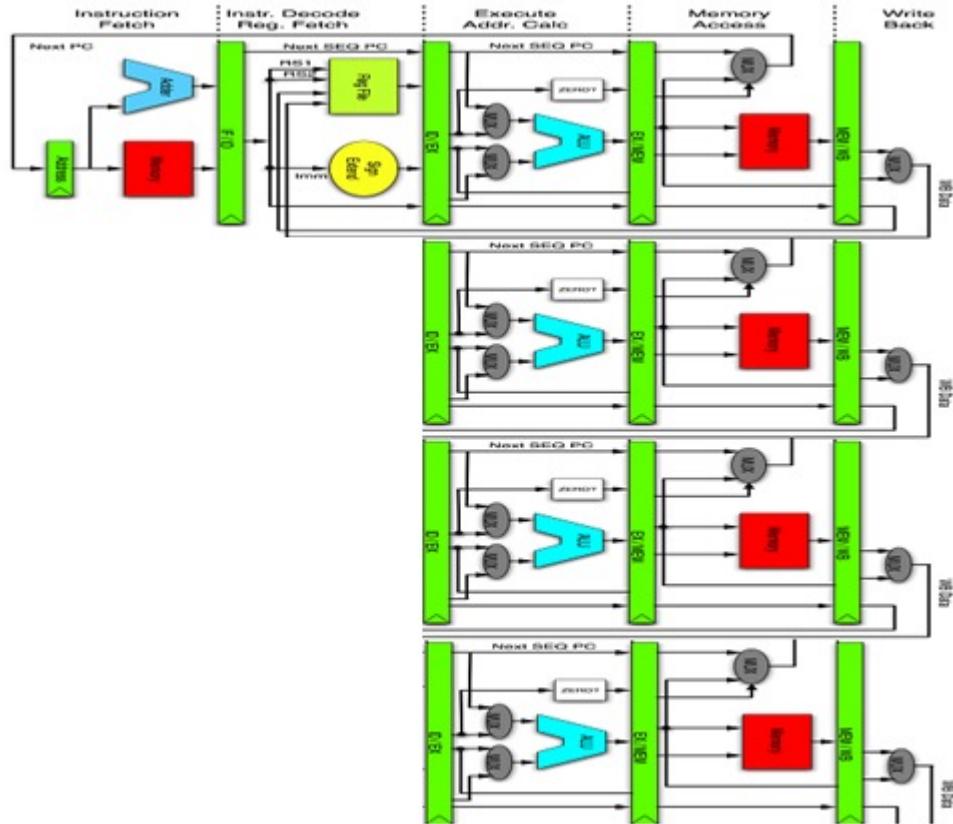


Review: what is a vector processor?

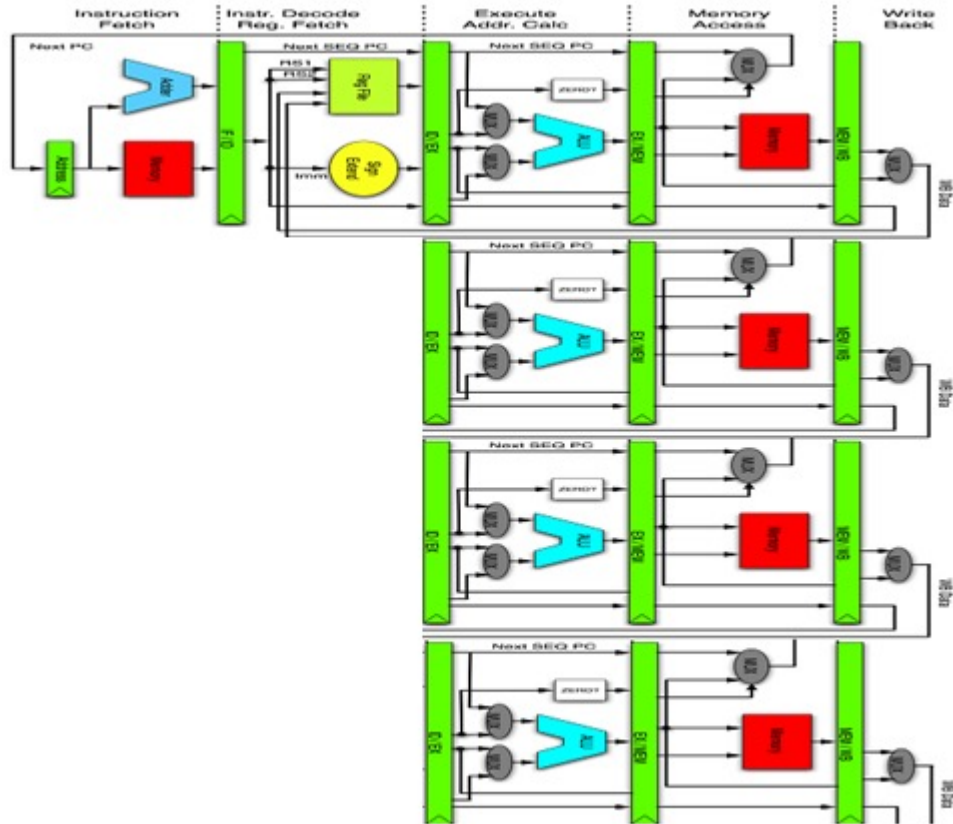
Dont decode same instruction over and over...



Review: what is a vector processor?



Review: what is a vector processor?

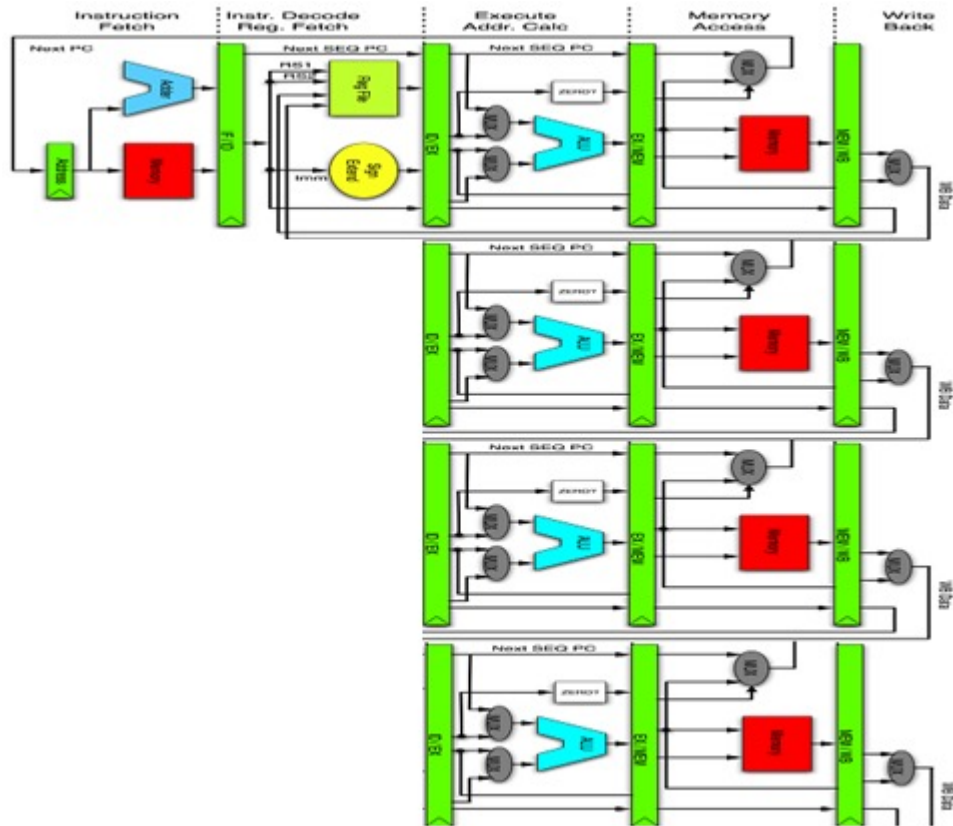


```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

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

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


Review: what is a vector processor?



Implementation:

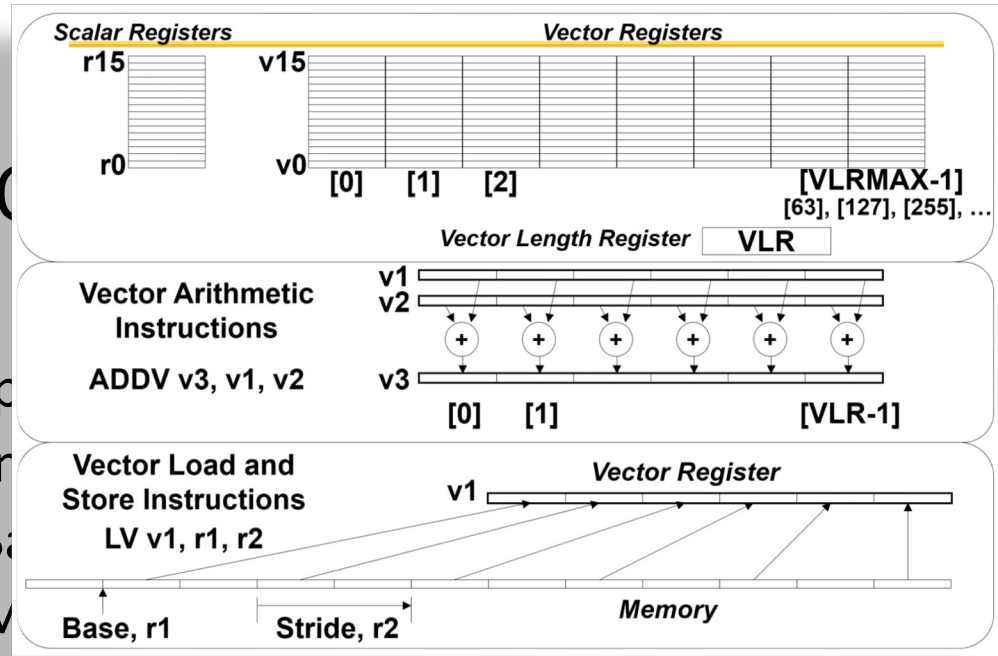
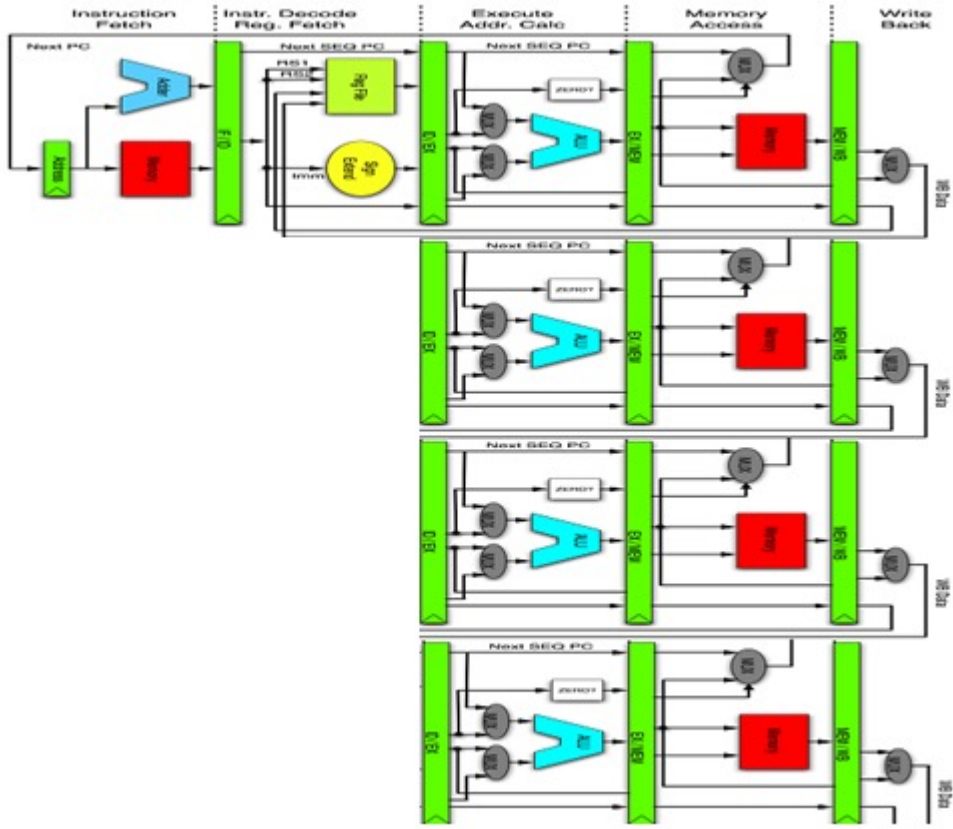
- Instruction fetch control logic shared
- Same instruction stream executed on
- Multiple pipelines
- Multiple different operands in parallel

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

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

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

Review: what is a vector processor



- Imp
- In
 - S
 - M
 - Multiple different operands in parallel

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

Review: Hardware multi-threading

Review: Hardware multi-threading

- Address memory bottleneck

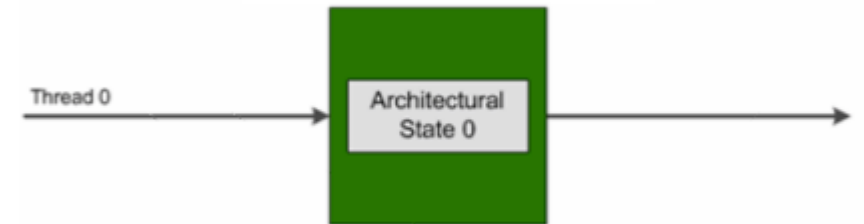
Review: Hardware multi-threading

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

Review: Hardware multi-threading

ZMM0	YMM0	XMM0	ZMM1	YMM1	XMM1	ST(0) MM0	ST(1) MM1	32-bit register RAX	32-bit register RBX	32-bit register RCX	CR0	CR4
ZMM2	YMM2	XMM2	ZMM3	YMM3	XMM3	ST(2) MM2	ST(3) MM3	32-bit register RDX	32-bit register RSP	32-bit register RBP	CR2	CR5
ZMM4	YMM4	XMM4	ZMM5	YMM5	XMM5	ST(4) MM4	ST(5) MM5	32-bit register RSI	32-bit register RDI	32-bit register R10	CR2	CR6
ZMM6	YMM6	XMM6	ZMM7	YMM7	XMM7	ST(6) MM6	ST(7) MM7	32-bit register R11	32-bit register R12	32-bit register R13	CR3	CR7
ZMM8	YMM8	XMM8	ZMM9	YMM9	XMM9			32-bit register R14	32-bit register R15	32-bit register R16	CR3	CR8
ZMM10	YMM10	XMM10	ZMM11	YMM11	XMM11	CW	FP_IP FP_DP FP_CS	16-bit register R17	16-bit register R18	16-bit register R19	MSW	CR9
ZMM12	YMM12	XMM12	ZMM13	YMM13	XMM13	SW		16-bit register R20	16-bit register R21	16-bit register R22		CR10
ZMM14	YMM14	XMM14	ZMM15	YMM15	XMM15	TW		16-bit register R23	16-bit register R24	16-bit register R25		CR11
ZMM16	YMM16	XMM16	ZMM17	YMM17	XMM17	FP_DS		16-bit register R26	16-bit register R27	16-bit register R28		CR12
ZMM18	YMM18	XMM18	ZMM19	YMM19	XMM19	FP_ORC FP_DP FP_IP		16-bit register R29	16-bit register R30	16-bit register R31		CR13
						CS	SS	DS	GS	TR	DR0	DR6
						ES	FS	GS		LOTR	DR2	DR7
											DR3	DR8
											DR4	DR9
											DR4	DR10
											DR5	DR11
											DR12	DR14
											DR13	DR15

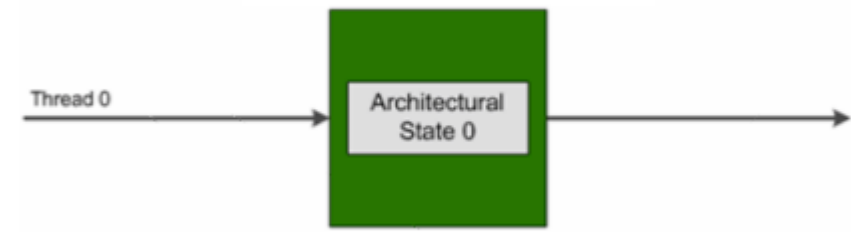
- Address memory bottleneck
- Share exec unit across
 - Instruction streams
 - Switch on stalls



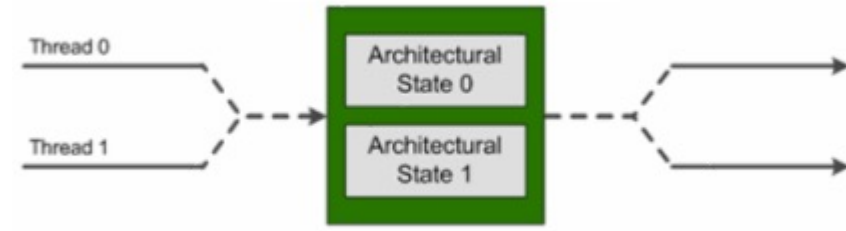
Review: Hardware multi-threading

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

ZMM0	YMM0	XMM0	ZMM1	YMM1	XMM1	ST(0) MM0	ST(1) MM1	MMX	MMX	RAX	RCX	RCX	R32	CR0	CR4	
ZMM2	YMM2	XMM2	ZMM3	YMM3	XMM3	ST(2) MM2	ST(3) MM3	MMX	MMX	RDX	RDX	R8	R13	CR1	CR5	
ZMM4	YMM4	XMM4	ZMM5	YMM5	XMM5	ST(4) MM4	ST(5) MM5	MMX	MMX	RDX	RDX	R10	R14	CR2	CR6	
ZMM6	YMM6	XMM6	ZMM7	YMM7	XMM7	ST(6) MM6	ST(7) MM7	MMX	MMX	RDX	RDX	R11	R15	CR3	CR7	
ZMM8	YMM8	XMM8	ZMM9	YMM9	XMM9			MMX	MMX	RDX	RDX	RIP	RIP	CR3	CR8	
ZMM10	YMM10	XMM10	ZMM11	YMM11	XMM11			CW	FP_IP	FP_DP	FP_CS	CS	SS	DS	MSW	CR9
ZMM12	YMM12	XMM12	ZMM13	YMM13	XMM13			SW								CR10
ZMM14	YMM14	XMM14	ZMM15	YMM15	XMM15			TW								CR11
ZMM16	YMM16	XMM16	ZMM17	YMM17	XMM17			FP_DS								CR12
ZMM18	YMM18	XMM18	ZMM19	YMM19	XMM19			FP_OP	FP_DP	FP_IP						CR13
ZMM20	YMM20	XMM20	ZMM21	YMM21	XMM21			ES	FS	OS	GDTR	IDTR	DR0	DR6	CR14	
ZMM22	YMM22	XMM22	ZMM23	YMM23	XMM23						TR	LDTR	CR2	DR7	CR15	
ZMM24	YMM24	XMM24	ZMM25	YMM25	XMM25								CR2	DR8	CR15	
ZMM26	YMM26	XMM26	ZMM27	YMM27	XMM27								CR3	DR9	CR15	
ZMM28	YMM28	XMM28	ZMM29	YMM29	XMM29								DR4	DR10	DR12	DR14
ZMM30	YMM30	XMM30	ZMM31	YMM31	XMM31								DR5	DR11	DR13	DR15



ZMM0	YMM0	XMM0	ZMM1	YMM1	XMM1	ST(0) MM0	ST(1) MM1	MMX	MMX	RAX	RCX	RCX	R32	CR0	CR4	
ZMM2	YMM2	XMM2	ZMM3	YMM3	XMM3	ST(2) MM2	ST(3) MM3	MMX	MMX	RDX	RDX	R8	R13	CR1	CR5	
ZMM4	YMM4	XMM4	ZMM5	YMM5	XMM5	ST(4) MM4	ST(5) MM5	MMX	MMX	RDX	RDX	R10	R14	CR2	CR6	
ZMM6	YMM6	XMM6	ZMM7	YMM7	XMM7	ST(6) MM6	ST(7) MM7	MMX	MMX	RDX	RDX	R11	R15	CR3	CR7	
ZMM8	YMM8	XMM8	ZMM9	YMM9	XMM9			MMX	MMX	RDX	RDX	RIP	RIP	CR3	CR8	
ZMM10	YMM10	XMM10	ZMM11	YMM11	XMM11			CW	FP_IP	FP_DP	FP_CS	CS	SS	DS	MSW	CR9
ZMM12	YMM12	XMM12	ZMM13	YMM13	XMM13			SW								CR10
ZMM14	YMM14	XMM14	ZMM15	YMM15	XMM15			TW								CR11
ZMM16	YMM16	XMM16	ZMM17	YMM17	XMM17			FP_DS								CR12
ZMM18	YMM18	XMM18	ZMM19	YMM19	XMM19			FP_OP	FP_DP	FP_IP						CR13
ZMM20	YMM20	XMM20	ZMM21	YMM21	XMM21			ES	FS	OS	GDTR	IDTR	DR0	DR6	CR14	
ZMM22	YMM22	XMM22	ZMM23	YMM23	XMM23						TR	LDTR	CR2	DR7	CR15	
ZMM24	YMM24	XMM24	ZMM25	YMM25	XMM25								CR2	DR8	CR15	
ZMM26	YMM26	XMM26	ZMM27	YMM27	XMM27								CR3	DR9	CR15	
ZMM28	YMM28	XMM28	ZMM29	YMM29	XMM29								DR4	DR10	DR12	DR14
ZMM30	YMM30	XMM30	ZMM31	YMM31	XMM31								DR5	DR11	DR13	DR15

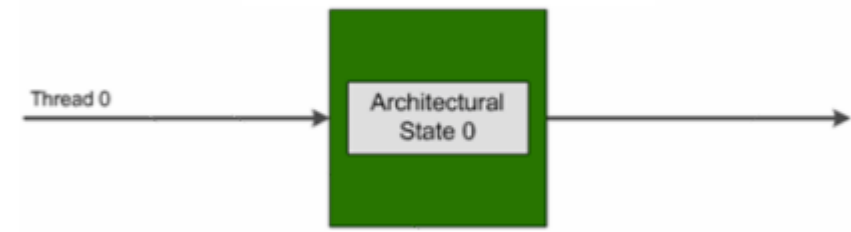


ZMM0	YMM0	XMM0	ZMM1	YMM1	XMM1	ST(0) MM0	ST(1) MM1	MMX	MMX	RAX	RCX	RCX	R32	CR0	CR4	
ZMM2	YMM2	XMM2	ZMM3	YMM3	XMM3	ST(2) MM2	ST(3) MM3	MMX	MMX	RDX	RDX	R8	R13	CR1	CR5	
ZMM4	YMM4	XMM4	ZMM5	YMM5	XMM5	ST(4) MM4	ST(5) MM5	MMX	MMX	RDX	RDX	R10	R14	CR2	CR6	
ZMM6	YMM6	XMM6	ZMM7	YMM7	XMM7	ST(6) MM6	ST(7) MM7	MMX	MMX	RDX	RDX	R11	R15	CR3	CR7	
ZMM8	YMM8	XMM8	ZMM9	YMM9	XMM9			MMX	MMX	RDX	RDX	RIP	RIP	CR3	CR8	
ZMM10	YMM10	XMM10	ZMM11	YMM11	XMM11			CW	FP_IP	FP_DP	FP_CS	CS	SS	DS	MSW	CR9
ZMM12	YMM12	XMM12	ZMM13	YMM13	XMM13			SW								CR10
ZMM14	YMM14	XMM14	ZMM15	YMM15	XMM15			TW								CR11
ZMM16	YMM16	XMM16	ZMM17	YMM17	XMM17			FP_DS								CR12
ZMM18	YMM18	XMM18	ZMM19	YMM19	XMM19			FP_OP	FP_DP	FP_IP						CR13
ZMM20	YMM20	XMM20	ZMM21	YMM21	XMM21			ES	FS	OS	GDTR	IDTR	DR0	DR6	CR14	
ZMM22	YMM22	XMM22	ZMM23	YMM23	XMM23						TR	LDTR	CR2	DR7	CR15	
ZMM24	YMM24	XMM24	ZMM25	YMM25	XMM25								CR2	DR8	CR15	
ZMM26	YMM26	XMM26	ZMM27	YMM27	XMM27								CR3	DR9	CR15	
ZMM28	YMM28	XMM28	ZMM29	YMM29	XMM29								DR4	DR10	DR12	DR14
ZMM30	YMM30	XMM30	ZMM31	YMM31	XMM31								DR5	DR11	DR13	DR15

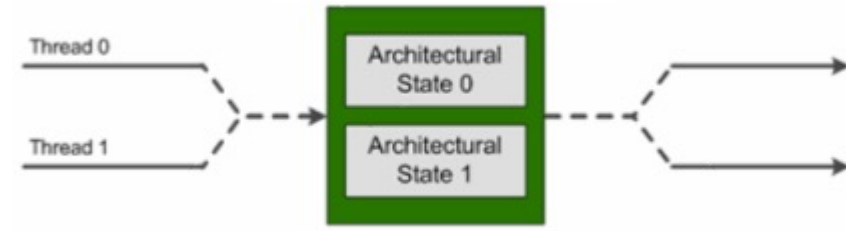
Review: Hardware multi-threading

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

ZMM0	YMM0	XMM0	ZMM1	YMM1	XMM1	ST(0) MM0	ST(1) MM1	MMX	MMX	RAX	CR0	CR4
ZMM2	YMM2	XMM2	ZMM3	YMM3	XMM3	ST(2) MM2	ST(3) MM3	CR2	CR3	R10	CR1	CR5
ZMM4	YMM4	XMM4	ZMM5	YMM5	XMM5	ST(4) MM4	ST(5) MM5	CR4	CR5	R11	CR2	CR6
ZMM6	YMM6	XMM6	ZMM7	YMM7	XMM7	ST(6) MM6	ST(7) MM7	CR6	CR7	R12	CR3	CR7
ZMM8	YMM8	XMM8	ZMM9	YMM9	XMM9			CR7	CR8	R13	CR4	CR8
ZMM10	YMM10	XMM10	ZMM11	YMM11	XMM11			CR8	CR9	R14	CR5	CR9
ZMM12	YMM12	XMM12	ZMM13	YMM13	XMM13			CR9	CR10	R15	CR6	CR10
ZMM14	YMM14	XMM14	ZMM15	YMM15	XMM15			CR10	CR11	CR15	CR11	CR12
ZMM16	YMM16	XMM16	ZMM17	YMM17	XMM17			CR11	CR12		CR12	CR13
ZMM18	YMM18	XMM18	ZMM19	YMM19	XMM19			CR12	CR13		CR13	CR14
ZMM20	YMM20	XMM20	ZMM21	YMM21	XMM21			CR13	CR14		CR14	CR15
ZMM22	YMM22	XMM22	ZMM23	YMM23	XMM23			CR14	CR15		CR15	MXCSR
ZMM24	YMM24	XMM24	ZMM25	YMM25	XMM25			CR15	MXCSR			



ZMM0	YMM0	XMM0	ZMM1	YMM1	XMM1	ST(0) MM0	ST(1) MM1	MMX	MMX	RAX	CR0	CR4
ZMM2	YMM2	XMM2	ZMM3	YMM3	XMM3	ST(2) MM2	ST(3) MM3	CR2	CR3	R10	CR1	CR5
ZMM4	YMM4	XMM4	ZMM5	YMM5	XMM5	ST(4) MM4	ST(5) MM5	CR4	CR5	R11	CR2	CR6
ZMM6	YMM6	XMM6	ZMM7	YMM7	XMM7	ST(6) MM6	ST(7) MM7	CR6	CR7	R12	CR3	CR7
ZMM8	YMM8	XMM8	ZMM9	YMM9	XMM9			CR7	CR8	R13	CR4	CR8
ZMM10	YMM10	XMM10	ZMM11	YMM11	XMM11			CR8	CR9	R14	CR5	CR9
ZMM12	YMM12	XMM12	ZMM13	YMM13	XMM13			CR9	CR10	R15	CR6	CR10
ZMM14	YMM14	XMM14	ZMM15	YMM15	XMM15			CR10	CR11	CR15	CR11	CR12
ZMM16	YMM16	XMM16	ZMM17	YMM17	XMM17			CR11	CR12		CR12	CR13
ZMM18	YMM18	XMM18	ZMM19	YMM19	XMM19			CR12	CR13		CR13	CR14
ZMM20	YMM20	XMM20	ZMM21	YMM21	XMM21			CR13	CR14		CR14	CR15
ZMM22	YMM22	XMM22	ZMM23	YMM23	XMM23			CR14	CR15		CR15	MXCSR
ZMM24	YMM24	XMM24	ZMM25	YMM25	XMM25			CR15	MXCSR			

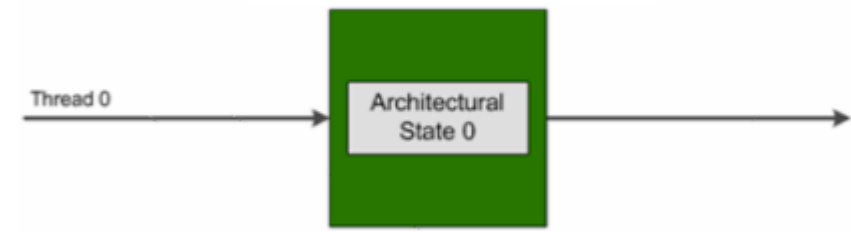


ZMM0	YMM0	XMM0	ZMM1	YMM1	XMM1	ST(0) MM0	ST(1) MM1	MMX	MMX	RAX	CR0	CR4
ZMM2	YMM2	XMM2	ZMM3	YMM3	XMM3	ST(2) MM2	ST(3) MM3	CR2	CR3	R10	CR1	CR5
ZMM4	YMM4	XMM4	ZMM5	YMM5	XMM5	ST(4) MM4	ST(5) MM5	CR4	CR5	R11	CR2	CR6
ZMM6	YMM6	XMM6	ZMM7	YMM7	XMM7	ST(6) MM6	ST(7) MM7	CR6	CR7	R12	CR3	CR7
ZMM8	YMM8	XMM8	ZMM9	YMM9	XMM9			CR7	CR8	R13	CR4	CR8
ZMM10	YMM10	XMM10	ZMM11	YMM11	XMM11			CR8	CR9	R14	CR5	CR9
ZMM12	YMM12	XMM12	ZMM13	YMM13	XMM13			CR9	CR10	R15	CR6	CR10
ZMM14	YMM14	XMM14	ZMM15	YMM15	XMM15			CR10	CR11	CR15	CR11	CR12
ZMM16	YMM16	XMM16	ZMM17	YMM17	XMM17			CR11	CR12		CR12	CR13
ZMM18	YMM18	XMM18	ZMM19	YMM19	XMM19			CR12	CR13		CR13	CR14
ZMM20	YMM20	XMM20	ZMM21	YMM21	XMM21			CR13	CR14		CR14	CR15
ZMM22	YMM22	XMM22	ZMM23	YMM23	XMM23			CR14	CR15		CR15	MXCSR
ZMM24	YMM24	XMM24	ZMM25	YMM25	XMM25			CR15	MXCSR			

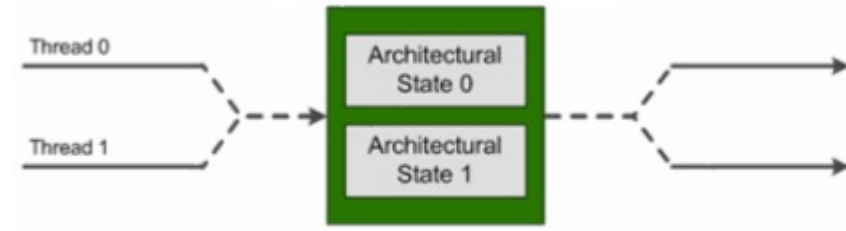
Review: Hardware multi-threading

- 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

ZMM0	YMM0	XMM0	ZMM1	YMM1	XMM1	ST(0) MM0	ST(1) MM1	MMX	MMX	RAX	CR0	CR4
ZMM2	YMM2	XMM2	ZMM3	YMM3	XMM3	ST(2) MM2	ST(3) MM3	CR2	CR3	R10	CR1	CR5
ZMM4	YMM4	XMM4	ZMM5	YMM5	XMM5	ST(4) MM4	ST(5) MM5	CR4	CR5	R11	CR2	CR6
ZMM6	YMM6	XMM6	ZMM7	YMM7	XMM7	ST(6) MM6	ST(7) MM7	CR6	CR7	R12	CR3	CR7
ZMM8	YMM8	XMM8	ZMM9	YMM9	XMM9			CR7	CR8	R13	CR4	CR8
ZMM10	YMM10	XMM10	ZMM11	YMM11	XMM11			CR8	CR9	R14	CR5	CR9
ZMM12	YMM12	XMM12	ZMM13	YMM13	XMM13			CR9	CR10	R15	CR6	CR10
ZMM14	YMM14	XMM14	ZMM15	YMM15	XMM15			CR10	CR11		CR7	CR11
ZMM16	YMM16	XMM16	ZMM17	YMM17	XMM17			CR11	CR12		CR8	CR12
ZMM18	YMM18	XMM18	ZMM19	YMM19	XMM19			CR12	CR13		CR9	CR13
ZMM20	YMM20	XMM20	ZMM21	YMM21	XMM21			CR13	CR14		CR10	CR14
ZMM22	YMM22	XMM22	ZMM23	YMM23	XMM23			CR14	CR15		CR11	CR15
ZMM24	YMM24	XMM24	ZMM25	YMM25	XMM25			CR15	MXCSR		CR12	



ZMM0	YMM0	XMM0	ZMM1	YMM1	XMM1	ST(0) MM0	ST(1) MM1	MMX	MMX	RAX	CR0	CR4
ZMM2	YMM2	XMM2	ZMM3	YMM3	XMM3	ST(2) MM2	ST(3) MM3	CR2	CR3	R10	CR1	CR5
ZMM4	YMM4	XMM4	ZMM5	YMM5	XMM5	ST(4) MM4	ST(5) MM5	CR4	CR5	R11	CR2	CR6
ZMM6	YMM6	XMM6	ZMM7	YMM7	XMM7	ST(6) MM6	ST(7) MM7	CR6	CR7	R12	CR3	CR7
ZMM8	YMM8	XMM8	ZMM9	YMM9	XMM9			CR7	CR8	R13	CR4	CR8
ZMM10	YMM10	XMM10	ZMM11	YMM11	XMM11			CR8	CR9	R14	CR5	CR9
ZMM12	YMM12	XMM12	ZMM13	YMM13	XMM13			CR9	CR10	R15	CR6	CR10
ZMM14	YMM14	XMM14	ZMM15	YMM15	XMM15			CR10	CR11		CR7	CR11
ZMM16	YMM16	XMM16	ZMM17	YMM17	XMM17			CR11	CR12		CR8	CR12
ZMM18	YMM18	XMM18	ZMM19	YMM19	XMM19			CR12	CR13		CR9	CR13
ZMM20	YMM20	XMM20	ZMM21	YMM21	XMM21			CR13	CR14		CR10	CR14
ZMM22	YMM22	XMM22	ZMM23	YMM23	XMM23			CR14	CR15		CR11	CR15
ZMM24	YMM24	XMM24	ZMM25	YMM25	XMM25			CR15	MXCSR		CR12	



ZMM0	YMM0	XMM0	ZMM1	YMM1	XMM1	ST(0) MM0	ST(1) MM1	MMX	MMX	RAX	CR0	CR4
ZMM2	YMM2	XMM2	ZMM3	YMM3	XMM3	ST(2) MM2	ST(3) MM3	CR2	CR3	R10	CR1	CR5
ZMM4	YMM4	XMM4	ZMM5	YMM5	XMM5	ST(4) MM4	ST(5) MM5	CR4	CR5	R11	CR2	CR6
ZMM6	YMM6	XMM6	ZMM7	YMM7	XMM7	ST(6) MM6	ST(7) MM7	CR6	CR7	R12	CR3	CR7
ZMM8	YMM8	XMM8	ZMM9	YMM9	XMM9			CR7	CR8	R13	CR4	CR8
ZMM10	YMM10	XMM10	ZMM11	YMM11	XMM11			CR8	CR9	R14	CR5	CR9
ZMM12	YMM12	XMM12	ZMM13	YMM13	XMM13			CR9	CR10	R15	CR6	CR10
ZMM14	YMM14	XMM14	ZMM15	YMM15	XMM15			CR10	CR11		CR7	CR11
ZMM16	YMM16	XMM16	ZMM17	YMM17	XMM17			CR11	CR12		CR8	CR12
ZMM18	YMM18	XMM18	ZMM19	YMM19	XMM19			CR12	CR13		CR9	CR13
ZMM20	YMM20	XMM20	ZMM21	YMM21	XMM21			CR13	CR14		CR10	CR14
ZMM22	YMM22	XMM22	ZMM23	YMM23	XMM23			CR14	CR15		CR11	CR15
ZMM24	YMM24	XMM24	ZMM25	YMM25	XMM25			CR15	MXCSR		CR12	

Programming Model

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

Programming Model

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

***Need codes that are 1000s-X
parallel...***

Parallel Algorithms

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

Parallel Algorithms

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

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

Map

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

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



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

Scatter and Gather

Scatter and Gather

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

Scatter and Gather

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

Scatter and Gather

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

Scatter and Gather

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

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

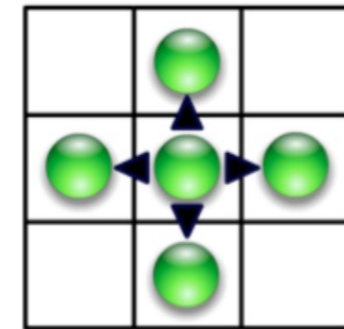
```
for (i=0; i<N; ++i)  
y[idx[i]] = x[i]; → scatter(x, y, idx)
```

Scatter and Gather

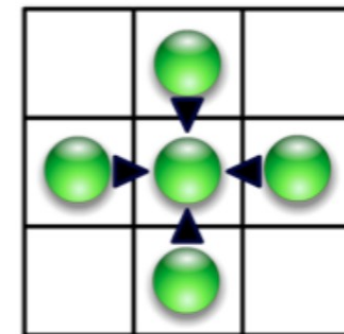
- Gather:
 - Read multiple items to single /packed location
- Scatter:
 - Write single/packed data item to multiple locations
- Inputs: x , y , indices, N

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

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



Scatter



Gather

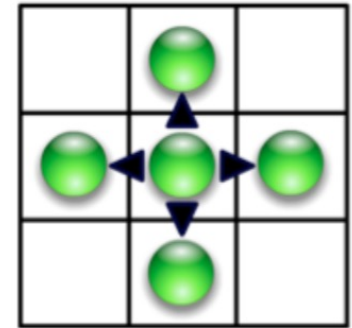
Scatter and Gather

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

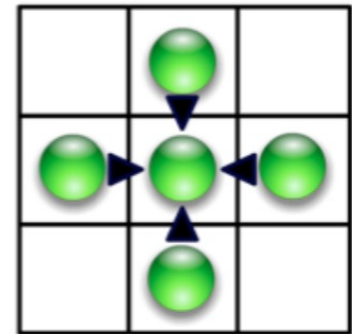
```
for (i=0; i<N; ++i)  
x[i] = y[idx[i]];      →      gather(x, y, idx)
```

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

Why is this useful on a box-drawing machine?



Scatter



Gather

Reduce

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

Reduce

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

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



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


Reduce

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

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

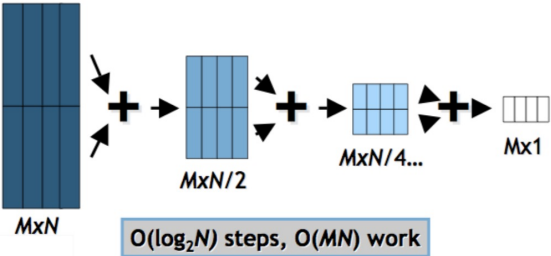
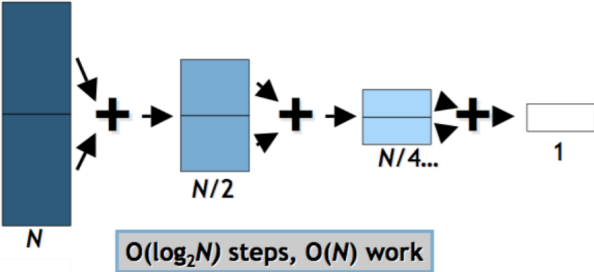


```
accum = reduce(*, point)
```

Why must op be associative?

Reduce

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



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

→ accum = reduce(*, point)

Why must op be associative?

Scan (prefix sum)

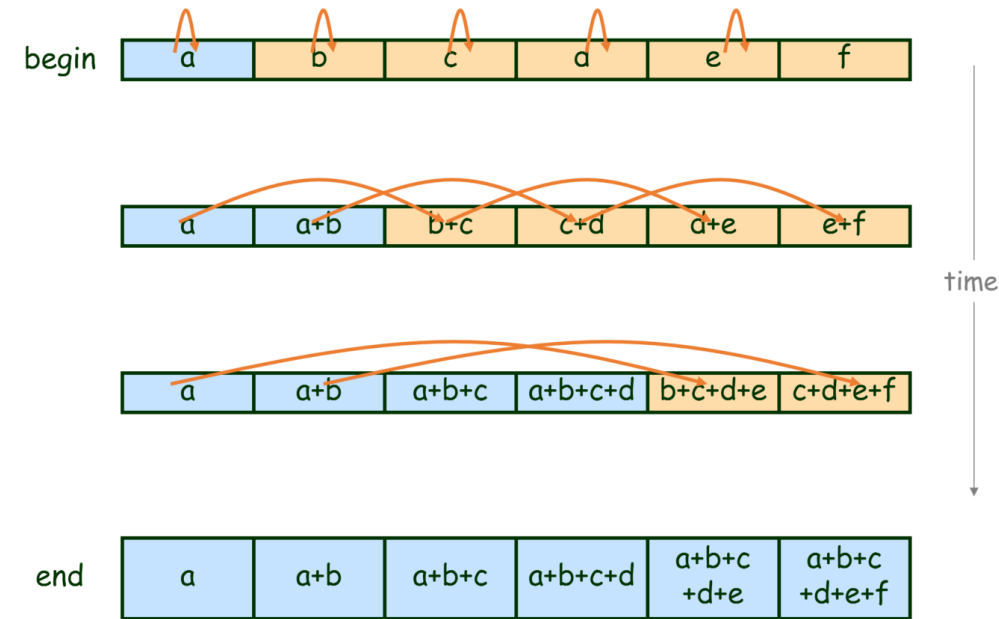
- Input

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

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

- Scan is the workhorse of parallel algorithms:

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



Reduce

Reduce

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

Reduce

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

Reduce

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

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

Reduce

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

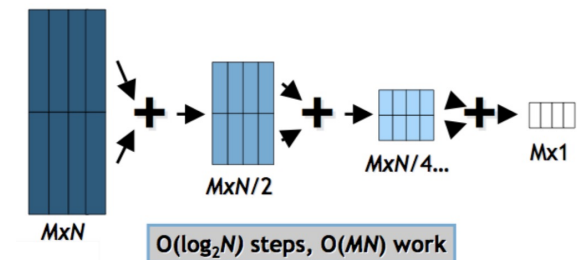
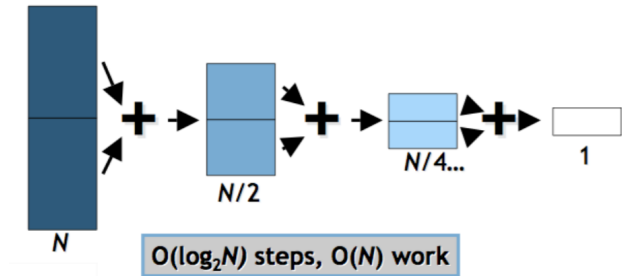
```
for(i=0; i<N; ++i) {  
    accum += point[i]  
}
```



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


Reduce

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



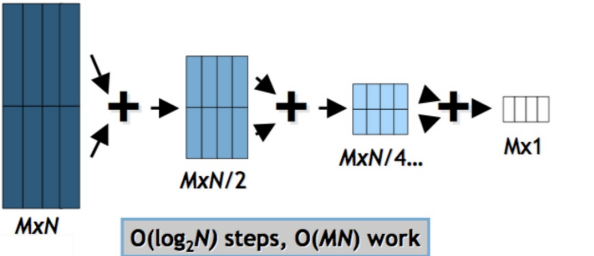
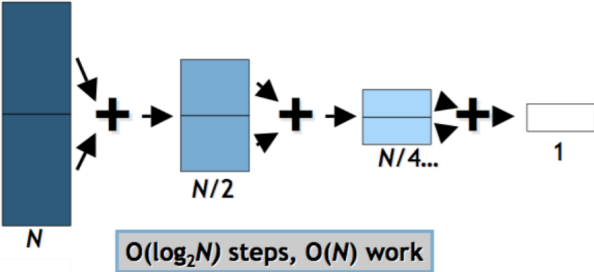
```
for(i=0; i<N; ++i) {  
    accum += point[i]  
}
```



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

Reduce

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



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

→ accum = reduce(+, point)

Why must op be associative?

Scan (prefix sum)

Scan (prefix sum)

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

Scan (prefix sum)

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

Scan (prefix sum)

- Input
 - Associative operator **op**
 - Ordered set $s = [a, b, c, \dots z]$
 - Identity I
- $\text{scan}(\text{op}, s) = [I, a, (a \text{ op } b), (a \text{ op } b \text{ op } c) \dots]$
- Scan is the workhorse of parallel algorithms:
 - Sort, histograms, sparse matrix, string compare, ...

Scan (prefix sum)

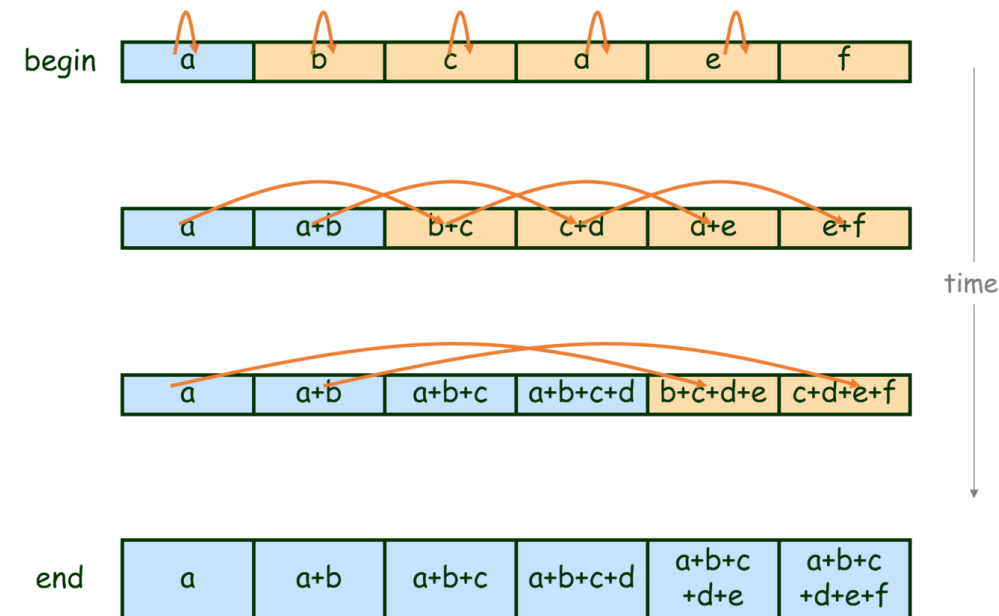
- Input

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

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

- Scan is the workhorse of parallel algorithms:

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



Example: Parallel GroupBy

- Group a collection by key
- Lambda function maps elements \rightarrow key

Example: Parallel GroupBy

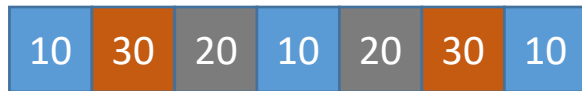
- Group a collection by key
- Lambda function maps elements \rightarrow key

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

Example: Parallel GroupBy

- Group a collection by key
- Lambda function maps elements \rightarrow key

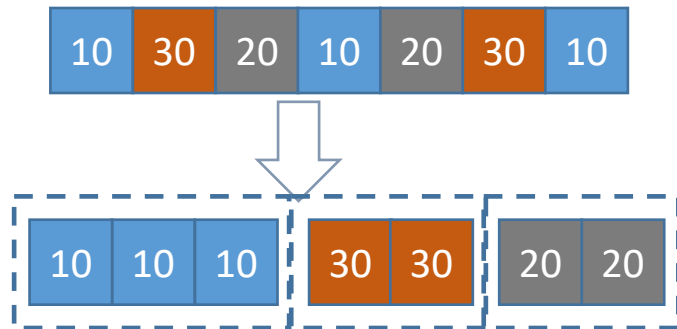
```
var res = ints.GroupBy(x => x);
```



Example: Parallel GroupBy

- Group a collection by key
- Lambda function maps elements \rightarrow key

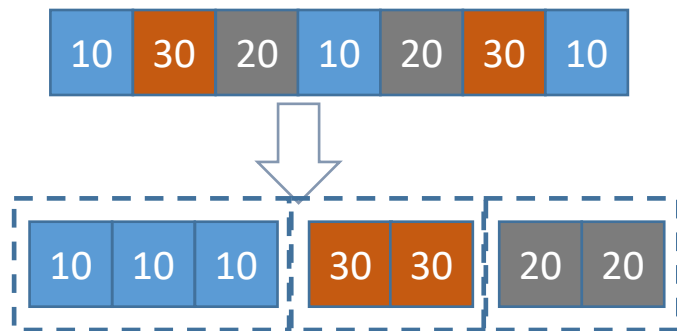
```
var res = ints.GroupBy(x => x) ;
```





Example: Parallel GroupBy

- Group a collection by key
- Lambda function maps elements \rightarrow key

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



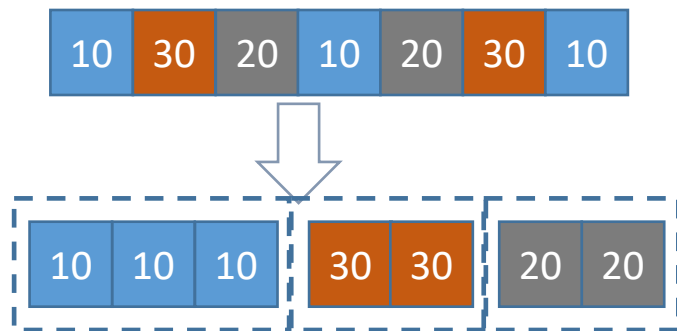
```
foreach(T elem in PF(ints))  
{  
    key    = KeyLambda(elem);  
    group = GetGroup(key)   
    group.Add(elem);   
}
```

Example: Parallel GroupBy

- Group a collection by key
- Lambda function maps elements \rightarrow key

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

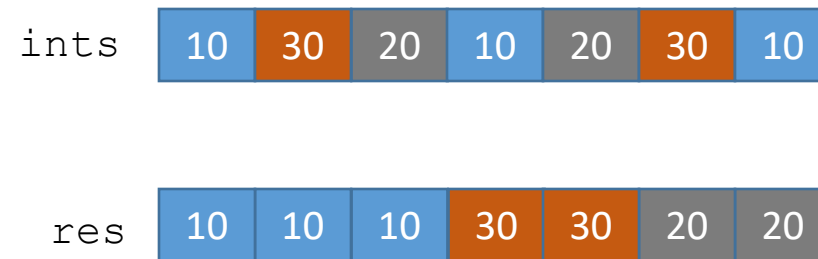
- *Insufficient Parallelism*
- *Requires synchronization*



```
foreach (T key in keys) {  
    group = GetGroup(key);  
    group.Add(elem);  
}
```

The code block is overlaid with a large blue 'no' symbol (a circle with a diagonal slash) and a padlock icon, indicating that the code is not parallel and requires synchronization.

Parallel GroupBy



Parallel GroupBy

Process each input element in parallel

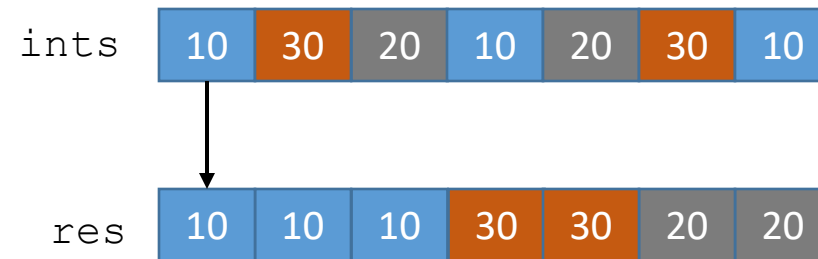
- grouping ~ shuffling
- input item → output offset such that groups are contiguous



Parallel GroupBy

Process each input element in parallel

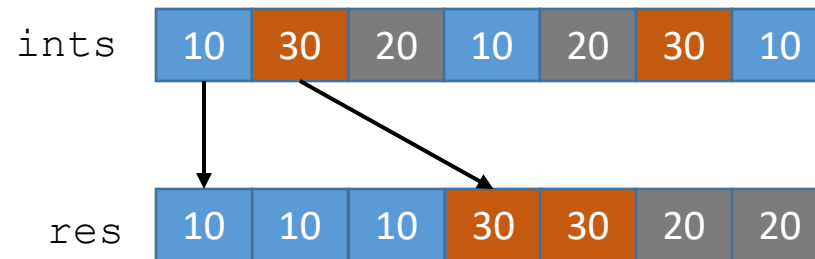
- grouping ~ shuffling
- input item \rightarrow output offset such that groups are contiguous



Parallel GroupBy

Process each input element in parallel

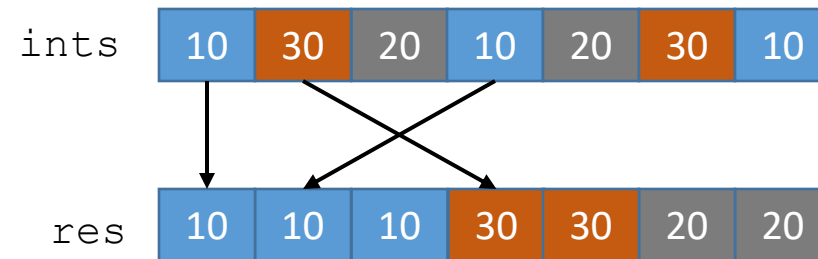
- grouping ~ shuffling
- input item \rightarrow output offset such that groups are contiguous



Parallel GroupBy

Process each input element in parallel

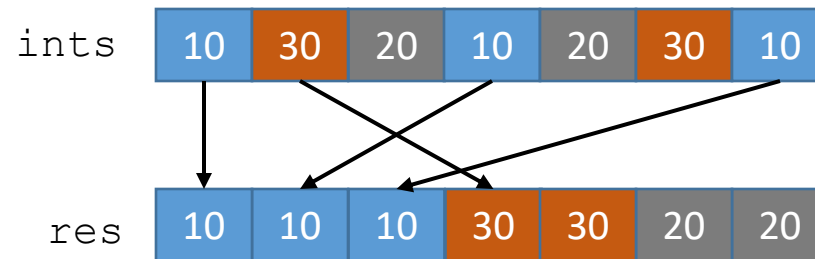
- grouping ~ shuffling
- input item \rightarrow output offset such that groups are contiguous



Parallel GroupBy

Process each input element in parallel

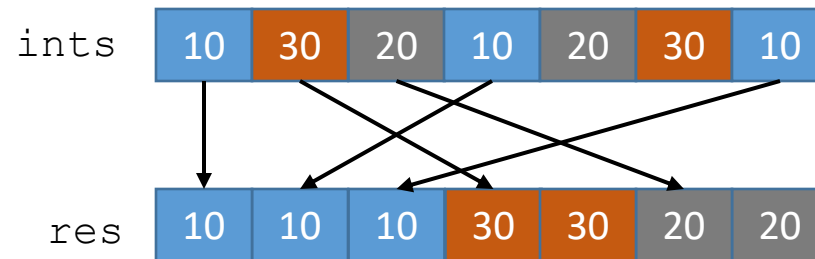
- grouping ~ shuffling
- input item \rightarrow output offset such that groups are contiguous



Parallel GroupBy

Process each input element in parallel

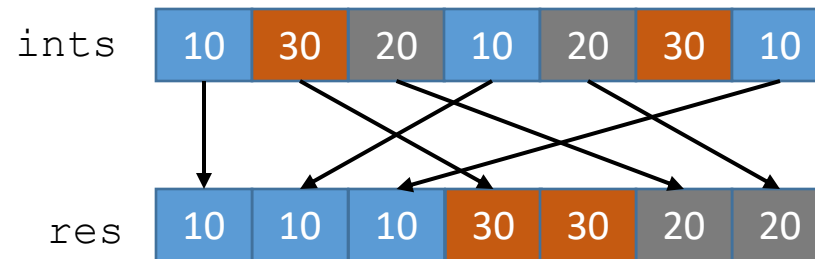
- grouping ~ shuffling
- input item \rightarrow output offset such that groups are contiguous



Parallel GroupBy

Process each input element in parallel

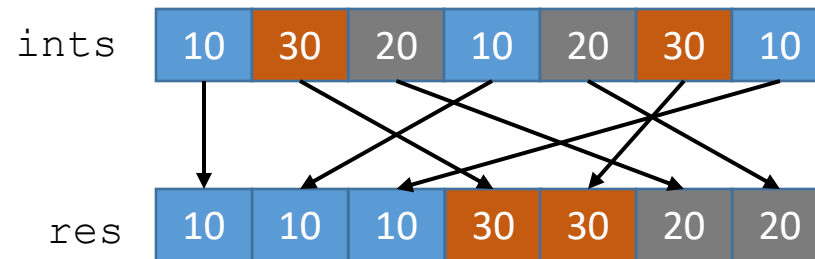
- grouping ~ shuffling
- input item \rightarrow output offset such that groups are contiguous



Parallel GroupBy

Process each input element in parallel

- grouping ~ shuffling
- input item \rightarrow output offset such that groups are contiguous



Parallel GroupBy

Process each input element in parallel

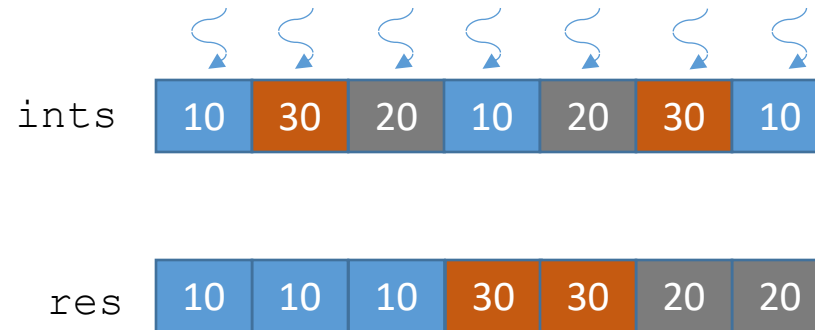
- grouping ~ shuffling
- input item → output offset such that groups are contiguous



Parallel GroupBy

Process each input element in parallel

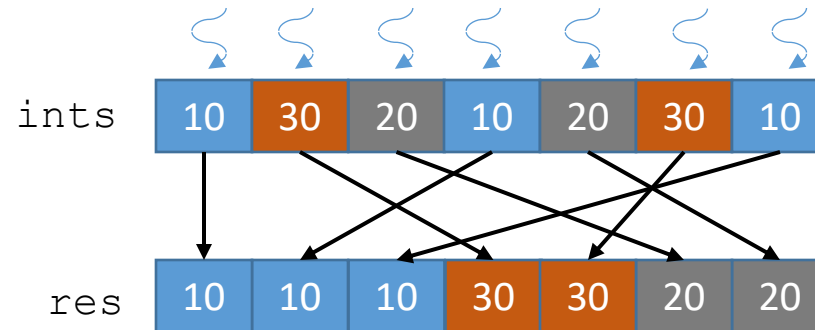
- grouping ~ shuffling
- input item \rightarrow output offset such that groups are contiguous



Parallel GroupBy

Process each input element in parallel

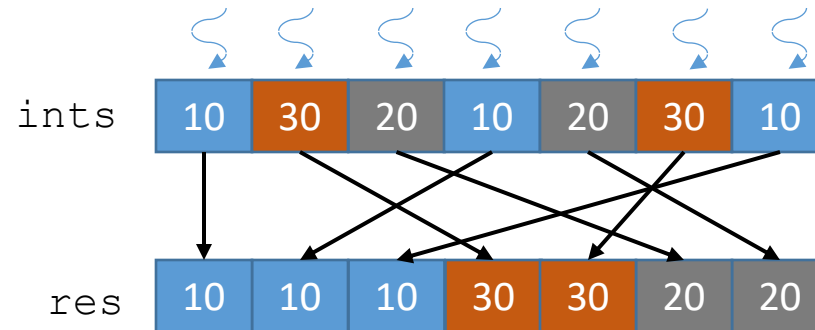
- grouping ~ shuffling
- input item \rightarrow output offset such that groups are contiguous



Parallel GroupBy

Process each input element in parallel

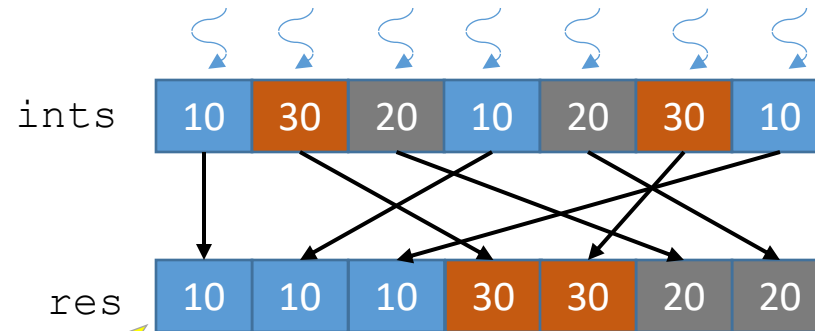
- grouping ~ shuffling
- input item \rightarrow output offset such that groups are contiguous
- output offset = group offset + item number
- ... but how to get the group offset, item number?



Parallel GroupBy

Process each input element in parallel

- grouping ~ shuffling
- input item \rightarrow output offset such that groups are contiguous
- output offset = group offset + item number
- ... but how to get the group offset, item number?

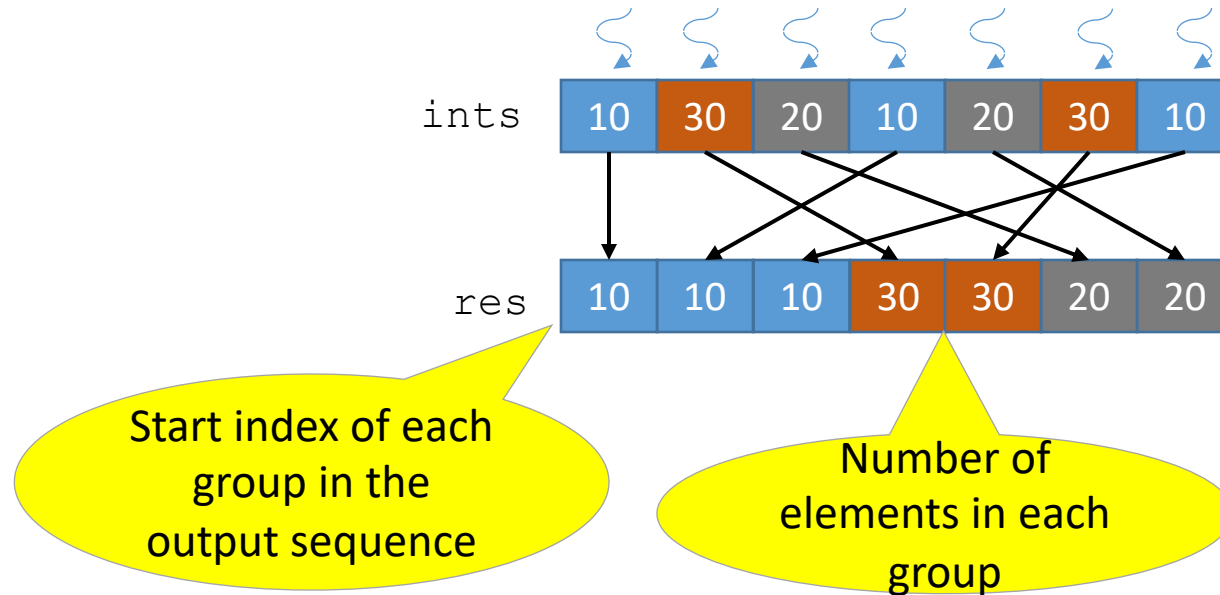


Start index of each group in the output sequence

Parallel GroupBy

Process each input element in parallel

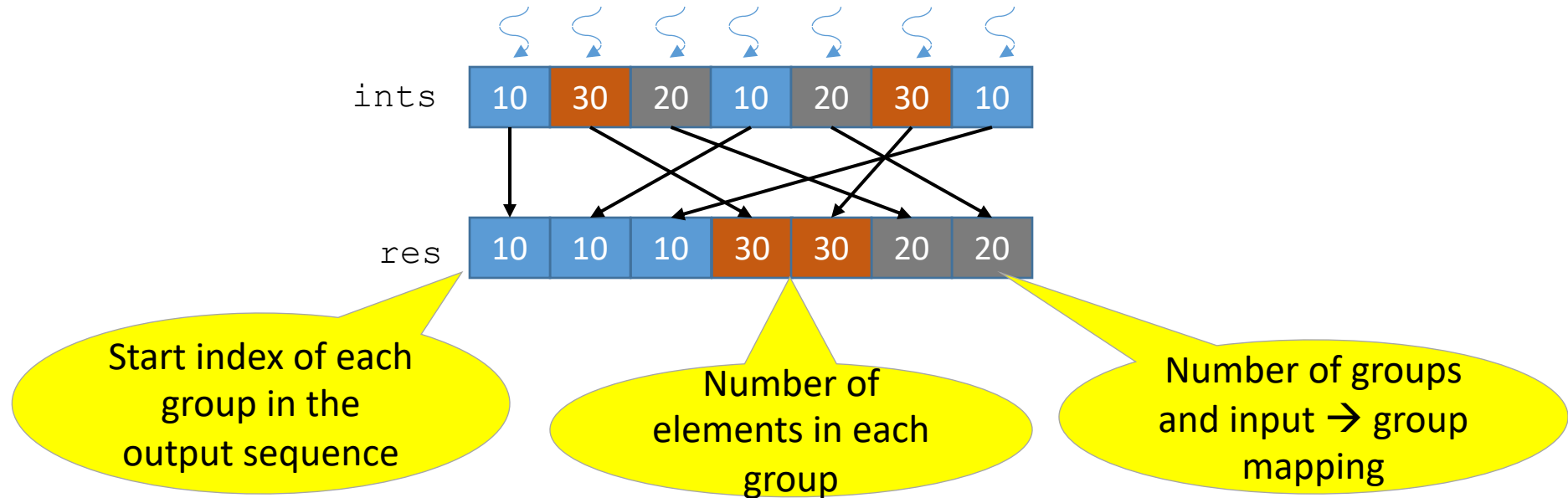
- grouping ~ shuffling
- input item \rightarrow output offset such that groups are contiguous
- output offset = group offset + item number
- ... but how to get the group offset, item number?



Parallel GroupBy

Process each input element in parallel

- grouping ~ shuffling
- input item \rightarrow output offset such that groups are contiguous
- output offset = group offset + item number
- ... but how to get the group offset, item number?

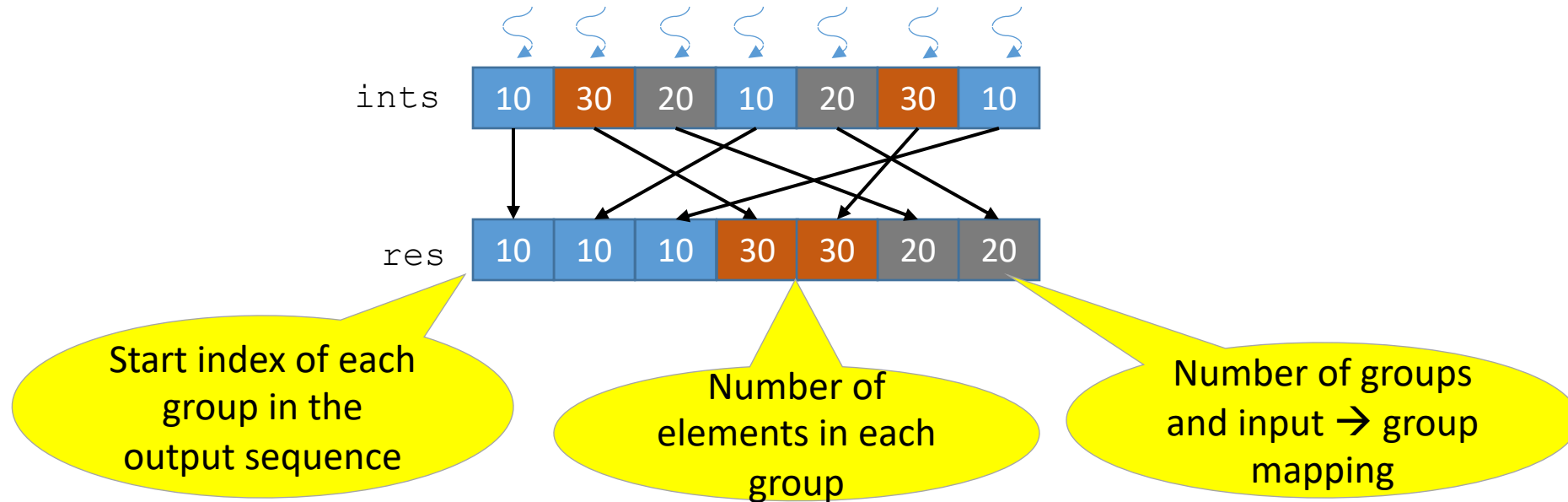


Parallel GroupBy



Process each input element in parallel

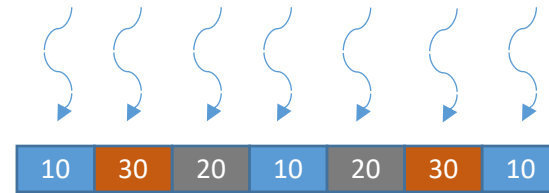
- grouping ~ shuffling
- input item \rightarrow output offset such that groups are contiguous
- output offset = group offset + item number
- ... but how to get the group offset, item number?



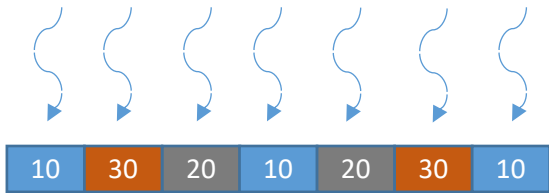
GroupBy using parallel primitives

10	30	20	10	20	30	10
----	----	----	----	----	----	----

GroupBy using parallel primitives



GroupBy using parallel primitives

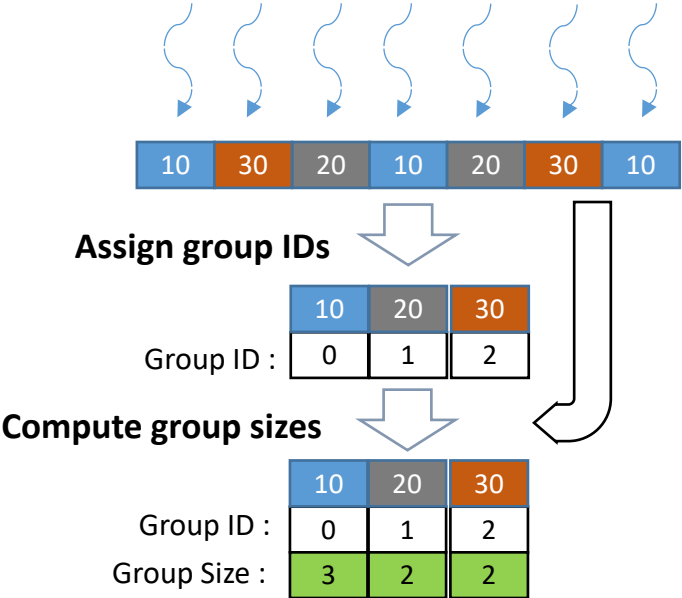


Assign group IDs

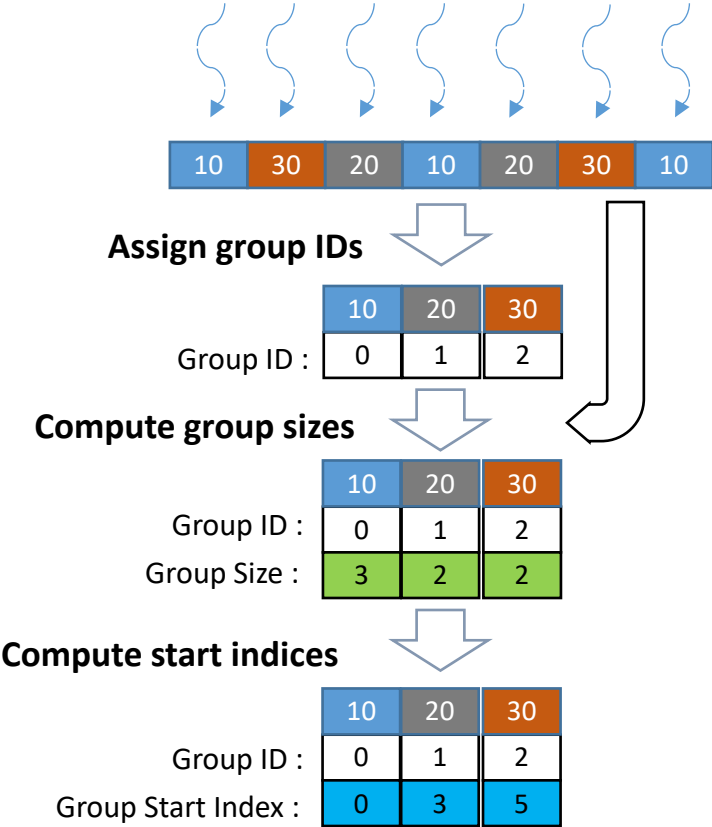


	10	20	30
Group ID :	0	1	2

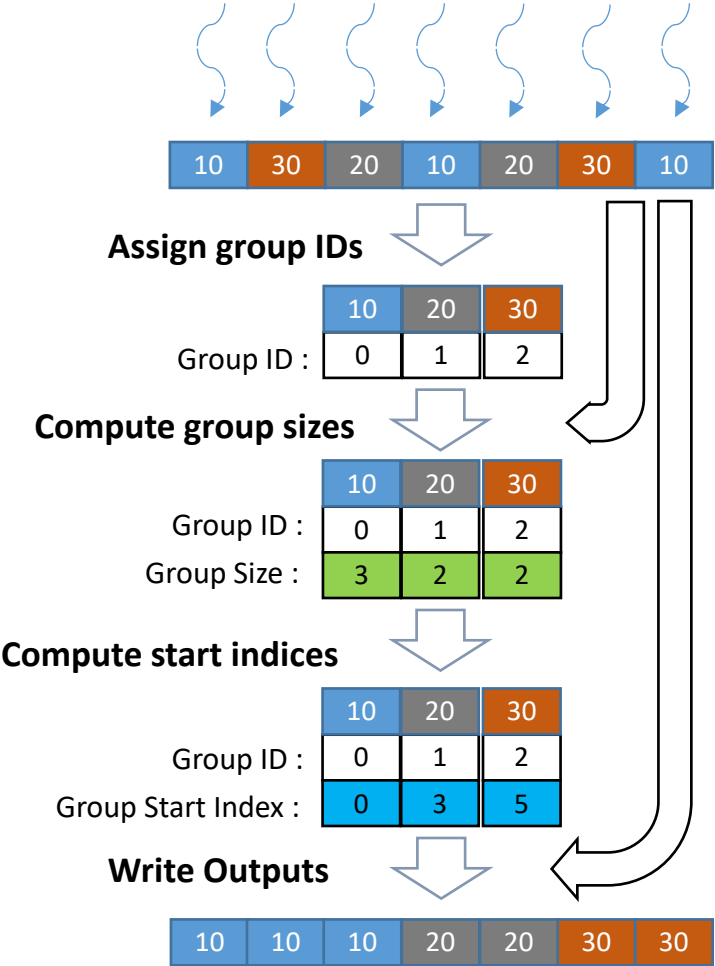
GroupBy using parallel primitives



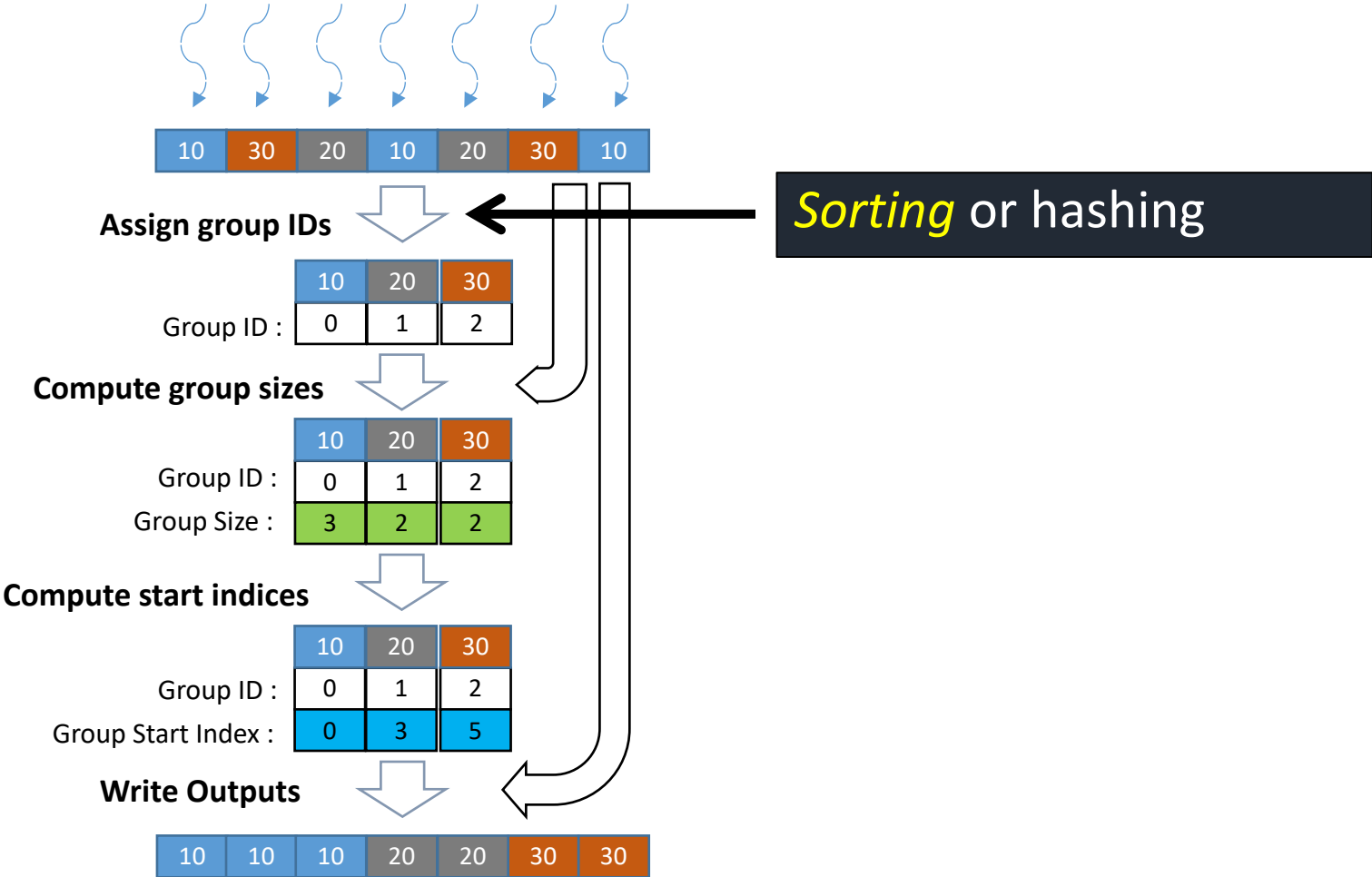
GroupBy using parallel primitives



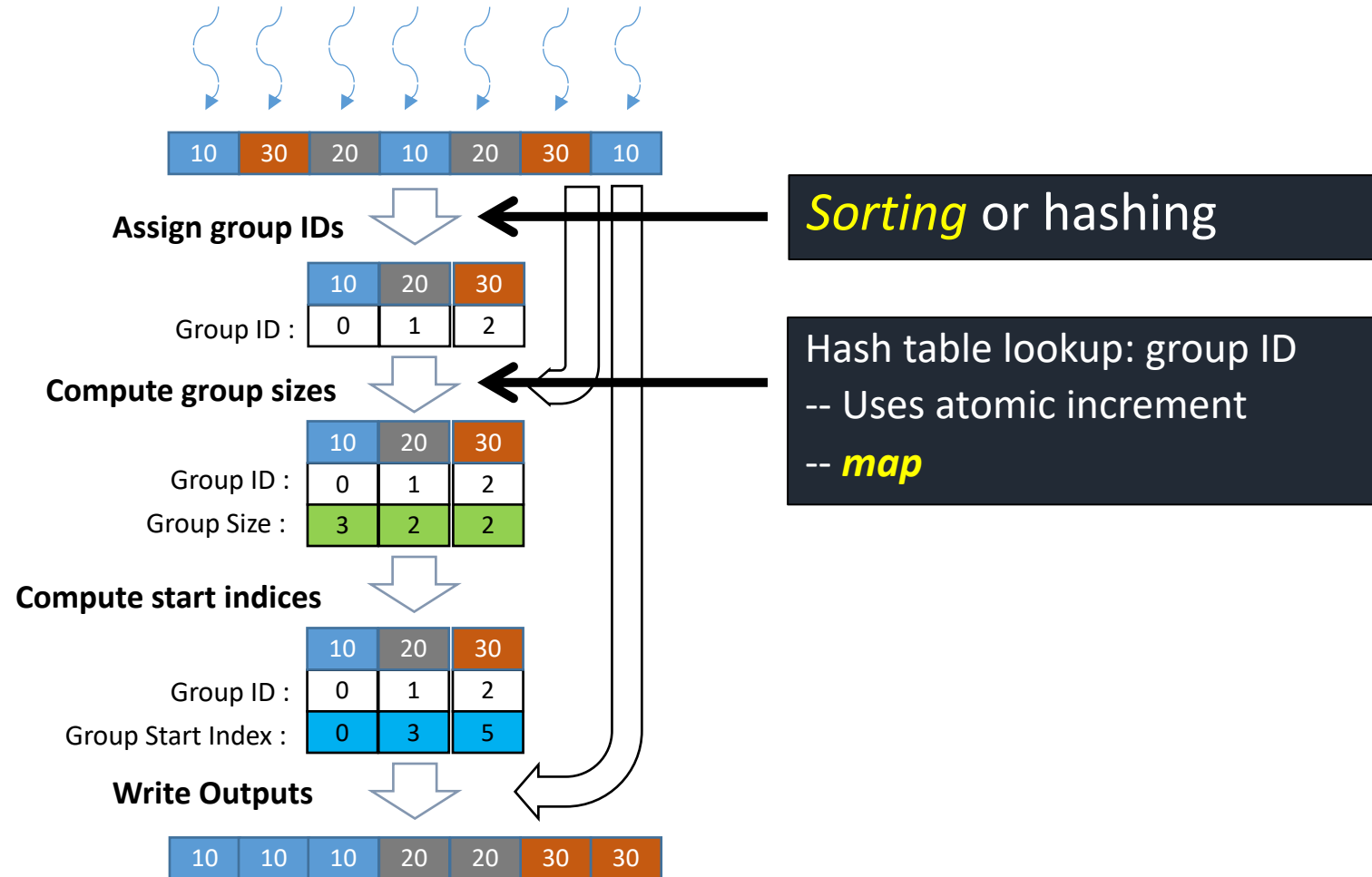
GroupBy using parallel primitives



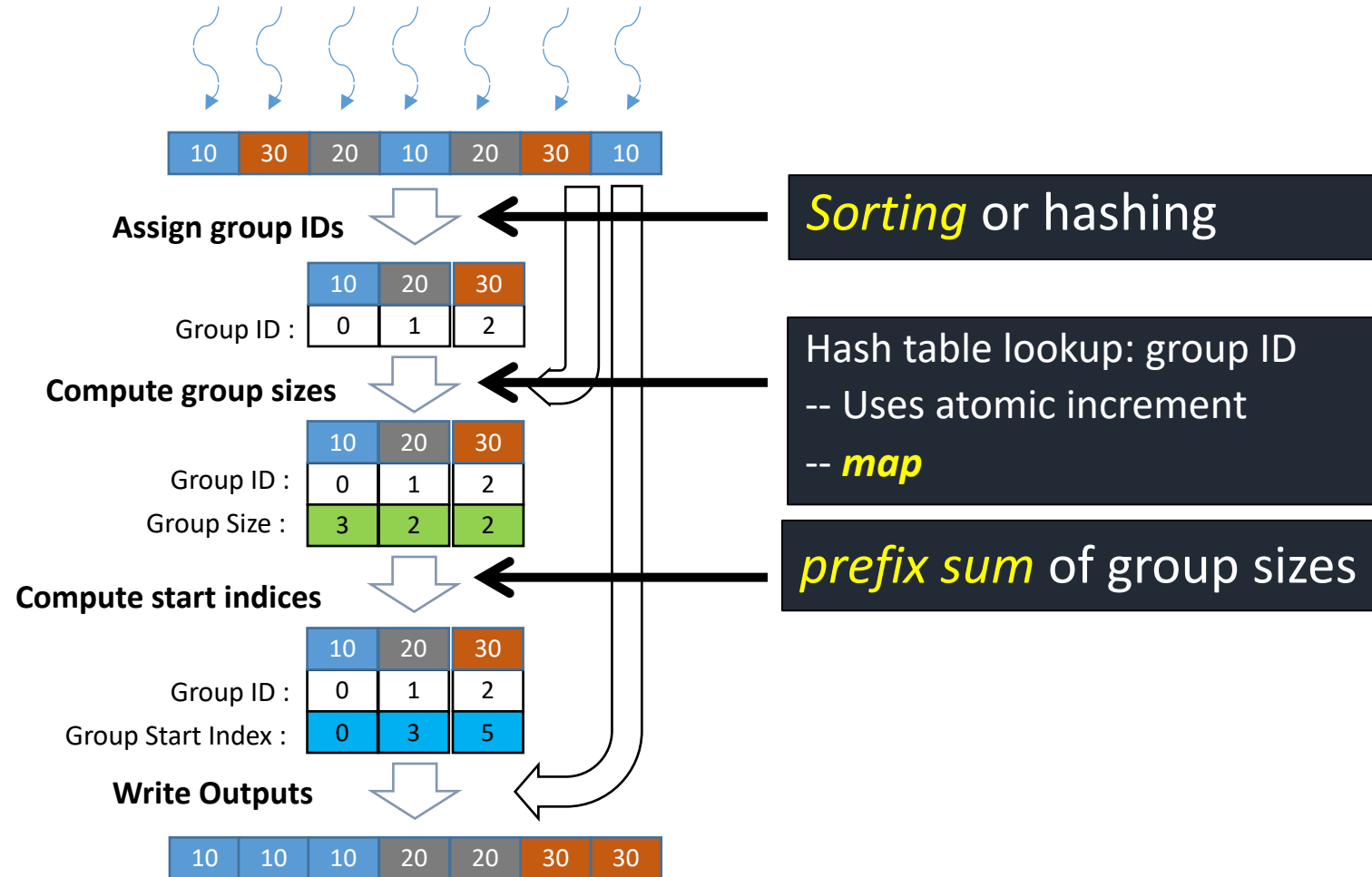
GroupBy using parallel primitives



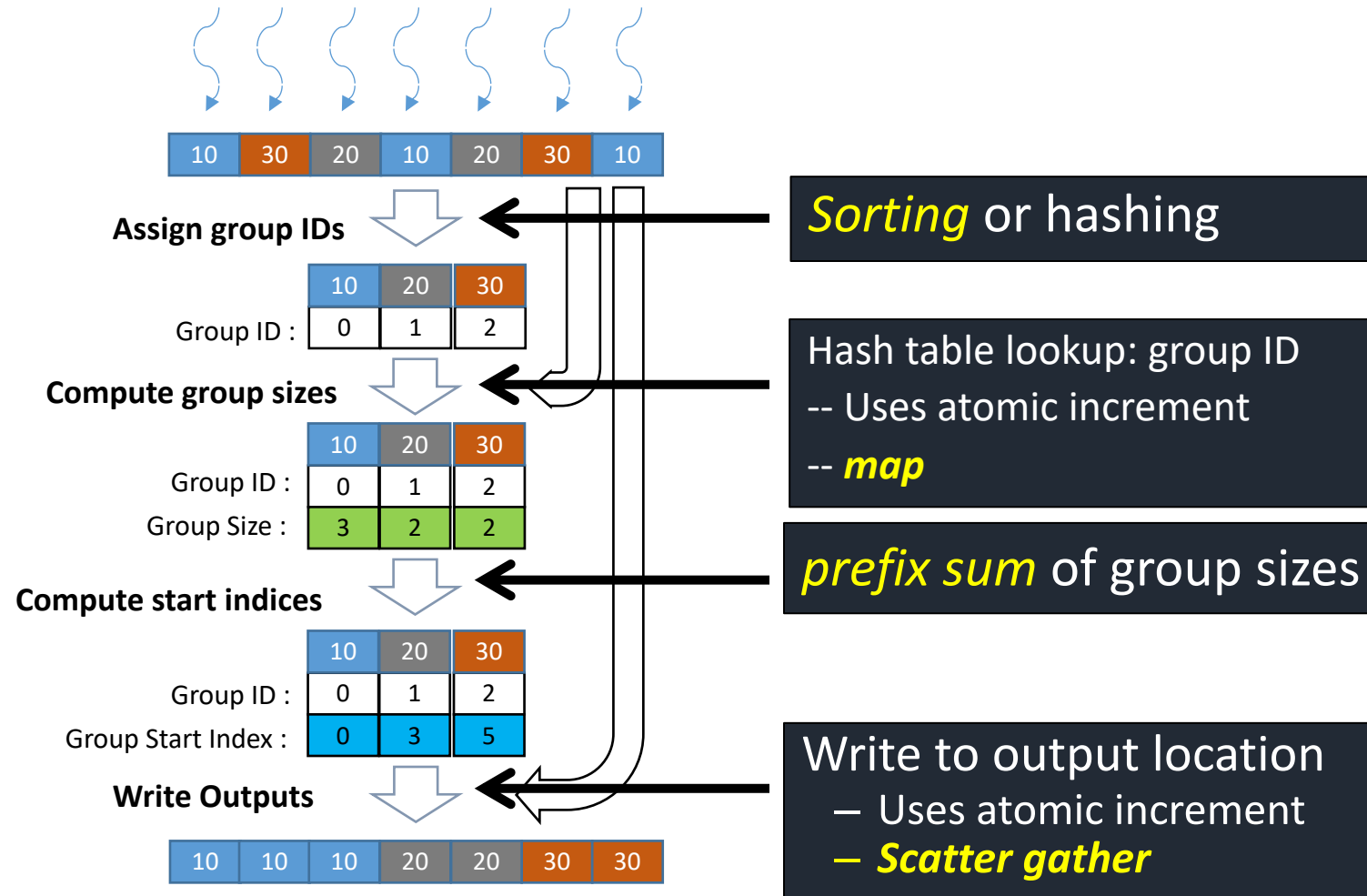
GroupBy using parallel primitives



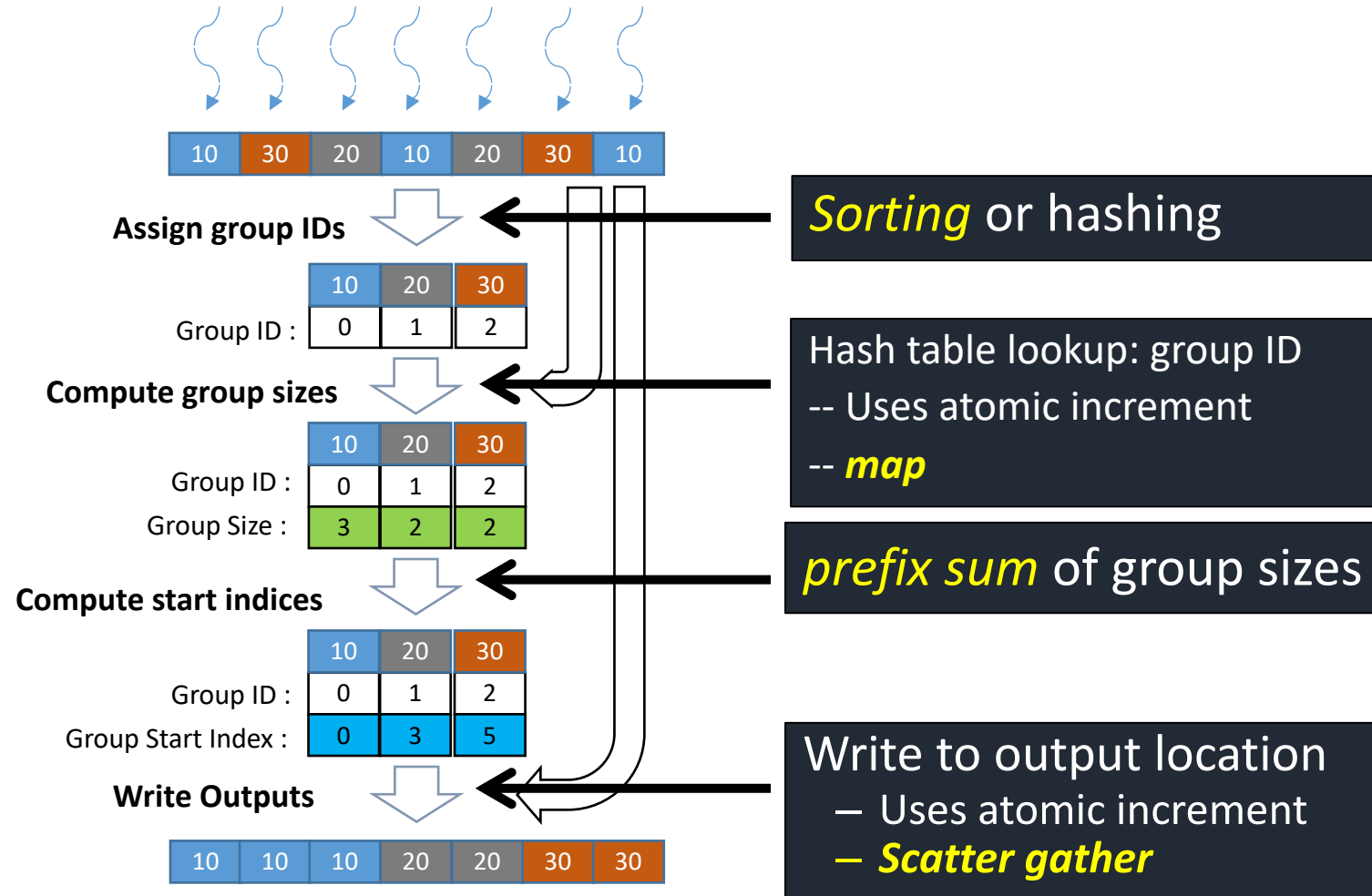
GroupBy using parallel primitives



GroupBy using parallel primitives



GroupBy using parallel primitives



We'll revisit after more CUDA background...

Sort

Many variations

- Enumeration sort
- Bitonic sort
- Merge sort
- Parallel Quicksort
- Radix sort
- Sample sort
- ...

Summary

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

- Reductions
- Scans
- Re-orderings (scatter/gather)
- Sort
- Map