

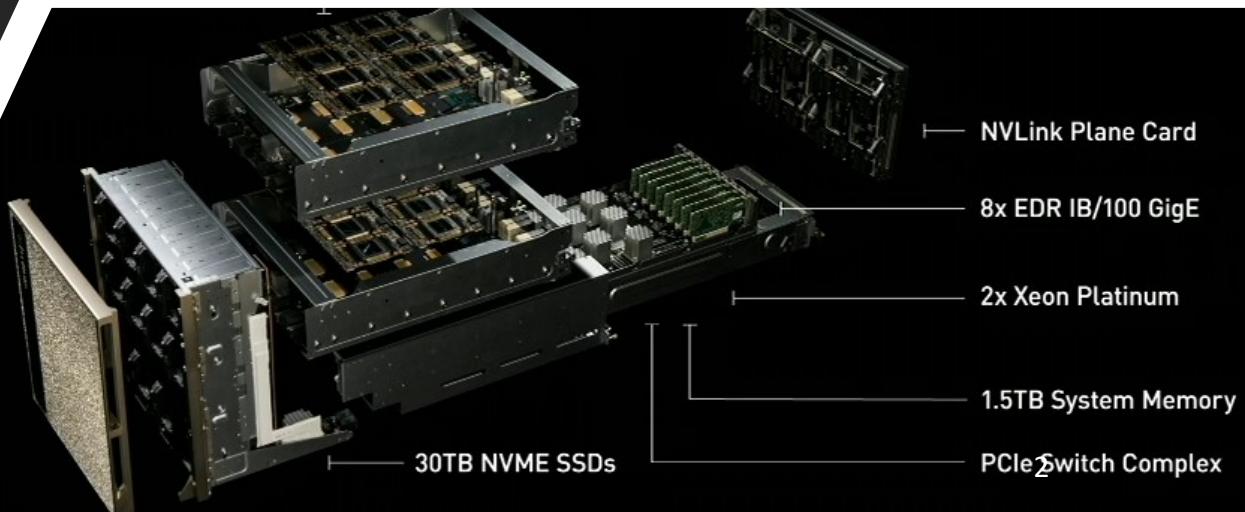
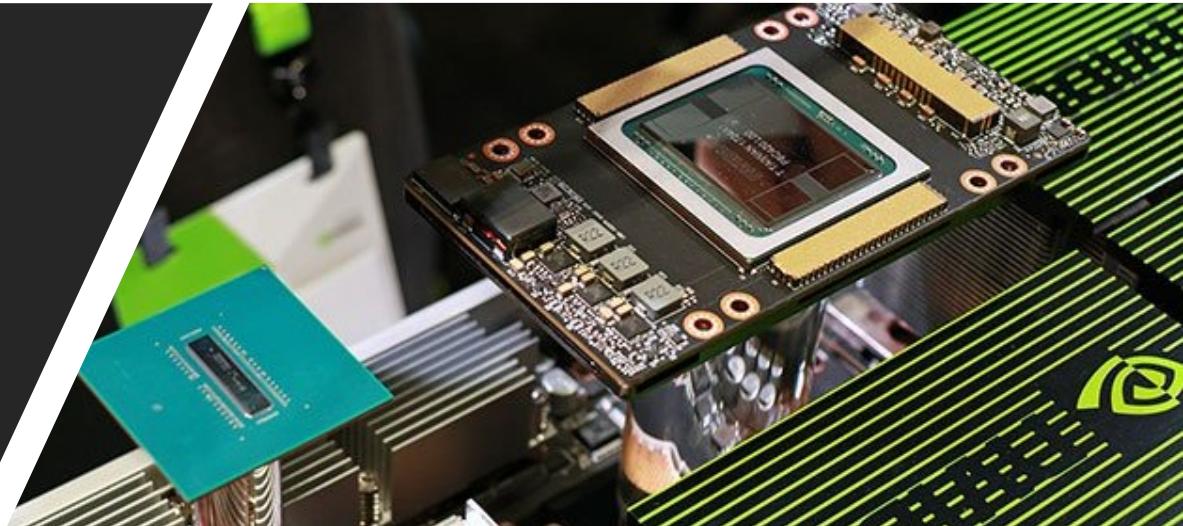
# Parallel Architectures Parallel Algorithms CUDA

Chris Rossbach

cs378h

# Outline for Today

- Questions?
- Administrivia
  - Eldar-\* machines should be available
- Agenda
  - Parallel Algorithms
  - CUDA
- Acknowledgements:  
[http://developer.download.nvidia.com/compute/developertools/materials/presentations/cuda\\_language/Introduction\\_to\\_CUDA\\_C.pptx](http://developer.download.nvidia.com/compute/developertools/materials/presentations/cuda_language/Introduction_to_CUDA_C.pptx)

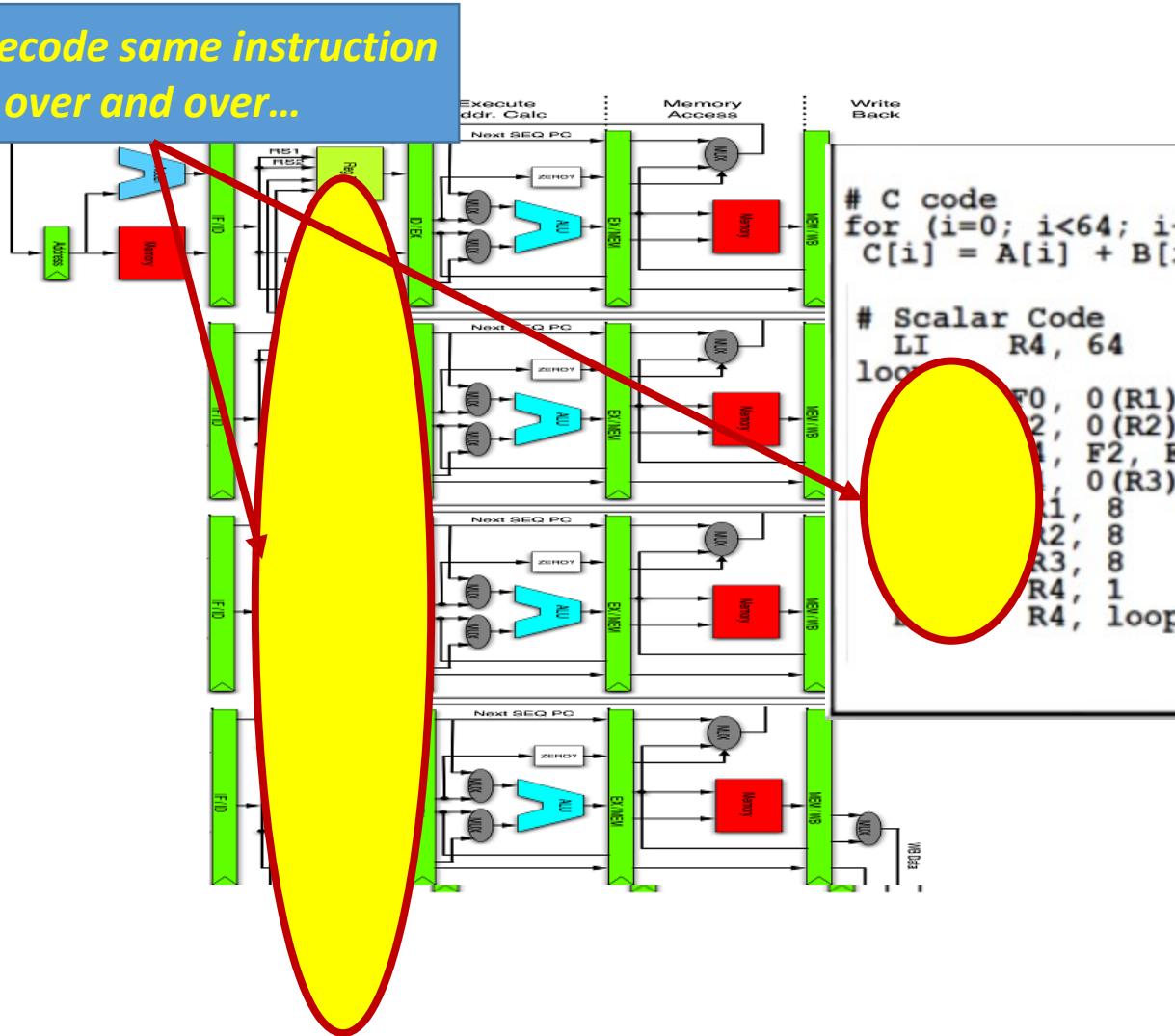


# Faux Quiz Questions

- 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?

*Dont decode same instruction over and over...*



Imp

- Instruction
- Scalar

Vector Load and Store Instructions

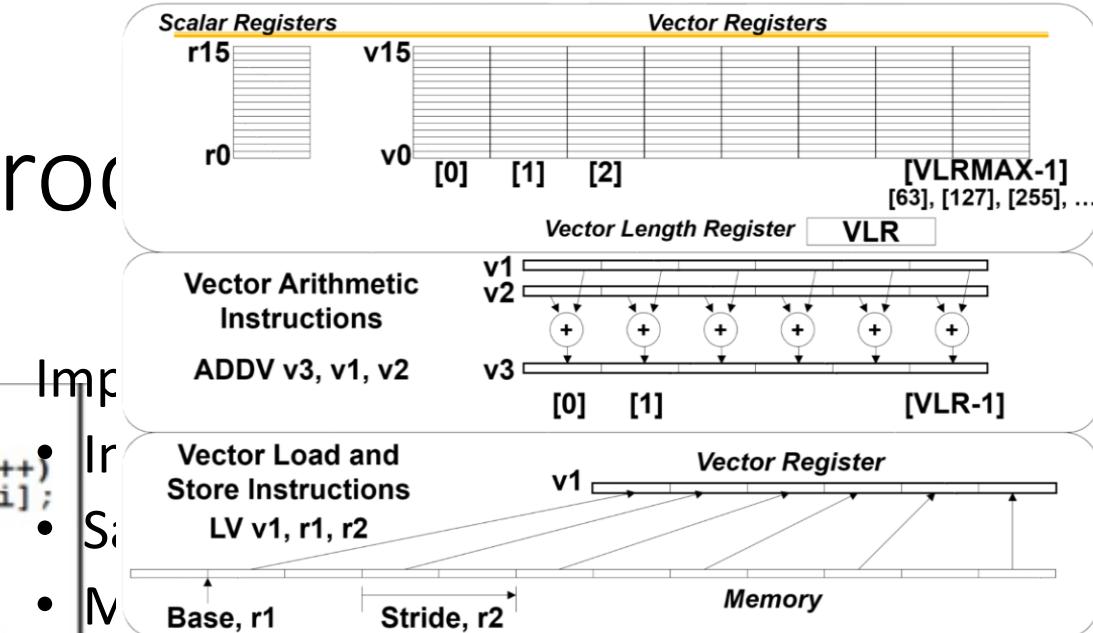
LV v1, r1, r2  
Base, r1      Stride, r2

- Multiple different operands in parallel

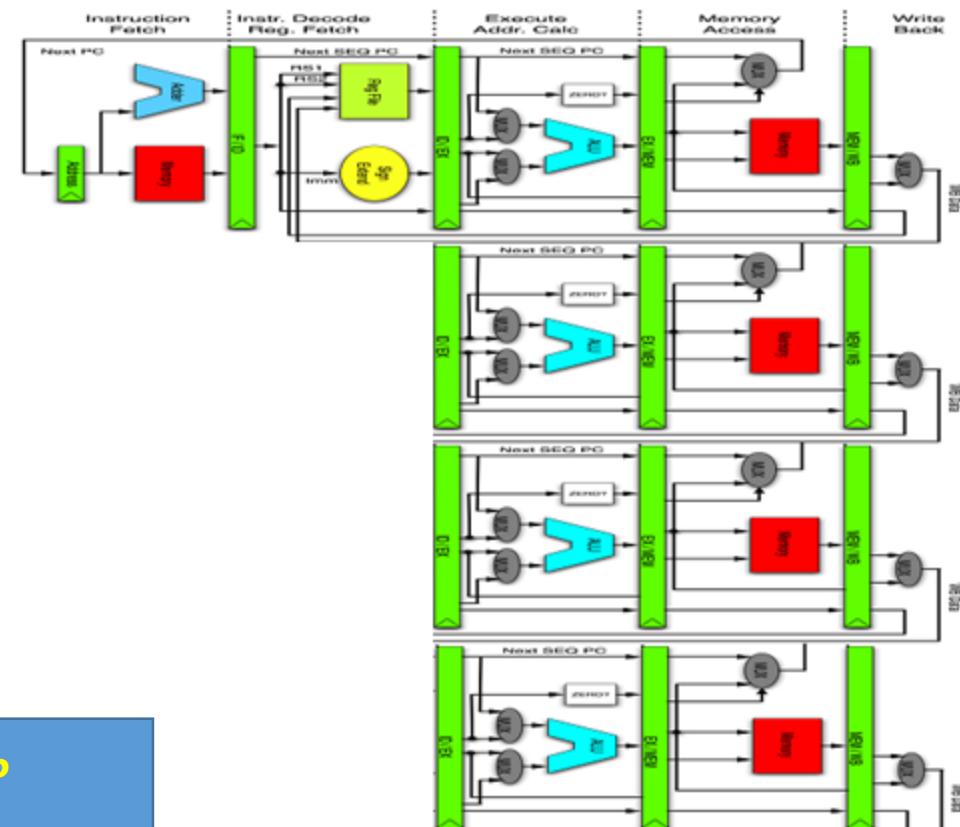
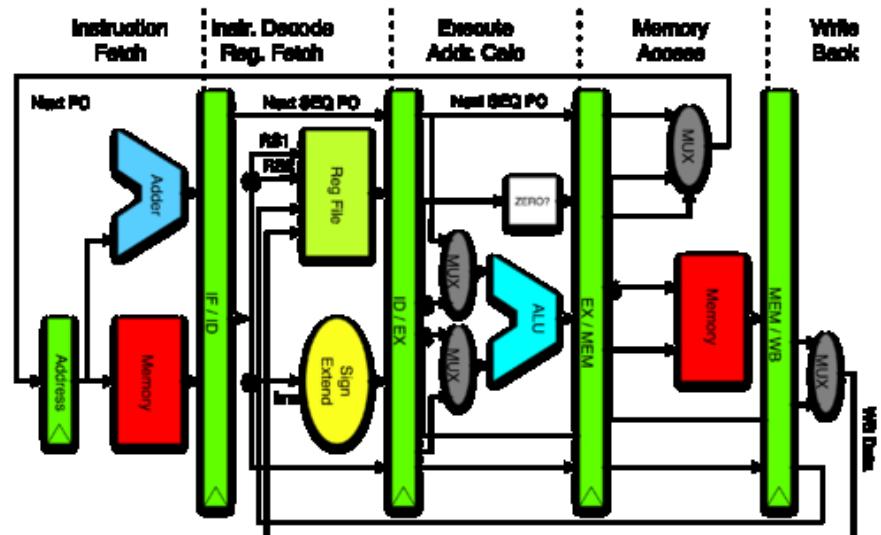
# C code  
for (i=0; i<64; i++)  
C[i] = A[i] + B[i];

# 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?



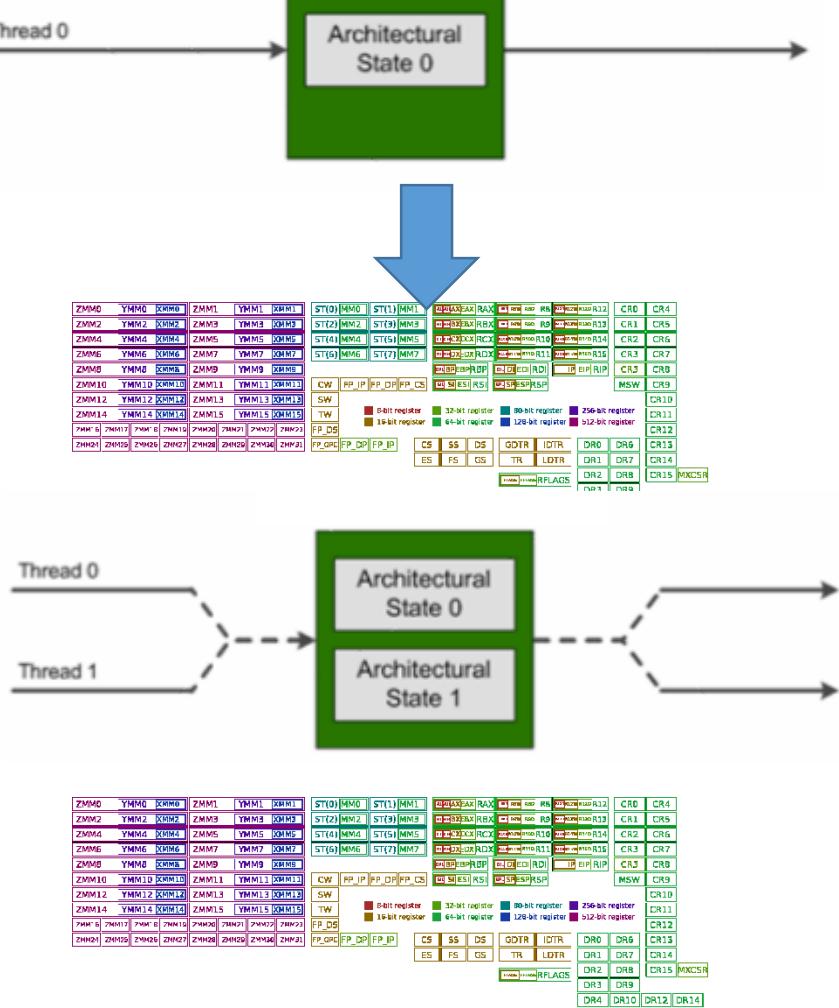
*What are the potential bottlenecks here?  
When can it improve throughput?*

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

# Hardware multi-threading

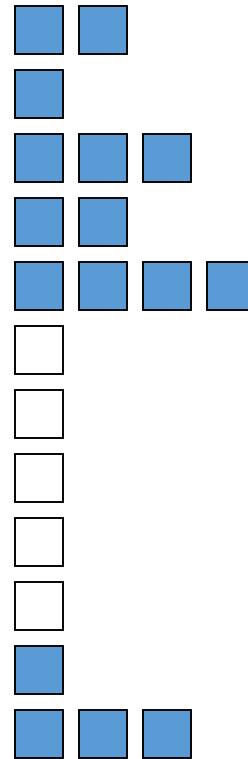
- Address memory bottleneck
- Share exec unit across
  - Instruction streams
  - Switch on stalls
- Looks like multiple cores to the OS
- Three variants:
  - Coarse
  - Fine-grain
  - Simultaneous

ZMM0	YMM0	XMM0	ZMM1	YMM1	XMM1	ST(0) [MM0]	ST(1) [MM1]	QWORD[RAX]	C1 [MM0]	R4	CR0	CR4
ZMM2	YMM2	XMM2	ZMM3	YMM3	XMM3	ST(2) [MM2]	ST(3) [MM3]	QWORD[RBX]	C1 [MM2]	R5	CR1	CR5
ZMM4	YMM4	XMM4	ZMM5	YMM5	XMM5	ST(4) [MM4]	ST(5) [MM5]	QWORD[RCX]	C1 [MM4]	R6	CR2	CR6
ZMM6	YMM6	XMM6	ZMM7	YMM7	XMM7	ST(6) [MM6]	ST(7) [MM7]	QWORD[RCX]	C1 [MM6]	R7	CR3	CR7
ZMM8	YMM8	XMM8	ZMM9	YMM9	XMM9	ST(8) [MM8]	ST(9) [MM9]	QWORD[RCX]	C1 [MM8]	R8	CR4	CR8
ZMM10	YMM10	XMM10	ZMM11	YMM11	XMM11	ST(10) [MM10]	ST(11) [MM11]	QWORD[RDX]	C1 [MM10]	R9	CR5	CR9
ZMM12	YMM12	XMM12	ZMM13	YMM13	XMM13	ST(12) [MM12]	ST(13) [MM13]	QWORD[RDX]	C1 [MM12]	R10	CR6	CR10
ZMM14	YMM14	XMM14	ZMM15	YMM15	XMM15	ST(14) [MM14]	ST(15) [MM15]	QWORD[RDX]	C1 [MM14]	R11	CR7	CR11
ZMM16	YMM16	XMM16	ZMM17	YMM17	XMM17	ST(16) [MM16]	ST(17) [MM17]	QWORD[RDX]	C1 [MM16]	R12	CR8	CR12
ZMM18	YMM18	XMM18	ZMM19	YMM19	XMM19	ST(18) [MM18]	ST(19) [MM19]	QWORD[RDX]	C1 [MM18]	R13	CR9	CR13
ZMM20	YMM20	XMM20	ZMM21	YMM21	XMM21	ST(20) [MM20]	ST(21) [MM21]	QWORD[RDX]	C1 [MM20]	R14	CR10	CR14
ZMM22	YMM22	XMM22	ZMM23	YMM23	XMM23	ST(22) [MM22]	ST(23) [MM23]	QWORD[RDX]	C1 [MM22]	R15	CR11	CR15
ZMM24	YMM24	XMM24	ZMM25	YMM25	XMM25	ST(24) [MM24]	ST(25) [MM25]	QWORD[RDX]	C1 [MM24]	R16	CR12	CR16
ZMM26	YMM26	XMM26	ZMM27	YMM27	XMM27	ST(26) [MM26]	ST(27) [MM27]	QWORD[RDX]	C1 [MM26]	R17	CR13	CR17
ZMM28	YMM28	XMM28	ZMM29	YMM29	XMM29	ST(28) [MM28]	ST(29) [MM29]	QWORD[RDX]	C1 [MM28]	R18	CR14	CR18
ZMM30	YMM30	XMM30	ZMM31	YMM31	XMM31	ST(30) [MM30]	ST(31) [MM31]	QWORD[RDX]	C1 [MM30]	R19	CR15	CR19
ZMM32	YMM32	XMM32	ZMM33	YMM33	XMM33	ST(32) [MM32]	ST(33) [MM33]	QWORD[RDX]	C1 [MM32]	R20	CR16	CR20
ZMM34	YMM34	XMM34	ZMM35	YMM35	XMM35	ST(34) [MM34]	ST(35) [MM35]	QWORD[RDX]	C1 [MM34]	R21	CR17	CR21
ZMM36	YMM36	XMM36	ZMM37	YMM37	XMM37	ST(36) [MM36]	ST(37) [MM37]	QWORD[RDX]	C1 [MM36]	R22	CR18	CR22
ZMM38	YMM38	XMM38	ZMM39	YMM39	XMM39	ST(38) [MM38]	ST(39) [MM39]	QWORD[RDX]	C1 [MM38]	R23	CR19	CR23
ZMM40	YMM40	XMM40	ZMM41	YMM41	XMM41	ST(40) [MM40]	ST(41) [MM41]	QWORD[RDX]	C1 [MM40]	R24	CR20	CR24
ZMM42	YMM42	XMM42	ZMM43	YMM43	XMM43	ST(42) [MM42]	ST(43) [MM43]	QWORD[RDX]	C1 [MM42]	R25	CR21	CR25
ZMM44	YMM44	XMM44	ZMM45	YMM45	XMM45	ST(44) [MM44]	ST(45) [MM45]	QWORD[RDX]	C1 [MM44]	R26	CR22	CR26
ZMM46	YMM46	XMM46	ZMM47	YMM47	XMM47	ST(46) [MM46]	ST(47) [MM47]	QWORD[RDX]	C1 [MM46]	R27	CR23	CR27
ZMM48	YMM48	XMM48	ZMM49	YMM49	XMM49	ST(48) [MM48]	ST(49) [MM49]	QWORD[RDX]	C1 [MM48]	R28	CR24	CR28
ZMM50	YMM50	XMM50	ZMM51	YMM51	XMM51	ST(50) [MM50]	ST(51) [MM51]	QWORD[RDX]	C1 [MM50]	R29	CR25	CR29
ZMM52	YMM52	XMM52	ZMM53	YMM53	XMM53	ST(52) [MM52]	ST(53) [MM53]	QWORD[RDX]	C1 [MM52]	R30	CR26	CR30
ZMM54	YMM54	XMM54	ZMM55	YMM55	XMM55	ST(54) [MM54]	ST(55) [MM55]	QWORD[RDX]	C1 [MM54]	R31	CR27	CR31
ZMM56	YMM56	XMM56	ZMM57	YMM57	XMM57	ST(56) [MM56]	ST(57) [MM57]	QWORD[RDX]	C1 [MM56]	R32	CR28	CR32
ZMM58	YMM58	XMM58	ZMM59	YMM59	XMM59	ST(58) [MM58]	ST(59) [MM59]	QWORD[RDX]	C1 [MM58]	R33	CR29	CR33
ZMM60	YMM60	XMM60	ZMM61	YMM61	XMM61	ST(60) [MM60]	ST(61) [MM61]	QWORD[RDX]	C1 [MM60]	R34	CR30	CR34
ZMM62	YMM62	XMM62	ZMM63	YMM63	XMM63	ST(62) [MM62]	ST(63) [MM63]	QWORD[RDX]	C1 [MM62]	R35	CR31	CR35
ZMM64	YMM64	XMM64	ZMM65	YMM65	XMM65	ST(64) [MM64]	ST(65) [MM65]	QWORD[RDX]	C1 [MM64]	R36	CR32	CR36
ZMM66	YMM66	XMM66	ZMM67	YMM67	XMM67	ST(66) [MM66]	ST(67) [MM67]	QWORD[RDX]	C1 [MM66]	R37	CR33	CR37
ZMM68	YMM68	XMM68	ZMM69	YMM69	XMM69	ST(68) [MM68]	ST(69) [MM69]	QWORD[RDX]	C1 [MM68]	R38	CR34	CR38
ZMM70	YMM70	XMM70	ZMM71	YMM71	XMM71	ST(70) [MM70]	ST(71) [MM71]	QWORD[RDX]	C1 [MM70]	R39	CR35	CR39
ZMM72	YMM72	XMM72	ZMM73	YMM73	XMM73	ST(72) [MM72]	ST(73) [MM73]	QWORD[RDX]	C1 [MM72]	R40	CR36	CR40
ZMM74	YMM74	XMM74	ZMM75	YMM75	XMM75	ST(74) [MM74]	ST(75) [MM75]	QWORD[RDX]	C1 [MM74]	R41	CR37	CR41
ZMM76	YMM76	XMM76	ZMM77	YMM77	XMM77	ST(76) [MM76]	ST(77) [MM77]	QWORD[RDX]	C1 [MM76]	R42	CR38	CR42
ZMM78	YMM78	XMM78	ZMM79	YMM79	XMM79	ST(78) [MM78]	ST(79) [MM79]	QWORD[RDX]	C1 [MM78]	R43	CR39	CR43
ZMM80	YMM80	XMM80	ZMM81	YMM81	XMM81	ST(80) [MM80]	ST(81) [MM81]	QWORD[RDX]	C1 [MM80]	R44	CR40	CR44
ZMM82	YMM82	XMM82	ZMM83	YMM83	XMM83	ST(82) [MM82]	ST(83) [MM83]	QWORD[RDX]	C1 [MM82]	R45	CR41	CR45
ZMM84	YMM84	XMM84	ZMM85	YMM85	XMM85	ST(84) [MM84]	ST(85) [MM85]	QWORD[RDX]	C1 [MM84]	R46	CR42	CR46
ZMM86	YMM86	XMM86	ZMM87	YMM87	XMM87	ST(86) [MM86]	ST(87) [MM87]	QWORD[RDX]	C1 [MM86]	R47	CR43	CR47
ZMM88	YMM88	XMM88	ZMM89	YMM89	XMM89	ST(88) [MM88]	ST(89) [MM89]	QWORD[RDX]	C1 [MM88]	R48	CR44	CR48
ZMM90	YMM90	XMM90	ZMM91	YMM91	XMM91	ST(90) [MM90]	ST(91) [MM91]	QWORD[RDX]	C1 [MM90]	R49	CR45	CR49
ZMM92	YMM92	XMM92	ZMM93	YMM93	XMM93	ST(92) [MM92]	ST(93) [MM93]	QWORD[RDX]	C1 [MM92]	R50	CR46	CR50
ZMM94	YMM94	XMM94	ZMM95	YMM95	XMM95	ST(94) [MM94]	ST(95) [MM95]	QWORD[RDX]	C1 [MM94]	R51	CR47	CR51
ZMM96	YMM96	XMM96	ZMM97	YMM97	XMM97	ST(96) [MM96]	ST(97) [MM97]	QWORD[RDX]	C1 [MM96]	R52	CR48	CR52
ZMM98	YMM98	XMM98	ZMM99	YMM99	XMM99	ST(98) [MM98]	ST(99) [MM99]	QWORD[RDX]	C1 [MM98]	R53	CR49	CR53
ZMM100	YMM100	XMM100	ZMM101	YMM101	XMM101	ST(100) [MM100]	ST(101) [MM101]	QWORD[RDX]	C1 [MM100]	R54	CR50	CR54
ZMM102	YMM102	XMM102	ZMM103	YMM103	XMM103	ST(102) [MM102]	ST(103) [MM103]	QWORD[RDX]	C1 [MM102]	R55	CR51	CR55
ZMM104	YMM104	XMM104	ZMM105	YMM105	XMM105	ST(104) [MM104]	ST(105) [MM105]	QWORD[RDX]	C1 [MM104]	R56	CR52	CR56
ZMM106	YMM106	XMM106	ZMM107	YMM107	XMM107	ST(106) [MM106]	ST(107) [MM107]	QWORD[RDX]	C1 [MM106]	R57	CR53	CR57
ZMM108	YMM108	XMM108	ZMM109	YMM109	XMM109	ST(108) [MM108]	ST(109) [MM109]	QWORD[RDX]	C1 [MM108]	R58	CR54	CR58
ZMM110	YMM110	XMM110	ZMM111	YMM111	XMM111	ST(110) [MM110]	ST(111) [MM111]	QWORD[RDX]	C1 [MM110]	R59	CR55	CR59
ZMM112	YMM112	XMM112	ZMM113	YMM113	XMM113	ST(112) [MM112]	ST(113) [MM113]	QWORD[RDX]	C1 [MM112]	R60	CR56	CR60
ZMM114	YMM114	XMM114	ZMM115	YMM115	XMM115	ST(114) [MM114]	ST(115) [MM115]	QWORD[RDX]	C1 [MM114]	R61	CR57	CR61
ZMM116	YMM116	XMM116	ZMM117	YMM117	XMM117	ST(116) [MM116]	ST(117) [MM117]	QWORD[RDX]	C1 [MM116]	R62	CR58	CR62
ZMM118	YMM118	XMM118	ZMM119	YMM119	XMM119	ST(118) [MM118]	ST(119) [MM119]	QWORD[RDX]	C1 [MM118]	R63	CR59	CR63
ZMM120	YMM120	XMM120	ZMM121	YMM121	XMM121	ST(120) [MM120]	ST(121) [MM121]	QWORD[RDX]	C1 [MM120]	R64	CR60	CR64
ZMM122	YMM122	XMM122	ZMM123	YMM123	XMM123	ST(122) [MM122]	ST(123) [MM123]	QWORD[RDX]	C1 [MM122]	R65	CR61	CR65
ZMM124	YMM124	XMM124	ZMM125	YMM125	XMM125	ST(124) [MM124]	ST(125) [MM125]	QWORD[RDX]	C1 [MM124]	R66	CR62	CR66
ZMM126	YMM126	XMM126	ZMM127	YMM127	XMM127	ST(126) [MM126]	ST(127) [MM127]	QWORD[RDX]	C1 [MM126]	R67	CR63	CR67
ZMM128	YMM128	XMM128	ZMM129	YMM129	XMM129	ST(128) [MM128]	ST(129) [MM129]	QWORD[RDX]	C1 [MM128]	R68	CR64	CR68
ZMM130	YMM130	XMM130	ZMM131	YMM131	XMM131	ST(130) [MM130]	ST(131) [MM131]	QWORD[RDX]	C1 [MM130]	R69	CR65	CR69
ZMM132	YMM132	XMM132	ZMM133	YMM133	XMM133	ST(132) [MM132]	ST(133) [MM133]	QWORD[RDX]	C1 [MM132]	R70	CR66	CR70
ZMM134	YMM134	XMM134	ZMM135	YMM135	XMM135	ST(134) [MM134]	ST(135) [MM135]	QWORD[RDX]	C1 [MM134]	R71	CR67	CR71
ZMM136	YMM136	XMM136	ZMM137	YMM137	XMM137	ST(136) [MM136]	ST(137) [MM137]	QWORD[RDX]	C1 [MM136]	R72	CR68	CR72
ZMM138	YMM138	XMM138	ZMM139	YMM139	XMM139	ST(138) [MM138]	ST(139) [MM139]	QWORD[RDX]	C1 [MM138]	R73	CR69	CR73
ZMM140	YMM140	XMM140	ZMM141	YMM141	XMM141	ST(140) [MM140]	ST(141) [MM141]	QWORD[RDX]	C1 [MM140]	R74	CR70	CR74
ZMM142	YMM142	XMM142	ZMM143	YMM143	XMM143	ST(142) [MM142]	ST(143) [MM143]	QWORD[RDX]	C1 [MM142]	R75	CR71	CR75
ZMM144	YMM144	XMM144	ZMM145	YMM145	XMM145	ST(144) [MM144]	ST(145) [MM145]	QWORD[RDX]	C1 [MM144]	R76	CR72	CR76
ZMM146	YMM146	XMM146	ZMM147	YMM147	XMM147	ST(146) [MM146]	ST(147) [MM147]	QWORD[RDX]	C1 [MM146]	R77	CR73	CR77
ZMM148	YMM148	XMM148	ZMM149	YMM149	XMM149	ST(148) [MM148]	ST(149) [MM149]	QWORD[RDX]	C1 [MM148]	R78	CR74	CR78
ZMM150	YMM150	XMM150	ZMM151	YMM151	XMM151	ST(150) [MM150]	ST(151) [MM151]	QWORD[RDX]	C1 [MM150]	R79	CR75	CR79
ZMM152	YMM152	XMM152	ZMM153	YMM153	XMM153	ST(152) [MM152]	ST(153) [MM153]	QWORD[RDX]	C1 [MM152]	R80	CR76	CR80
ZMM154	YMM154	XMM154	ZMM155	YMM155	XMM155	ST(154) [MM154]	ST(155) [MM155]	QWORD[RDX]	C1 [MM154]	R81	CR77	CR81
ZMM156	YMM156	XMM156	ZMM157	YMM157	XMM157	ST(156) [MM156]	ST(157) [MM157]	QWORD[RDX]	C1 [MM156]	R82	CR78	CR82
ZMM158	YMM158	XMM158	ZMM159	YMM159	XMM159	ST(158) [MM158]	ST(159) [MM159]	QWORD[RDX]	C1 [MM158]	R83	CR79	CR83
ZMM160	YMM160	XMM160	ZMM161	YMM161	XMM161	ST(160) [MM160]	ST(161) [MM161]	QWORD[RDX]	C1 [MM160]	R84	CR80	CR84

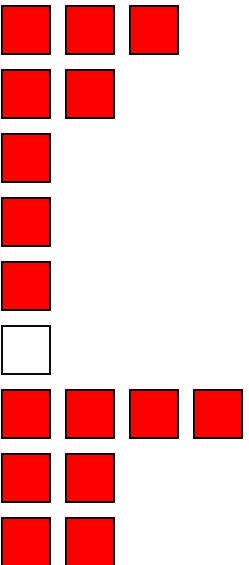


# Running example

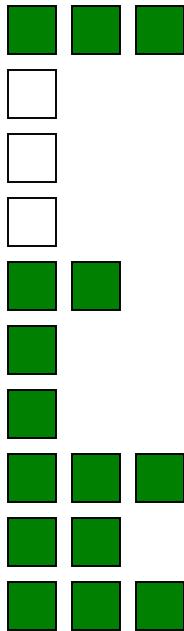
Thread A



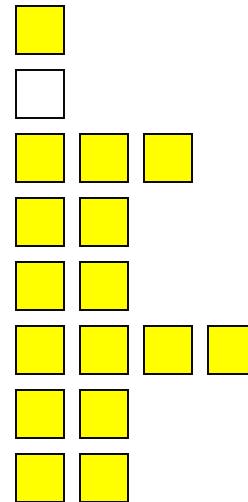
Thread B



Thread C



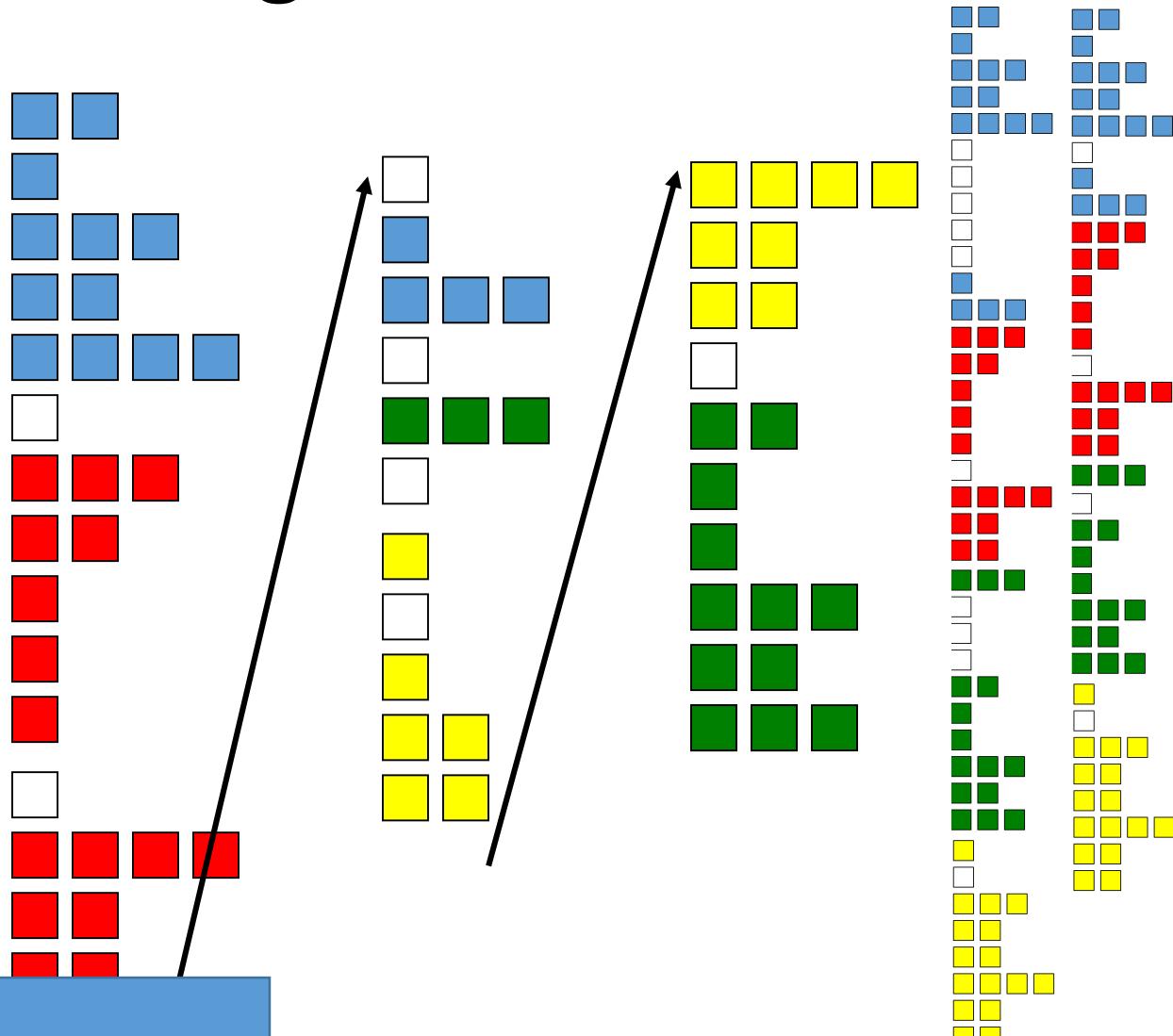
Thread D



- Colors → pipeline full
- White → stall

# Coarse- grained multithreading

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

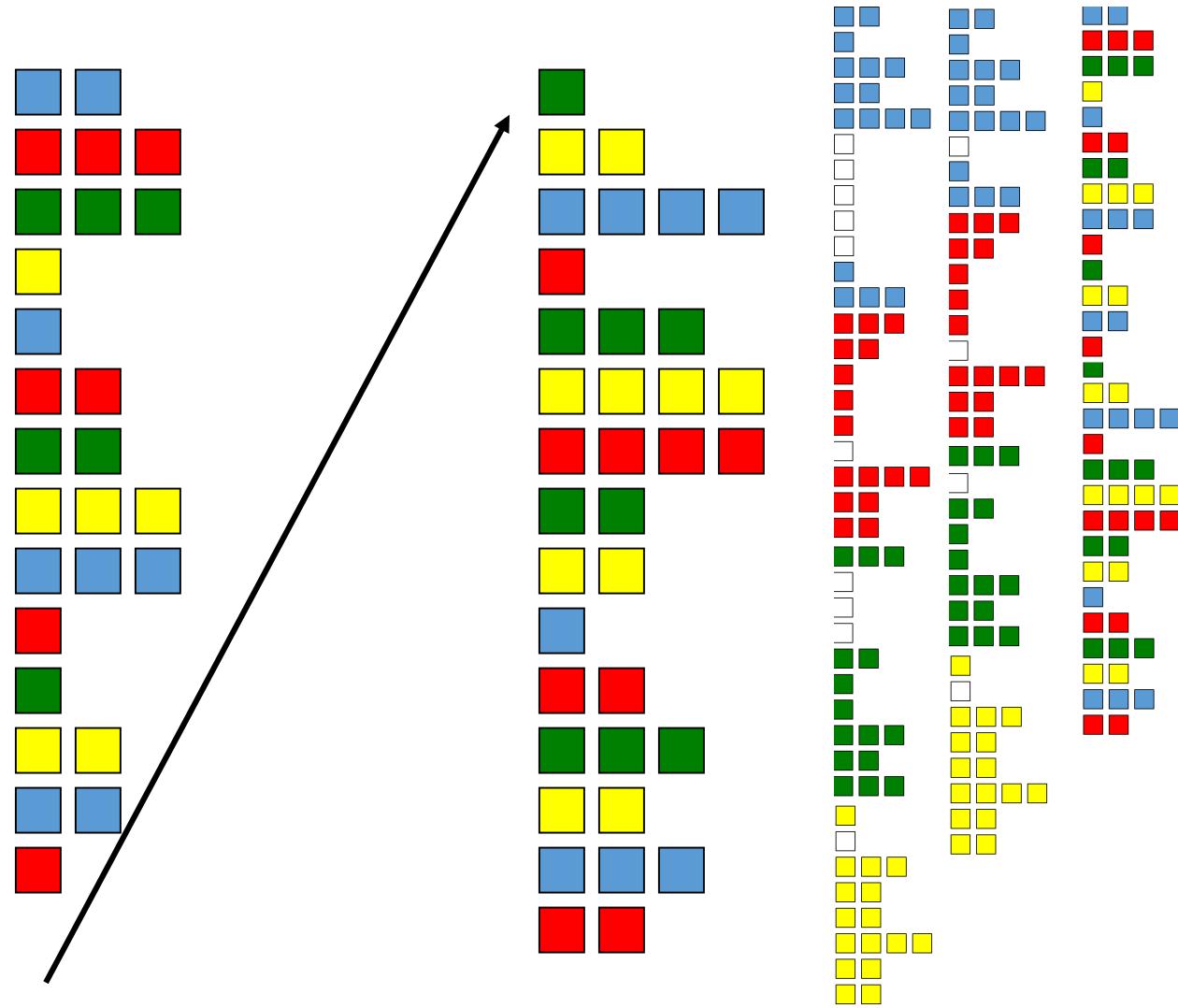


# *Pros? Cons?*

# Fine-grained multithreading

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

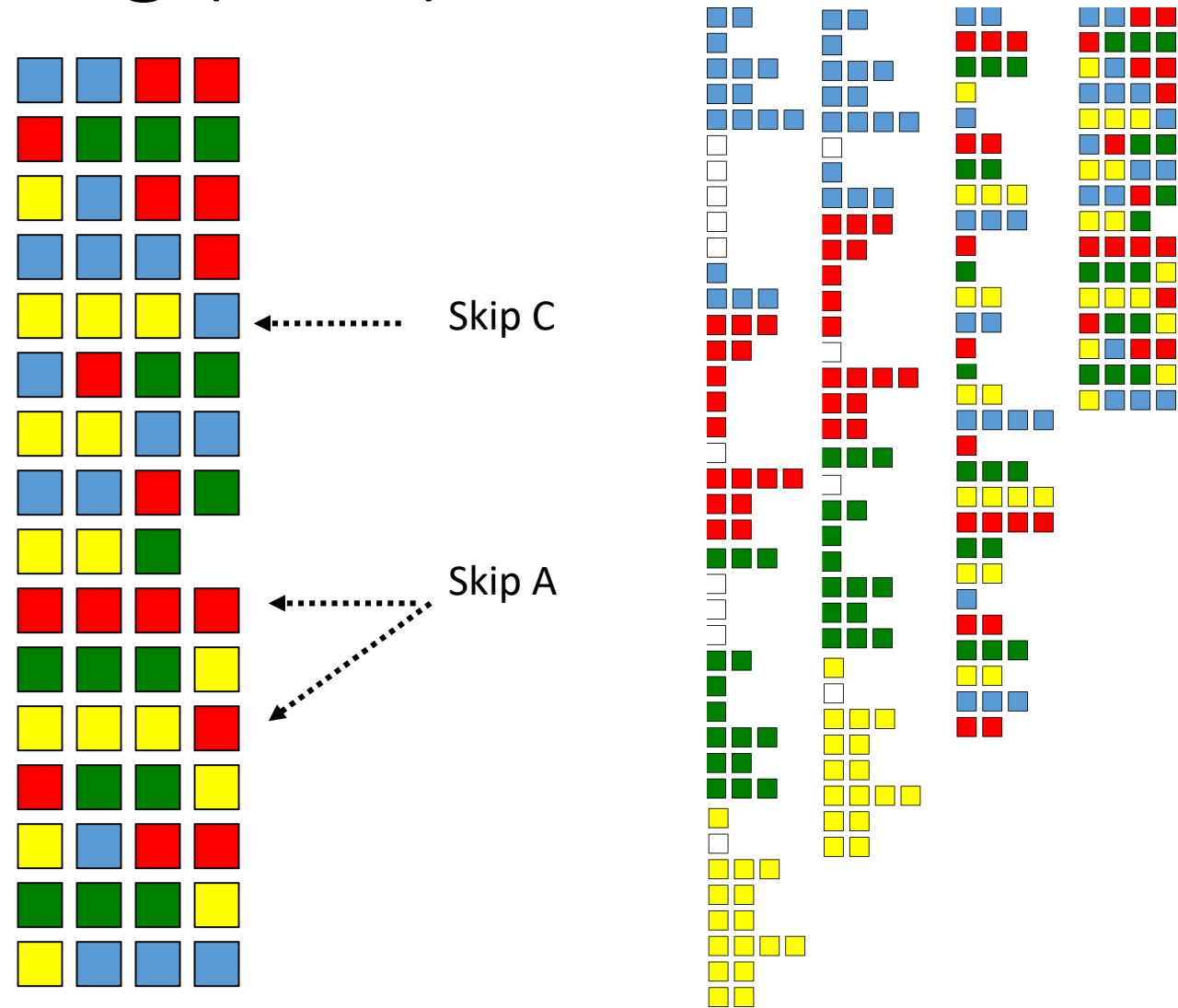
*Pros? Cons?*



# Simultaneous Multithreading (SMT)

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

*Pros? Cons?*



# Why Vector and Multithreading Background?

GPU:

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

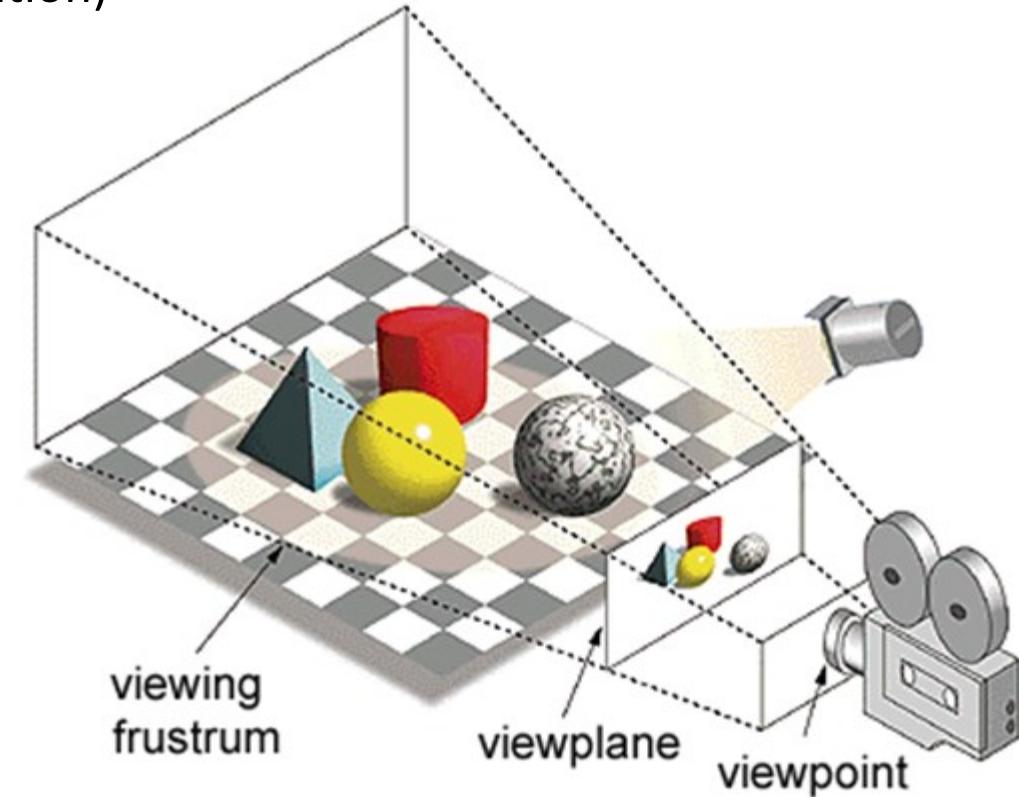
# Graphics $\approx$ Rendering

## Inputs

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

## Output

- 2D projection seen from the view-point



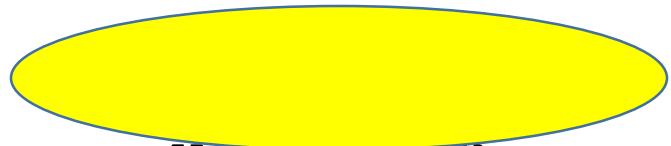
# Grossly over-simplified rendering algorithm

```
foreach(vertex v in model)
    map vmodel → vview
fragment[] frags = {};
foreach triangle t (v0, v1, v2)
    frags.add(rasterize(t));
foreach fragment f in frags
    choose_color(f);
display(visible_fragments(frags));
```



# Algorithm → Graphics Pipeline

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

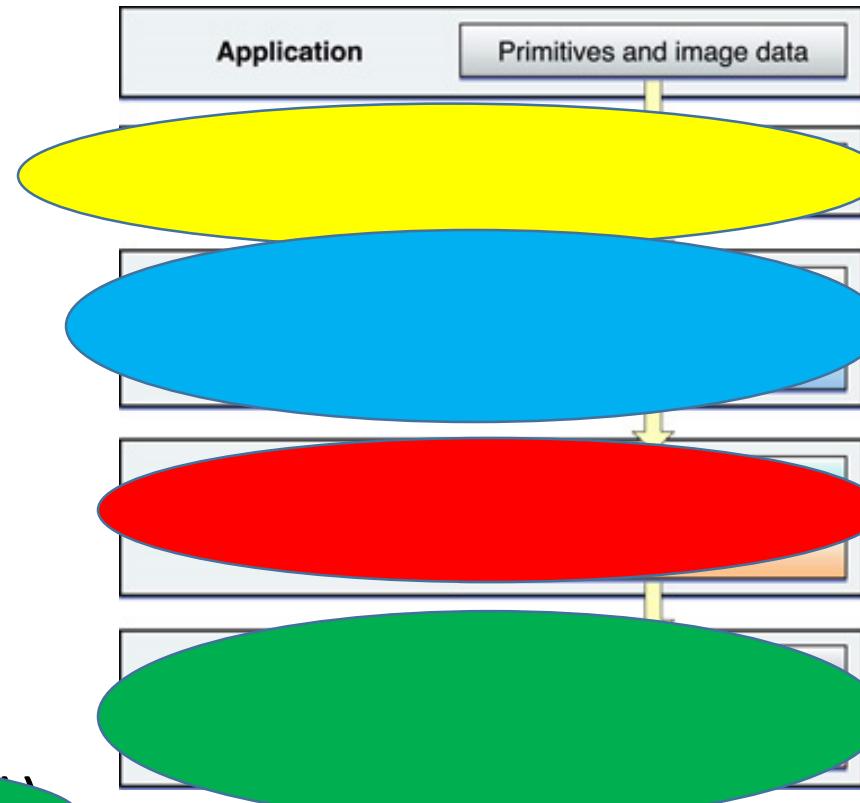


```
fragment[] frags = {};
```

```
foreach triangle t (v0, v1, v2)
```



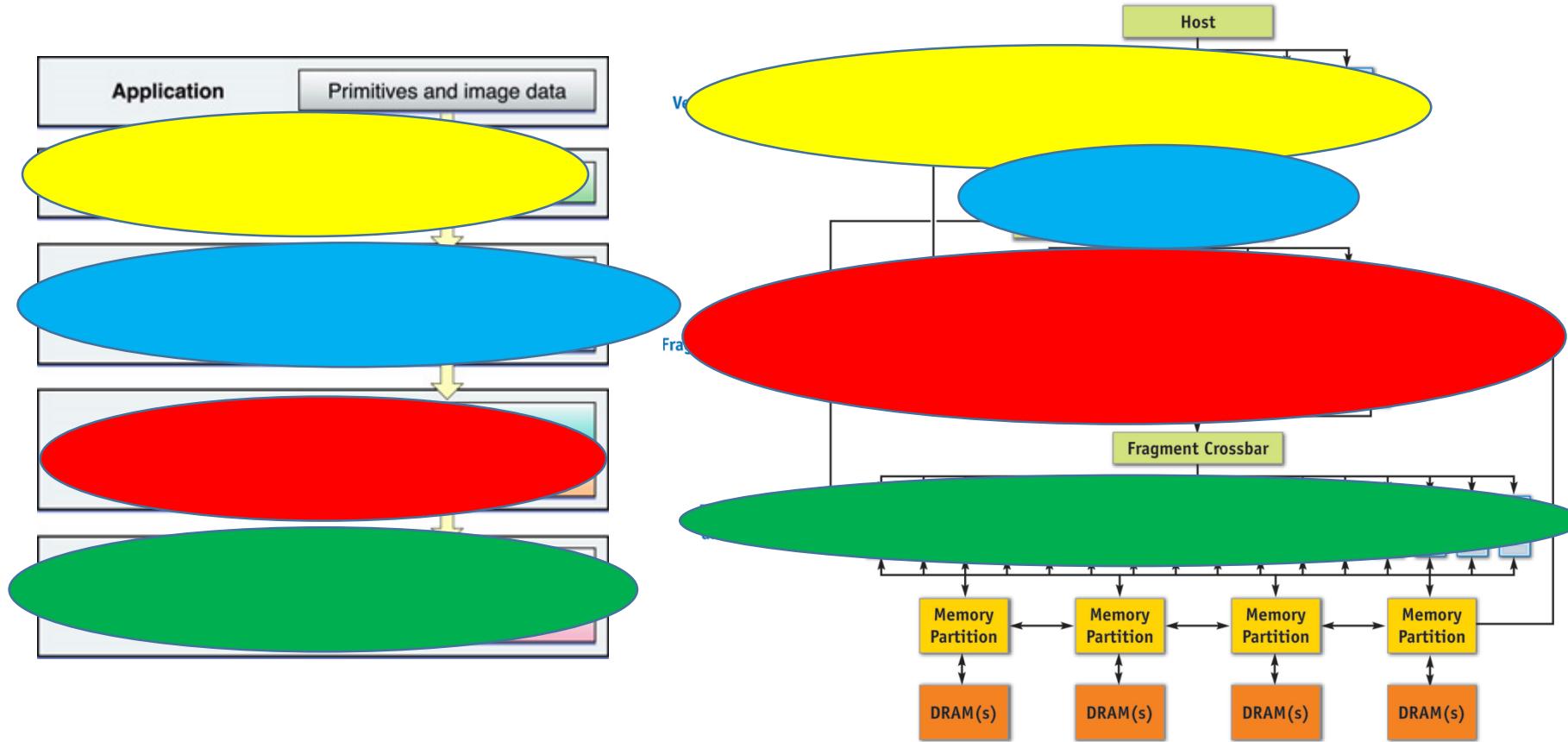
```
foreach fragment f in frags
```



OpenGL pipeline

To first order, DirectX looks the same!

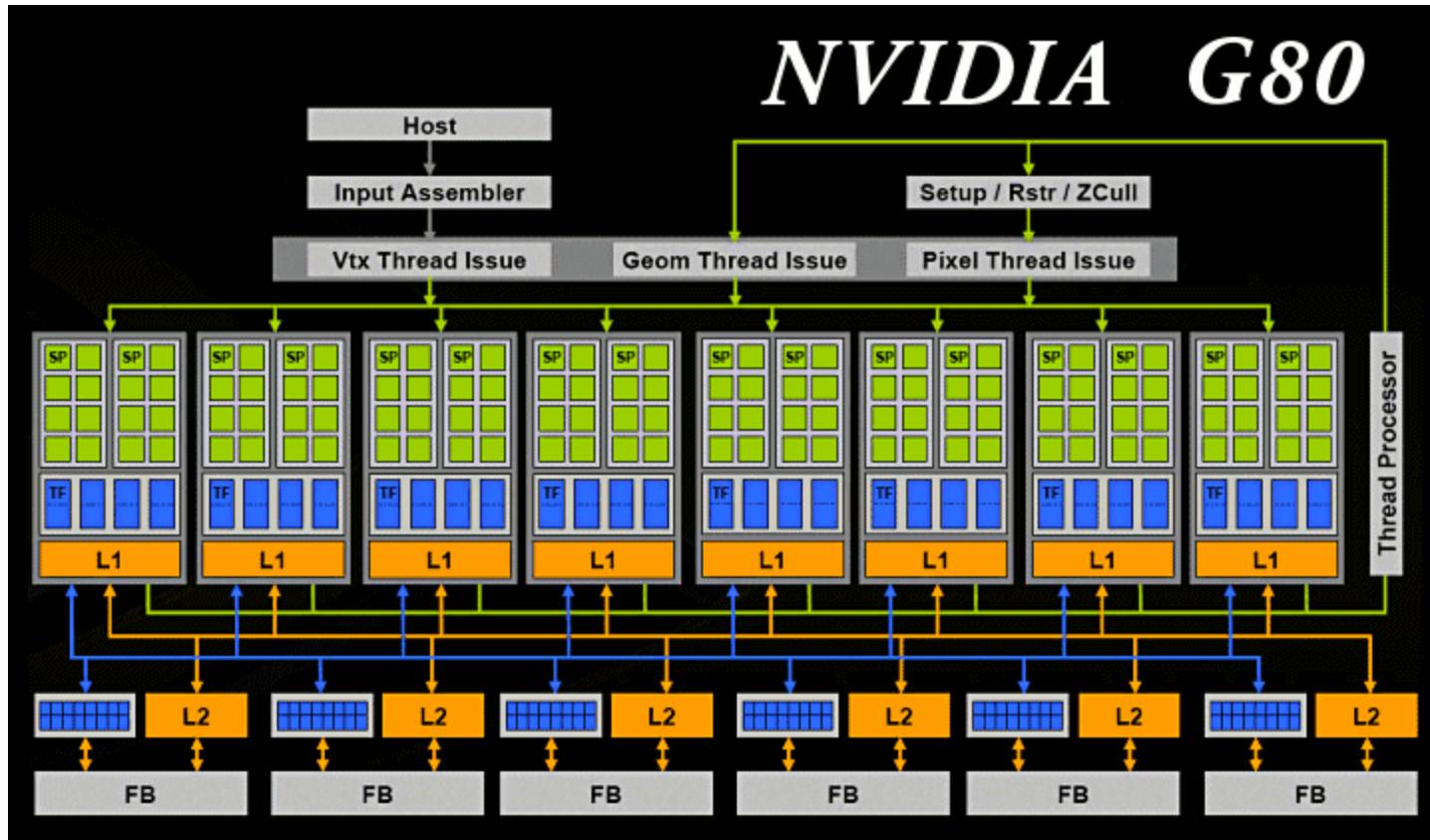
# Graphics pipeline → GPU architecture



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

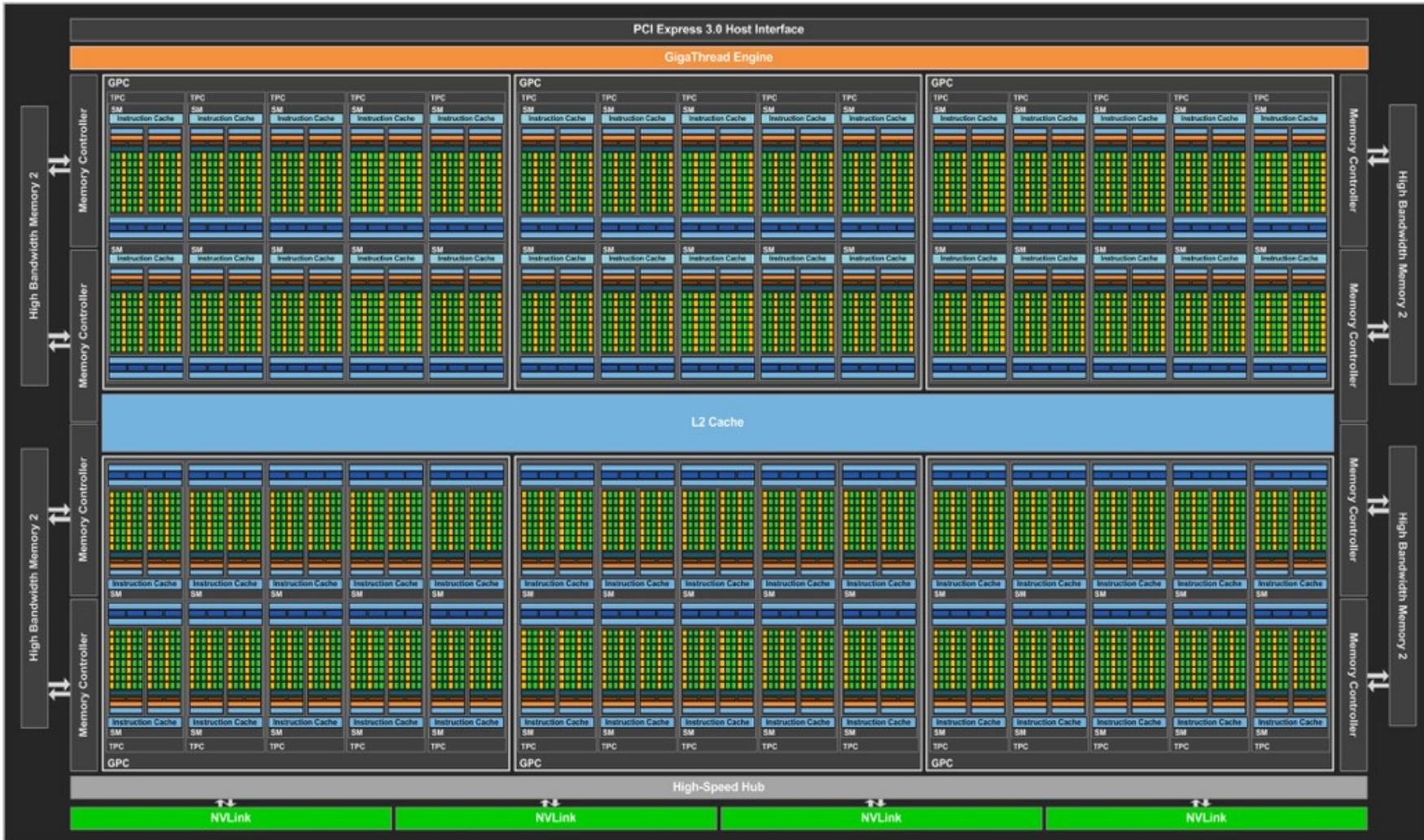
GeForce 6 series

# Late Modernity: unified shaders

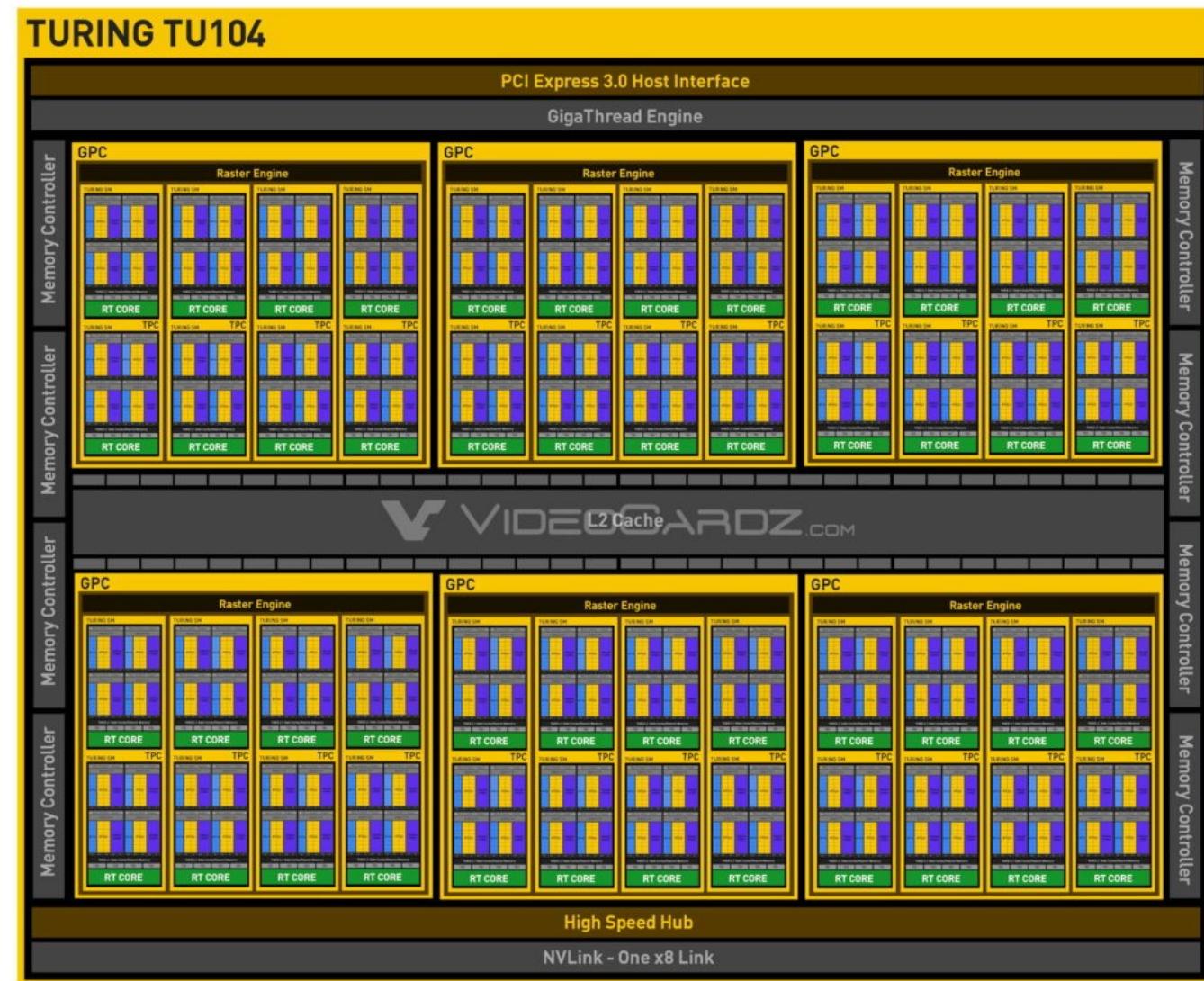
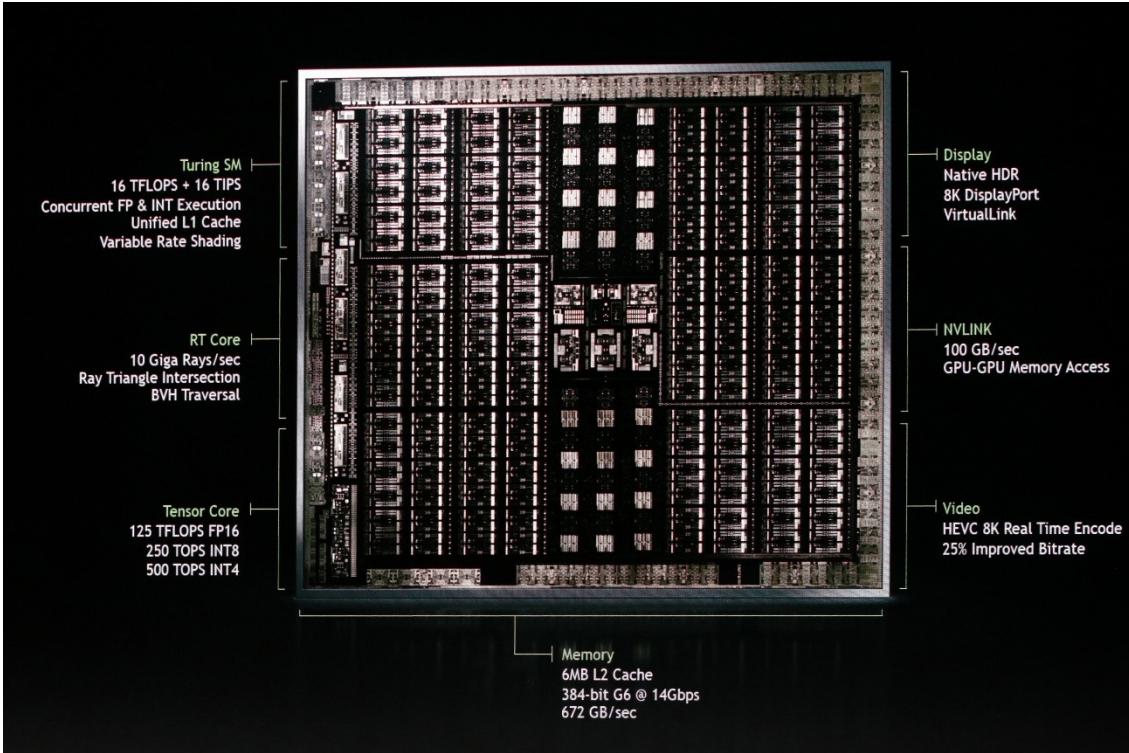


Mapping to Graphics pipeline no longer apparent  
Processing elements no longer specialized to a particular role  
Model supports *real* control flow, larger instr count

# Mostly Modern: Pascal



# Definitely Modern: Turing



# Cross-generational GPU observations

GPUs designed for parallelism in graphics pipeline:

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

- Simple cores
- Single instruction stream

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?!?!*

OR  
ing

# 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

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

# Map

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

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



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

Why is this useful on  
a box-drawing  
machine?

# Scatter and Gather

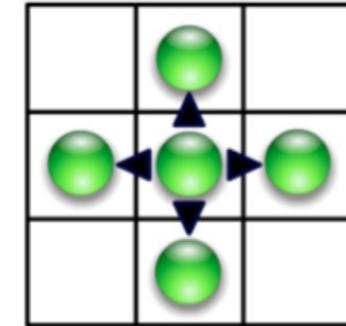
- Gather:
  - Read multiple items to single /packed location
- Scatter:
  - Write single/packed data item to multiple locations
- Inputs: x, y, indeces, N

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

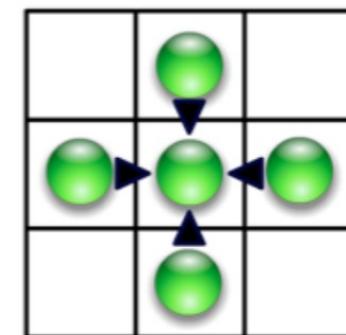
gather(x, y, idx)

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

scatter(x, y, idx)



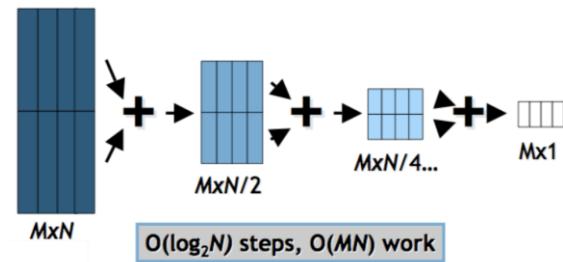
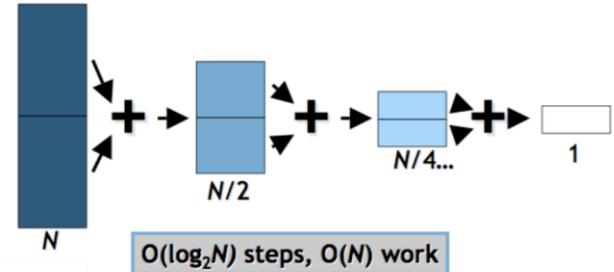
Scatter



Gather

# Reduce

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



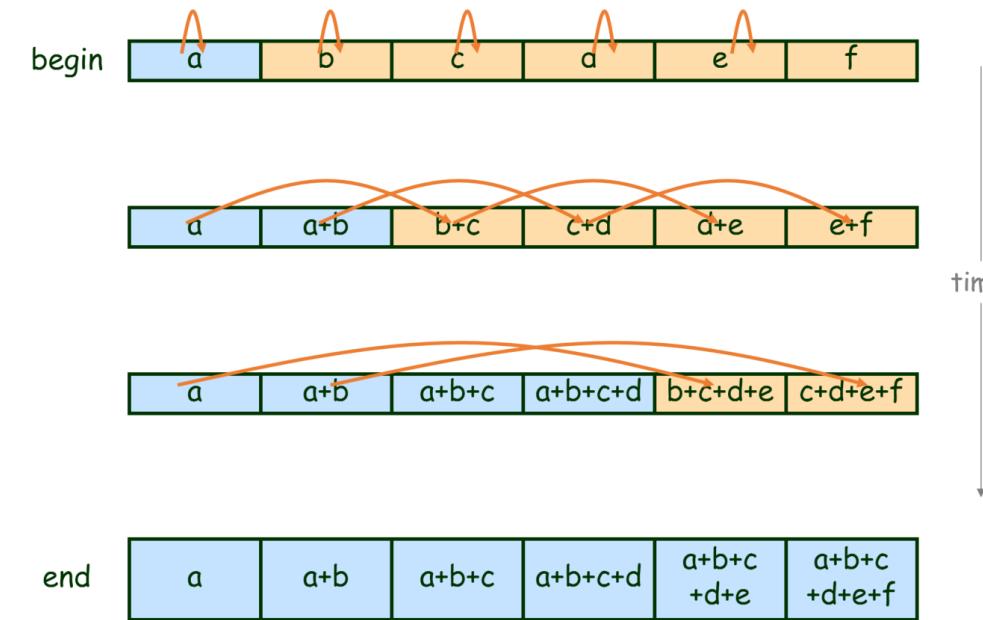
```
for(i=0; i<N; ++i) {  
    accum += (point[i]*point[i])  
}
```

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

Why must op be associative?

# Scan (prefix sum)

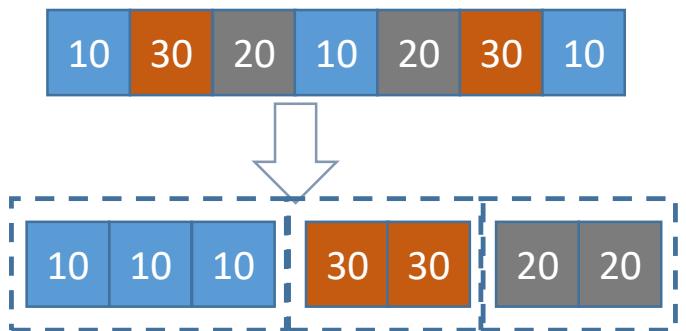
- Input
  - Associative operator **op**
  - Ordered set  $s = [a, b, c, \dots z]$
  - Identity  $I$
- $\text{scan}(\text{op}, s) = [I, a, (a \text{ op } b), (a \text{ op } b \text{ op } c) \dots]$
- Scan is the workhorse of parallel algorithms:
  - Sort, histograms, sparse matrix, string compare, ...



# GroupBy

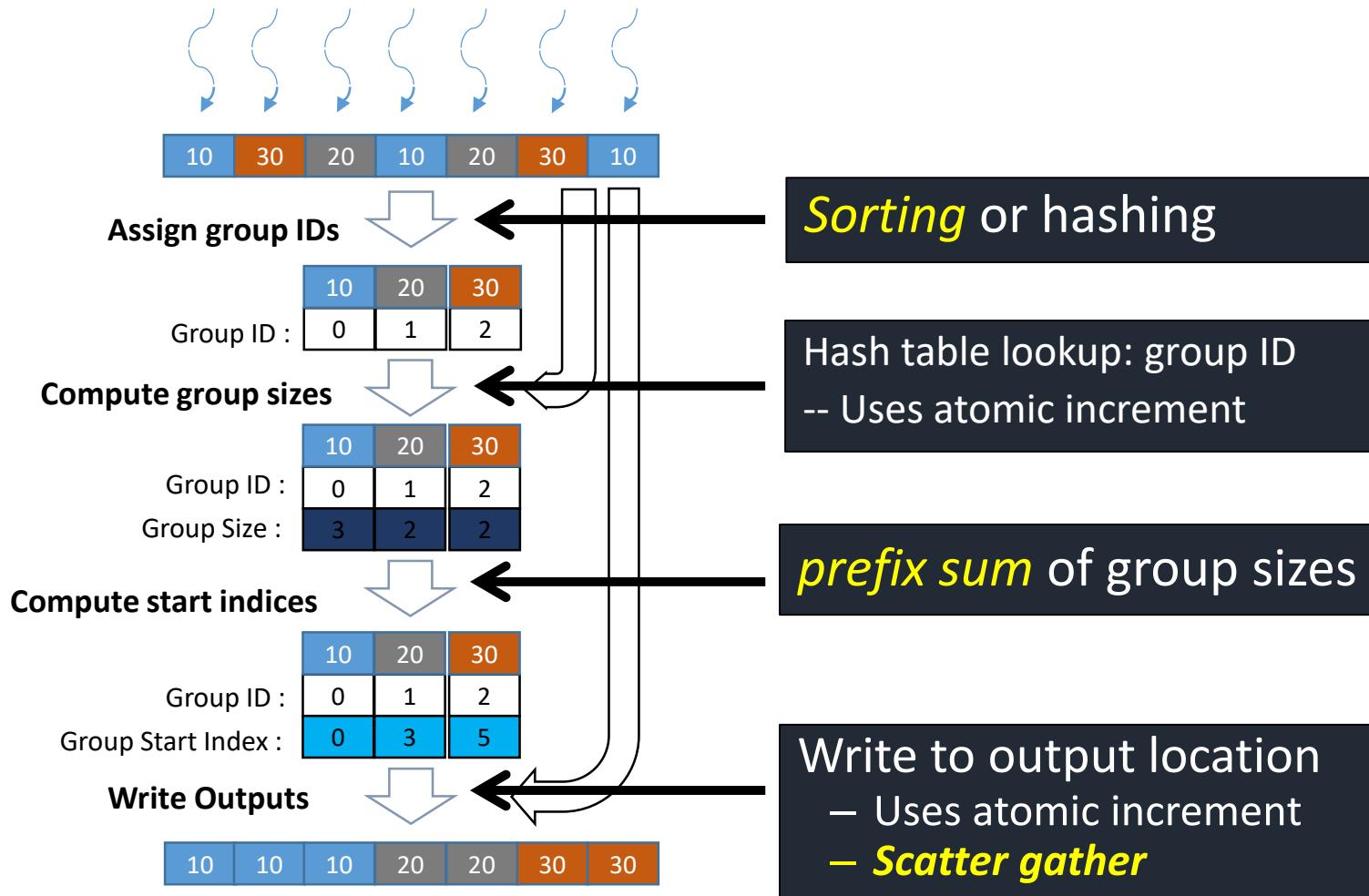
- Group a collection by key
- Lambda function maps elements → key

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



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

# GroupBy using parallel primitives



# Sort

- OK, let's build a parallel sort

# Summary

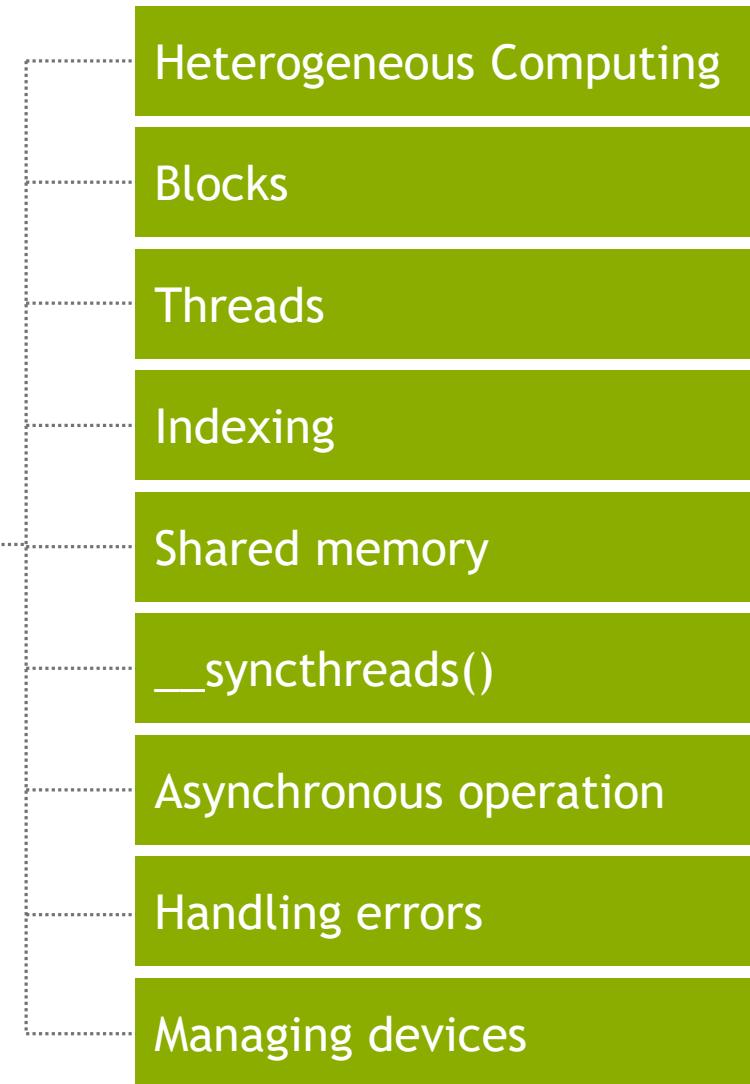
Re-expressing apparently sequential algorithms as combinations of parallel patterns is a common technique when targeting GPUs

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

# What is CUDA?

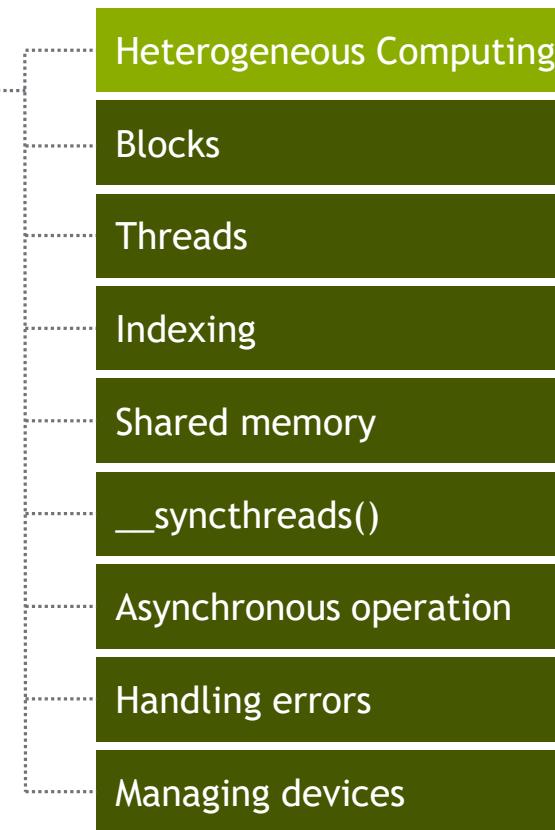
- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Retain performance
- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.

# CONCEPTS



# HELLO WORLD!

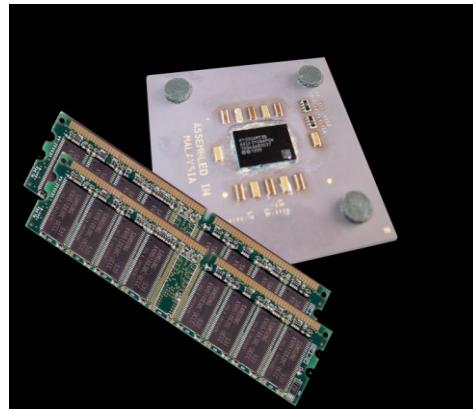
## CONCEPTS



# Heterogeneous Computing

- Terminology:

- *Host* The CPU and its memory (host memory)
- *Device* The GPU and its memory (device memory)



Host



Device

# Heterogeneous Computing

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockDim.x * blockIdx.x;
    int index = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[index] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[index - RADIUS] = in[gindex - RADIUS];
        temp[index + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[index + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out;           // host copies of a, b, c
    int *d_in, *d_out;       // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int *)malloc(size); fill_ints(in, N + 2*RADIUS);
    out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc((void **) & d_in, size);
    cudaMalloc((void **) & d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<(N/BLOCK_SIZE,BLOCK_SIZE>>(d_in + RADIUS,
d_out + RADIUS));

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

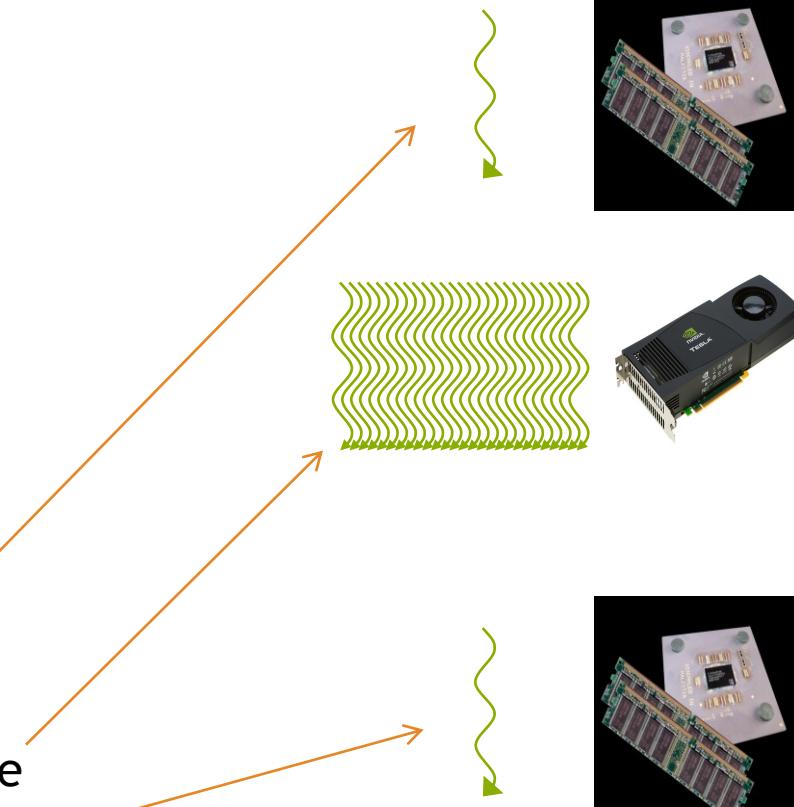
    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

parallel fn

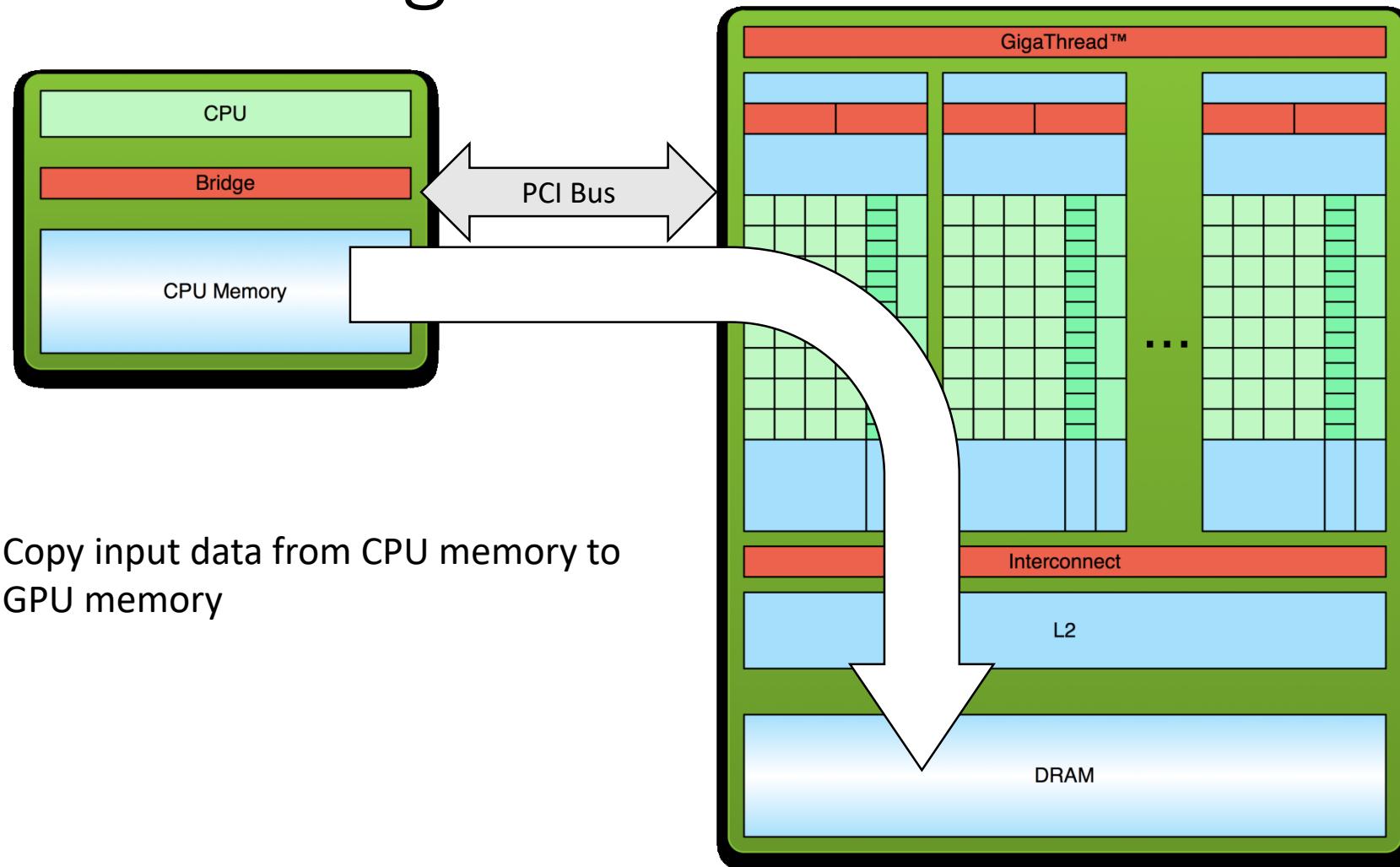
serial code

parallel code

serial code

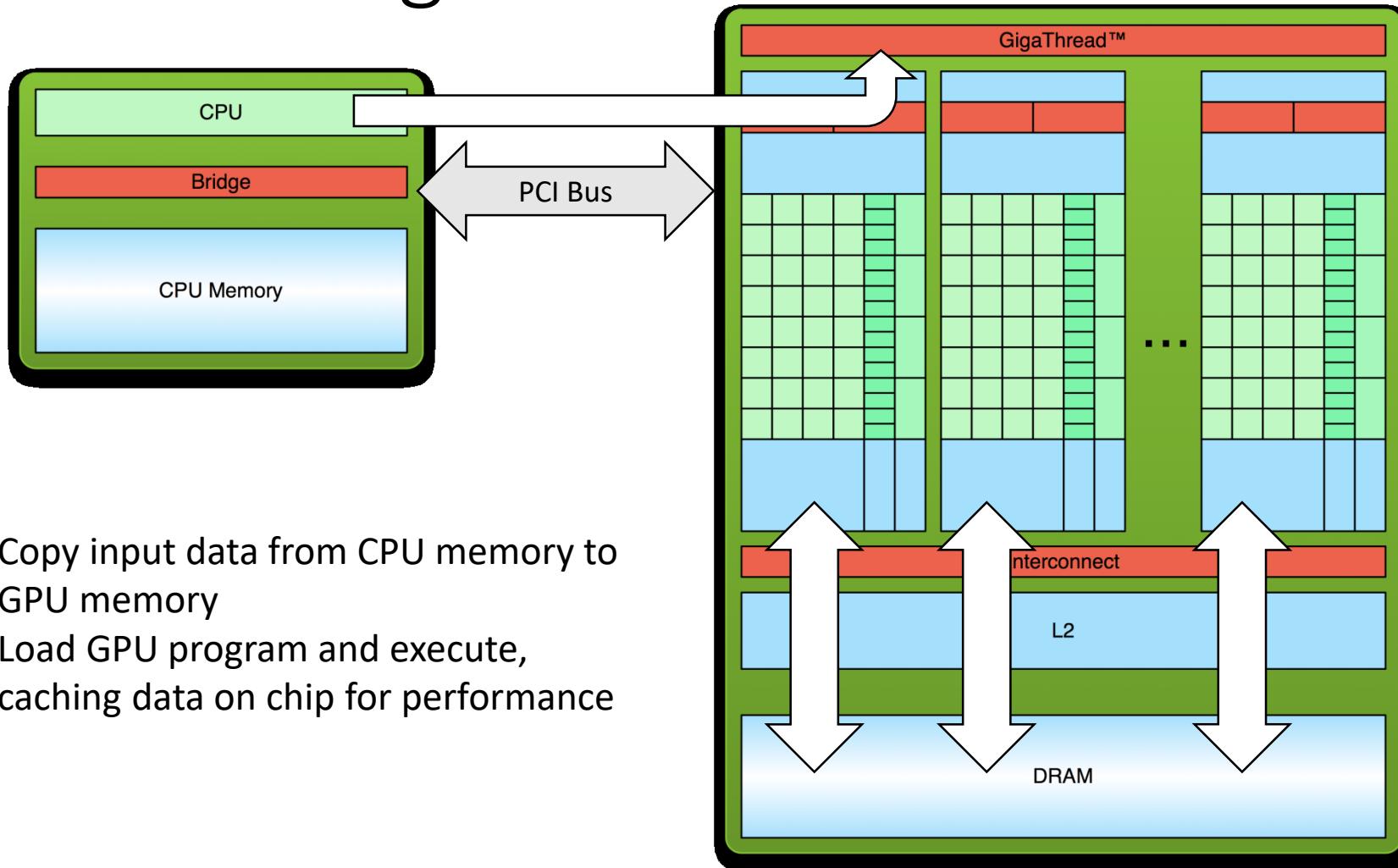


# Simple Processing Flow

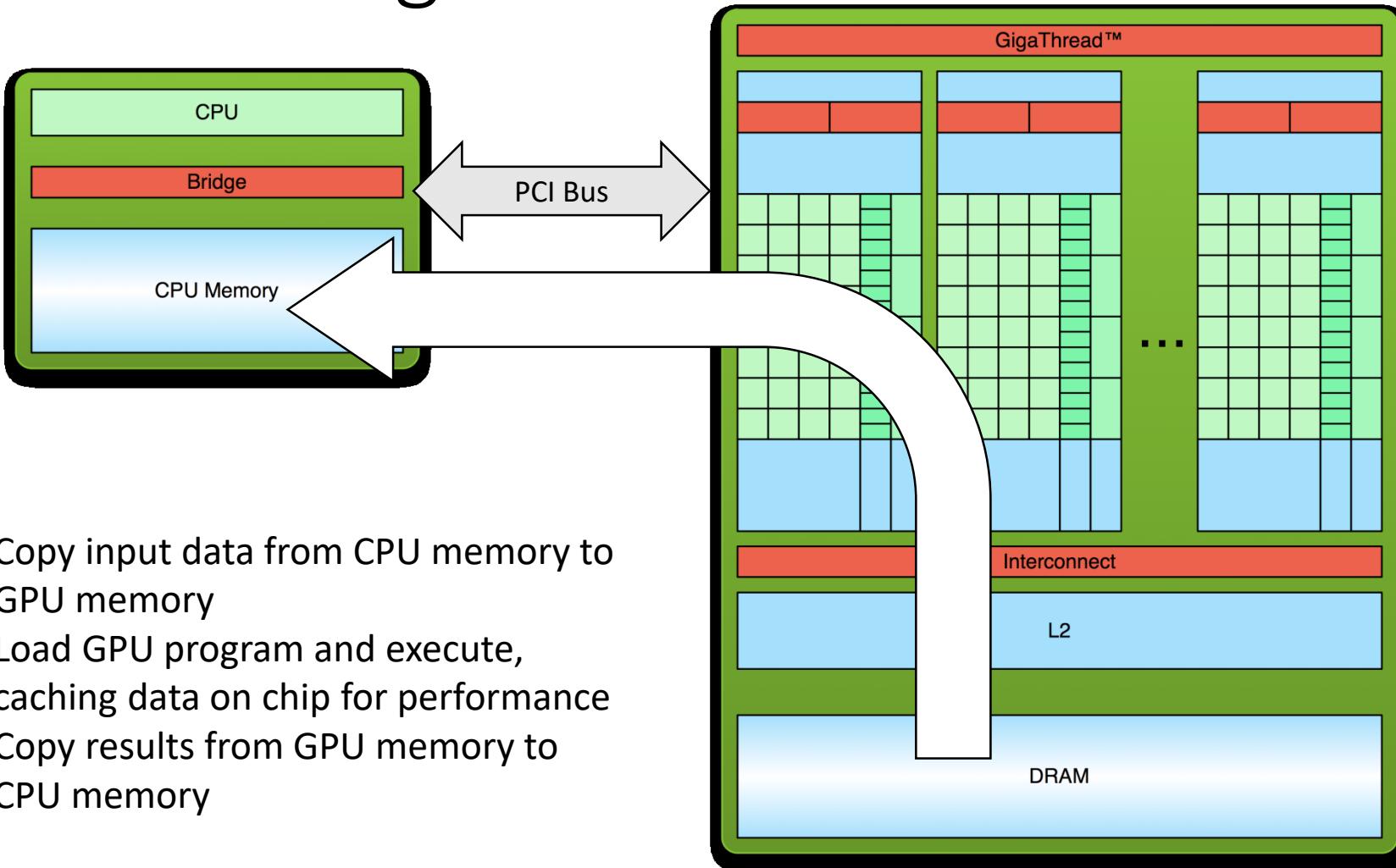


1. Copy input data from CPU memory to GPU memory

# Simple Processing Flow



# Simple Processing Flow



# Hello World!

```
int main(void) {  
    printf("Hello World!\n");  
    return 0;  
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

## Output:

```
$ nvcc  
hello_world.  
cu  
$ a.out  
Hello World!  
$
```

# Hello World! with Device Code

```
__global__ void mykernel(void) {  
}  
  
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

- Two new syntactic elements...

# Hello World! with Device Code

```
__global__ void mykernel(void) {  
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code
- nvcc separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - `gcc, cl.exe`

# Hello World! with Device COde

```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment
- That’s all that is required to execute a function on the GPU!

# Hello World! with Device Code

```
__global__ void mykernel(void) {  
}
```

```
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

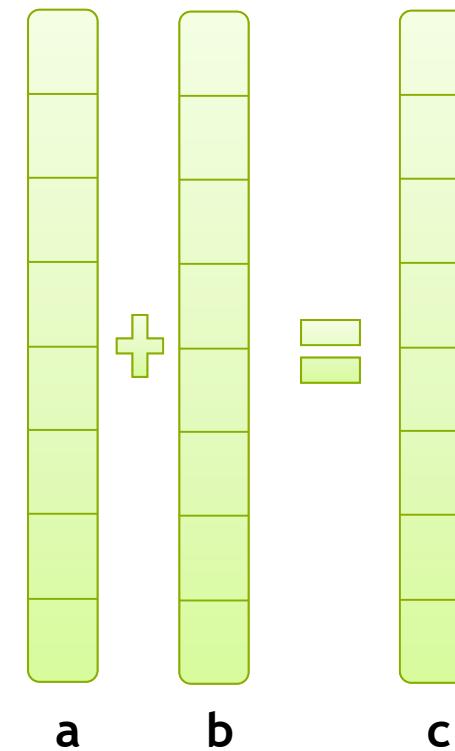
- **mykernel()** does nothing,  
somewhat anticlimactic!

## Output:

```
$ nvcc  
hello.cu  
$ a.out  
Hello World!  
$
```

# Parallel Programming in CUDA C/C++

- But wait... GPU computing is about massive parallelism!
- We need a more interesting example...
- We'll start by adding two integers and build up to vector addition



# Addition on the Device

- A simple kernel to add two integers

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- As before `__global__` is a CUDA C/C++ keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host

# Addition on the Device

- Note that we use pointers for the variables

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory
- We need to allocate memory on the GPU

# Memory Management

- Host and device memory are separate entities

- *Device* pointers point to GPU memory

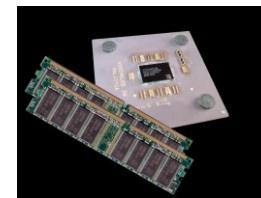
- May be passed to/from host code

- May *not* be dereferenced in host code

- *Host* pointers point to CPU memory

- May be passed to/from device code

- May *not* be dereferenced in device code



- Simple CUDA API for handling device memory

- `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`

- Similar to the C equivalents `malloc()`, `free()`, `memcpy()`

# Addition on the Device: add()

- Returning to our add() kernel

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- Let's take a look at main()...

# Addition on the Device: main()

```
int main(void) {
    int a, b, c;                      // host copies of a, b, c
    int *d_a, *d_b, *d_c;              // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Setup input values
    a = 2;
    b = 7;
```

# Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

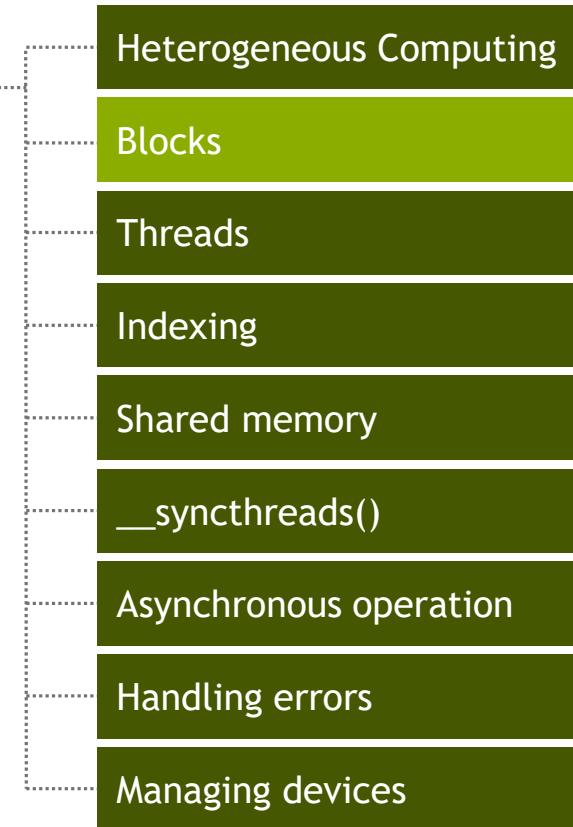
// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# RUNNING IN PARALLEL

## CONCEPTS



# Moving to Parallel

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>();  
      ↓  
add<<< N, 1 >>>();
```

- Instead of executing add () once, execute N times in parallel

# Vector Addition on the Device

- With `add()` running in parallel we can do vector addition
- Terminology: each parallel invocation of `add()` is a **block**
  - The set of blocks is referred to as a **grid**
  - Each invocation can refer to its block index using `blockIdx.x`

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- By using `blockIdx.x` to index into the array, each block handles a different index

# Vector Addition on the Device

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- On the device, each block can execute in parallel:

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

```
c[3] = a[3] + b[3];
```

# Vector Addition on the Device: add()

- Returning to our parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- Let's take a look at `main()`...

# Vector Addition on the Device: main()

```
#define N 512

int main(void) {
    int *a  *b  *c          // host copies of a, b, c
    int *d_a, *d_b, *d_c;    // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

# Vector Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

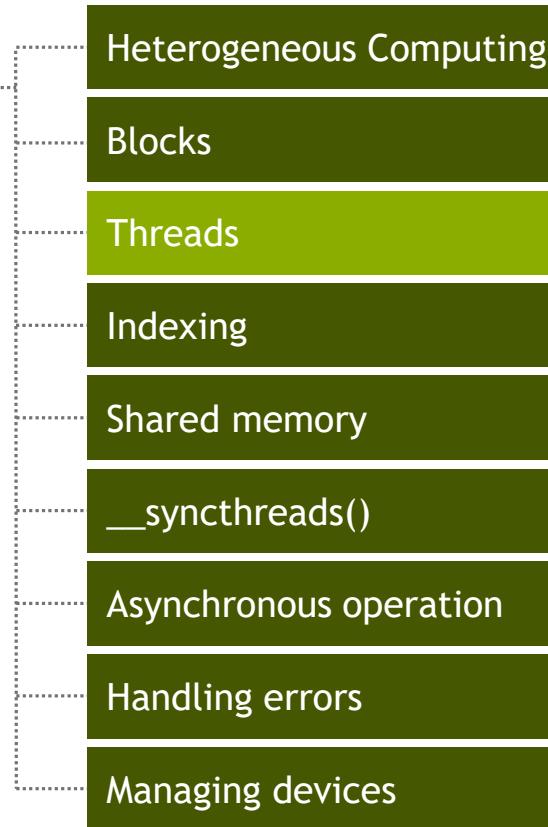
// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# Review

- Difference between *host* and *device*
  - *Host* CPU
  - *Device* GPU
- `__global__` declares device code
  - Executes on the device
  - Called from the host
- Passing parameters from host code to a device function
- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`
- Launching parallel kernels
  - Launch `N` copies of `add()` with `add<<<N, 1>>>(...);`
  - Use `blockIdx.x` to access block index

# INTRODUCING THREADS

## CONCEPTS



# CUDA Threads

- Terminology: a block can be split into parallel *threads*
- Change `add()` to use parallel *threads* instead of parallel *blocks*:

```
__global__ void add(int *a, int *b, int *c) {  
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];  
}
```

- Use `threadIdx.x` instead of `blockIdx.x`
- Need to make one change in `main()` ...

# Vector Addition Using Threads: main()

```
#define N 512

int main(void) {
    int *a, *b, *c;                                // host copies of a, b, c
    int *d_a, *d_b, *d_c;                            // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

# Vector Addition Using Threads: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

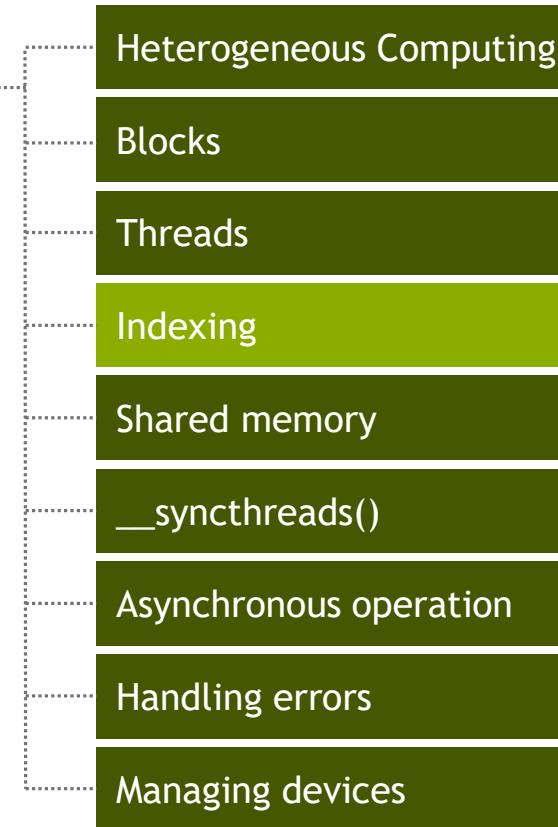
// Launch add() kernel on GPU with N threads
add<<<1,N>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# COMBINING THREADS AND BLOCKS

## CONCEPTS

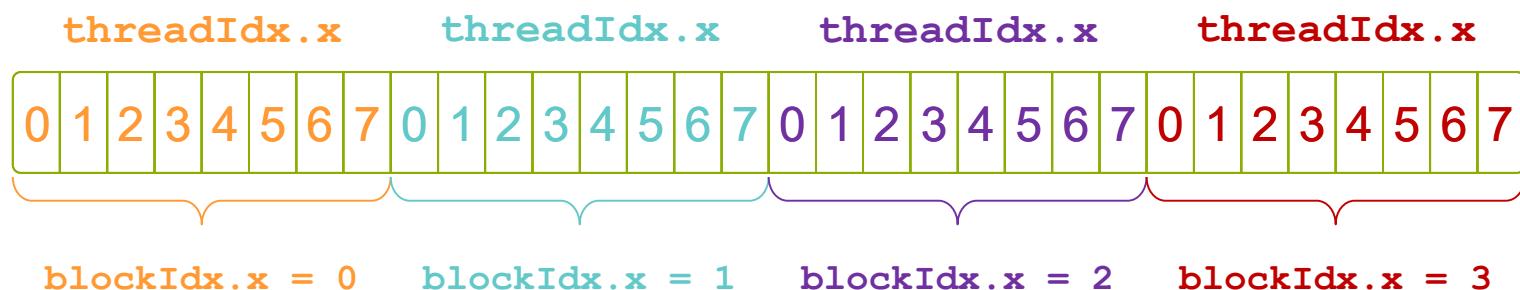


# Combining Blocks and Threads

- We've seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads
- Let's adapt vector addition to use both blocks and threads
- Why? We'll come to that...
- First let's discuss data indexing...

# Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Index an array with one elem. per thread (8 threads/block)

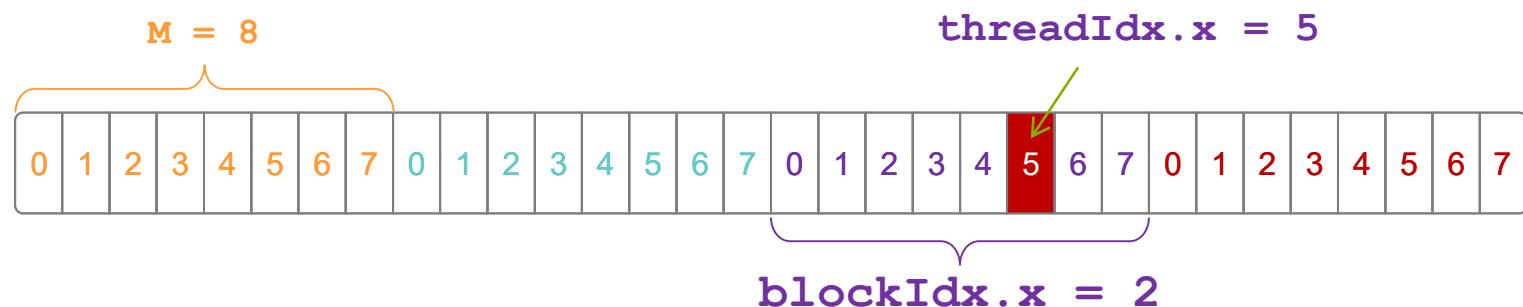


- With M threads/block, unique index per thread is :

```
int index = threadIdx.x + blockIdx.x * M;
```

# Indexing Arrays: Example

- Which thread will operate on the red element?



```
int index = threadIdx.x + blockIdx.x * M;  
= 5 + 2 * 8;  
= 21;
```

# Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

- Combined add () using parallel threads *and* blocks

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

- What changes need to be made in `main()`?

# Addition with Blocks and Threads:

```
main()
```

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;                                // host copies of a, b, c
    int *d_a, *d_b, *d_c;                            // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

# Addition with Blocks and Threads:

```
main()
```

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`
- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

- Update the kernel launch:

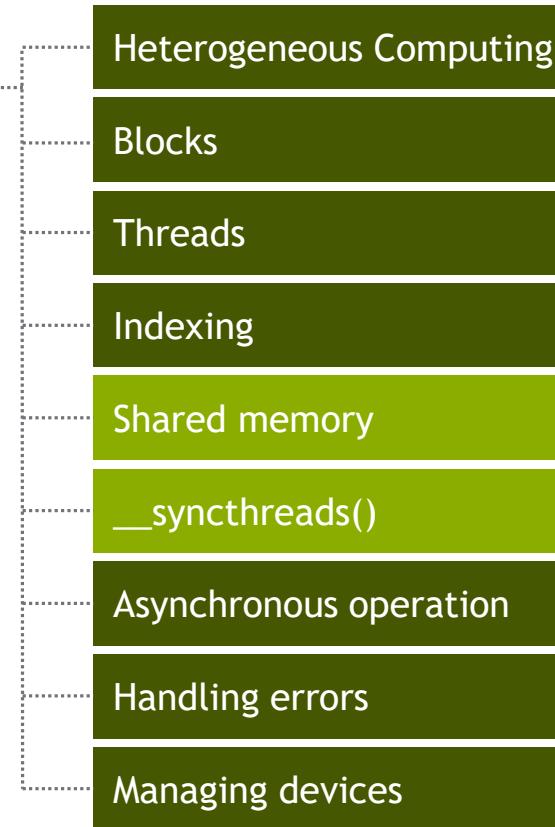
```
add<<<(N + M-1) / M, M>>>(d_a, d_b, d_c, N);
```

# Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize
- To look closer, we need a new example...

# COOPERATING THREADS

## CONCEPTS



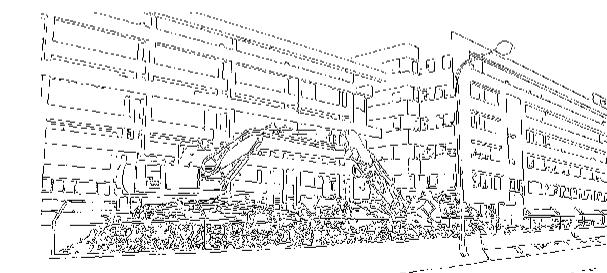
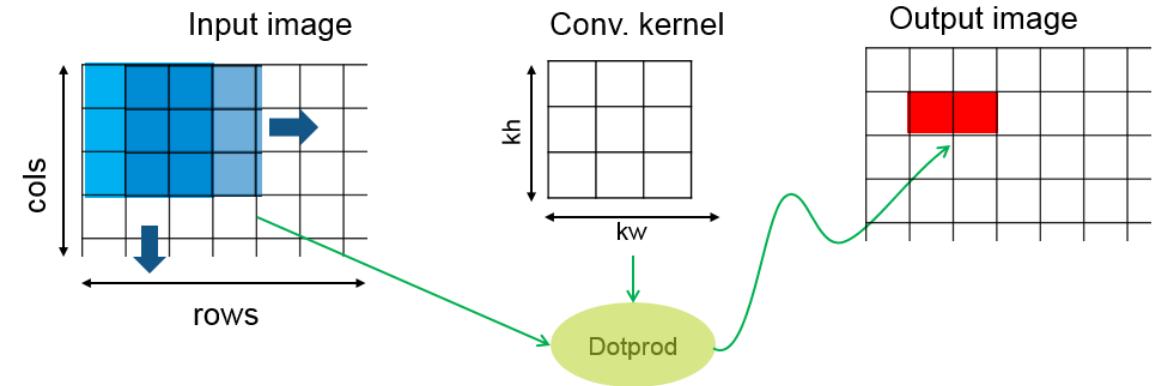
# Stencils

- Each pixel → function of neighbors
- Edge detection:

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

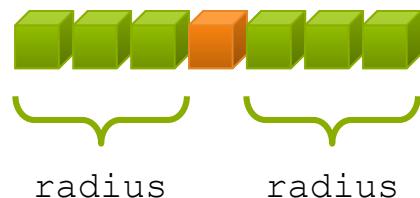
- Blur:

1/16	1/8	1/16
1/8	1/4	1/8
1/16	1/8	1/16



# 1D Stencil

- Consider 1D stencil over 1D array of elements
  - Each output element is the sum of input elements within a radius
- Radius == 3 → each output element is sum of 7 input elements:



# Implementation within a block

- Each thread: process 1 output element
  - blockDim.x elements per block
- Input elements read many times
  - With radius 3, each input element is read seven times

```
__global__ void stencil_1d(int *in, int *out) {  
    // note: idx comp & edge conditions omitted...  
    int result = 0;  
    for (int offset = -R; offset <= R; offset++)  
        result += in[idx + offset];  
  
    // Store the result  
    out[idx] = result;  
}
```

# Implementation within a block

- Each thread: process 1 output element
  - blockDim.x elements per block
- Input elements read many times
  - With radius 3, each input element is read seven times

```
_global_ void stencil_1d(int *in, int *out) {  
    // note: idx comp & edge conditions omitted...  
    int result = 0;  
    for (int offset = -R; offset <= R; offset++)  
        result += in[idx + offset];  
  
    // Store the result  
    out[idx] = result;  
}
```



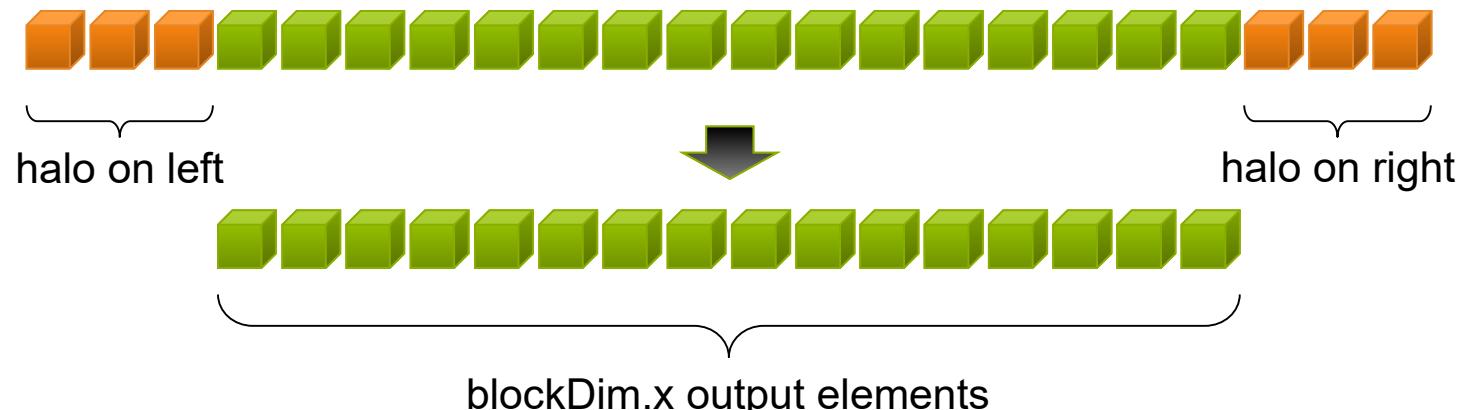
Why is this a problem?

# Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block
- Data is *not visible* to threads in other blocks

# Stencil with Shared Memory

- Cache data in shared memory
  - Read  $(blockDim.x + 2 * radius)$  elements from memory to shared
  - Compute  $blockDim.x$  output elements
  - Write  $blockDim.x$  output elements to global memory
  - Each block needs a **halo** of  $radius$  elements at each boundary



# Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] =
            in[gindex + BLOCK_SIZE];
    }

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

Are we done?

# Data Race!

- The stencil example will not work...
- Suppose thread 15 reads the halo before thread 0 has fetched it...

```
temp[lindex] = in[gindex];           Store at temp[18] ████  
if (threadIdx.x < RADIUS) {  
    temp[lindex - RADIUS] = in[gindex - RADIUS];   Skipped, threadIdx > RADIUS  
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];  
}  
  
int result = 0;  
result += temp[lindex + 1];           Load from temp[19] ████
```

# \_\_syncthreads()

- `void __syncthreads();`
- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

# Correct Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] =
            in[gindex + BLOCK_SIZE];
    }
    __syncthreads();
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];
    __syncthreads();
    // Store the result
    out[gindex] = result;
}
```



# Notes on \_\_syncthreads()

- void \_\_syncthreads();

- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards

```
__global__ void some_kernel(int *in, int *out) {  
    // good idea?  
    if(threadIdx.x == SOME_VALUE)  
        __syncthreads();  
}
```

- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

```
__device__ void lock_trick(int *in, int *out) {  
    __syncthreads();  
    if(myIndex == 0)  
        critical_section();  
    __syncthreads();  
}
```

# Atomics

## Race conditions –

- Traditional locks are to be avoided
- How do we synchronize?

## Read-Modify-Write uninterruptible – atomic

atomicAdd ()	atomicInc ()
atomicSub ()	atomicDec ()
atomicMin ()	atomicExch ()
atomicMax ()	atomicCAS ()

# Recap

- Launching parallel threads
  - Launch  $N$  blocks with  $M$  threads per block with `kernel<<<N,M>>>(...);`
  - Use `blockIdx.x` to access block index within grid
  - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x
```

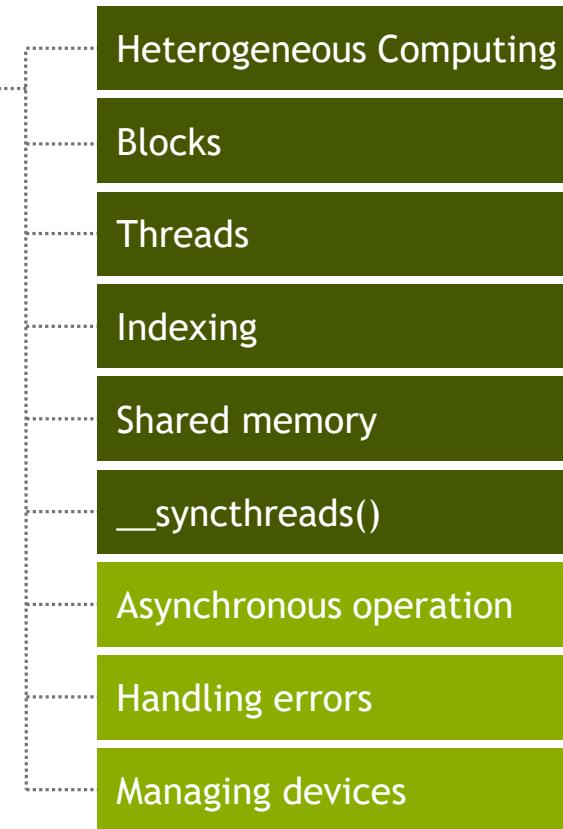
Use `__shared__` to declare a variable/array in shared memory

Data is shared between threads in a block  
Not visible to threads in other blocks

Use `__syncthreads()` as a barrier  
Use to prevent data hazards

# MANAGING THE DEVICE

## CONCEPTS



# Coordinating Host & Device

- Kernel launches are **asynchronous**
  - Control returns to the CPU immediately
- CPU needs to synchronize before consuming the results

<b>cudaMemcpy()</b>	Blocks the CPU until the copy is complete Copy begins when all preceding CUDA calls have completed
<b>cudaMemcpyAsync()</b>	Asynchronous, does not block the CPU
<b>cudaDeviceSynchronize()</b>	Blocks the CPU until all preceding CUDA calls have completed

# Reporting Errors

- All CUDA API calls return an error code (`cudaError_t`)
  - Error in the API call itself
    - OR
  - Error in an earlier asynchronous operation (e.g. kernel)
- Get the error code for the last error:  
`cudaError_t cudaGetLastError(void)`
- Get a string to describe the error:  
`char *cudaGetString(cudaError_t)`  
  
`printf("%s\n", cudaGetString(cudaGetLastError()));`

# Device Management

- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)
cudaSetDevice(int device)
cudaGetDevice(int *device)
cudaGetDeviceProperties(cudaDeviceProp *prop, int
device)
```

- Multiple threads can share a device
- A single thread can manage multiple devices

```
cudaSetDevice(i) to select current device
cudaMemcpy(...) for peer-to-peer copies†
```

<sup>†</sup> requires OS and device support

# Questions?