Multithreaded Architectures and Programming on Multithreaded Architectures (II)

Hung-Wei Tseng

Recap: What about “linked list”

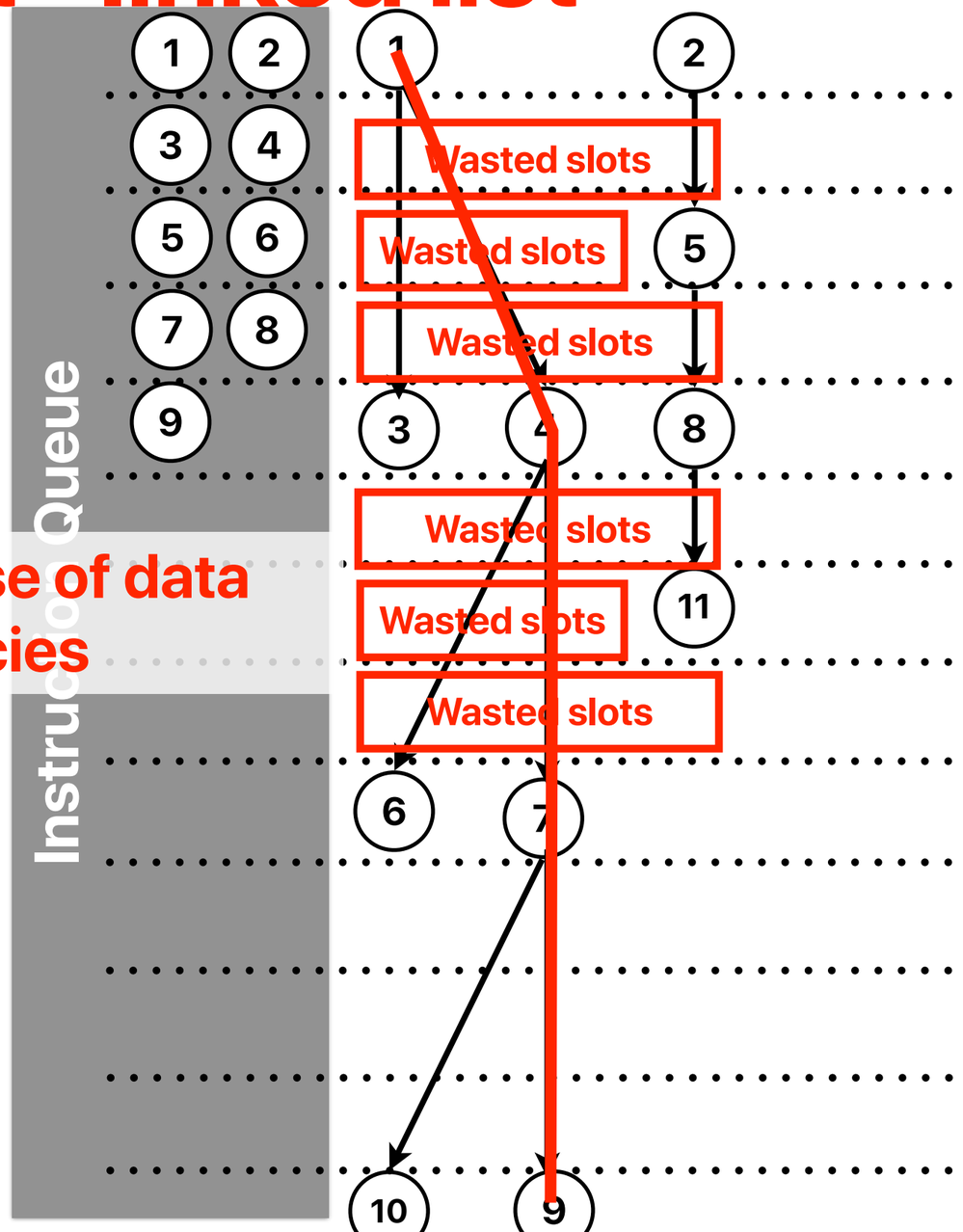
Static instructions

```
LOOP: ld   X10, 8(X10)
      addi  X7, X7, 1
      bne  X10, X0, LOOP
```

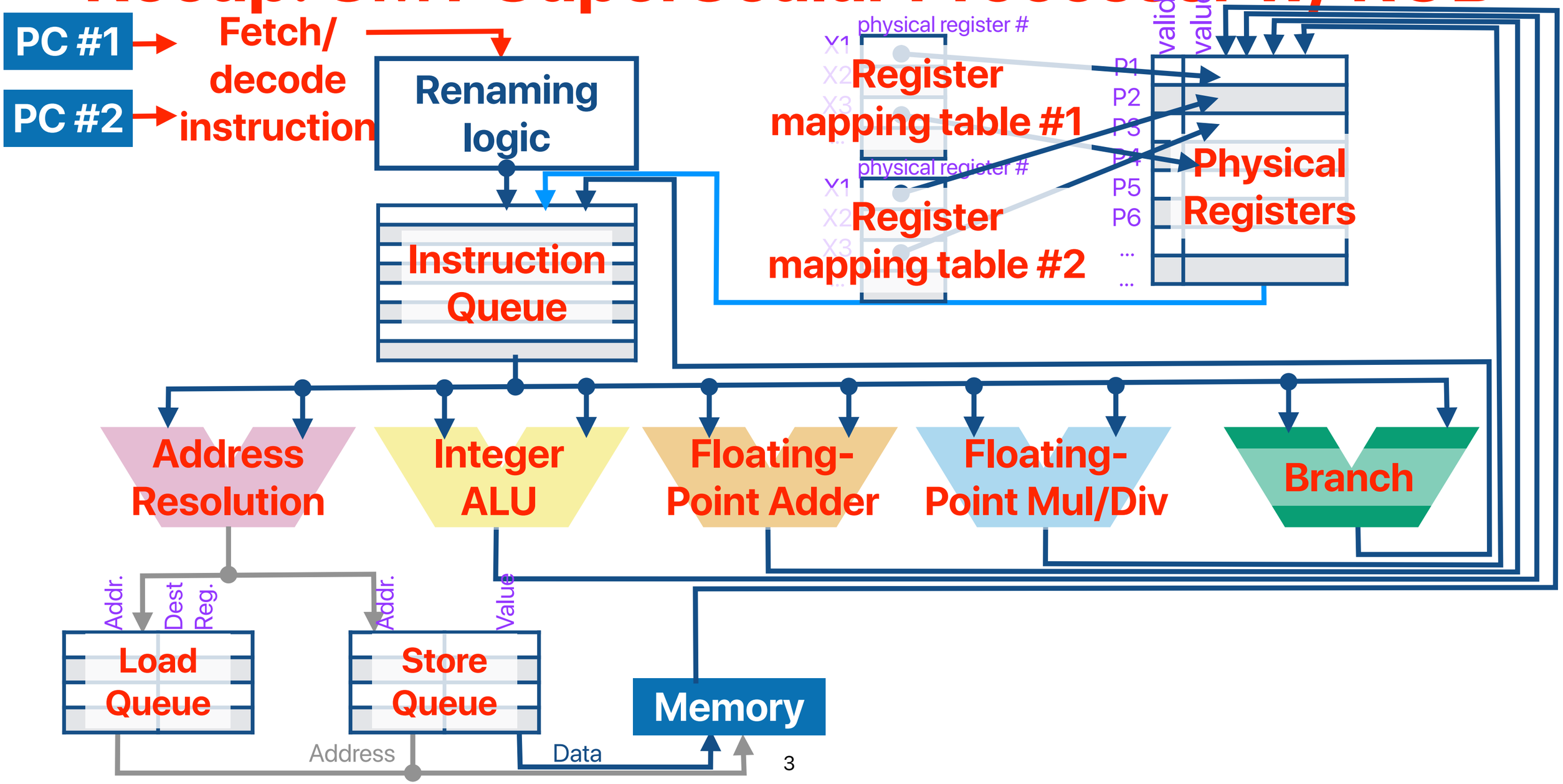
Dynamic instructions

```
① ld   X10, 8(X10)
② addi  X7, X7, 1
③ bne  X10, X0, LOOP
④ ld   X10, 8(X10)
⑤ addi  X7, X7, 1
⑥ bne  X10, X0, LOOP
⑦ ld   X10, 8(X10)
⑧ addi  X7, X7, 1
⑨ bne  X10, X0, LOOP
```

ILP is low because of data dependencies

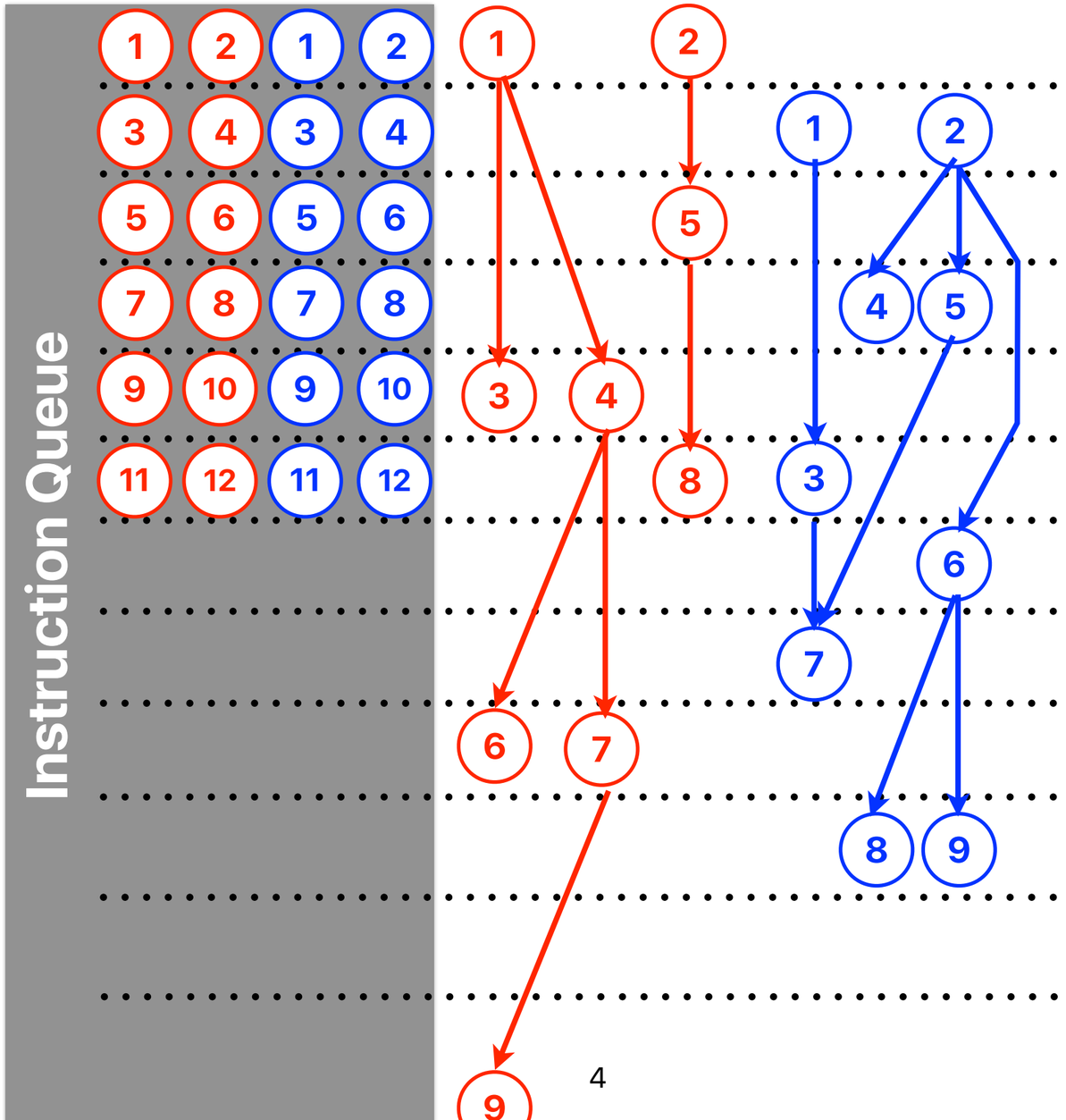


Recap: SMT SuperScalar Processor w/ ROB



Recap: Simultaneous multithreading

```
① ld   X10, 8(X10)
② addi  X7, X7, 1
③ bne  X10, X0, LOOP
④ ld   X10, 8(X10)
⑤ addi  X7, X7, 1
⑥ bne  X10, X0, LOOP
⑦ ld   X10, 8(X10)
⑧ addi  X7, X7, 1
⑨ bne  X10, X0, LOOP
```

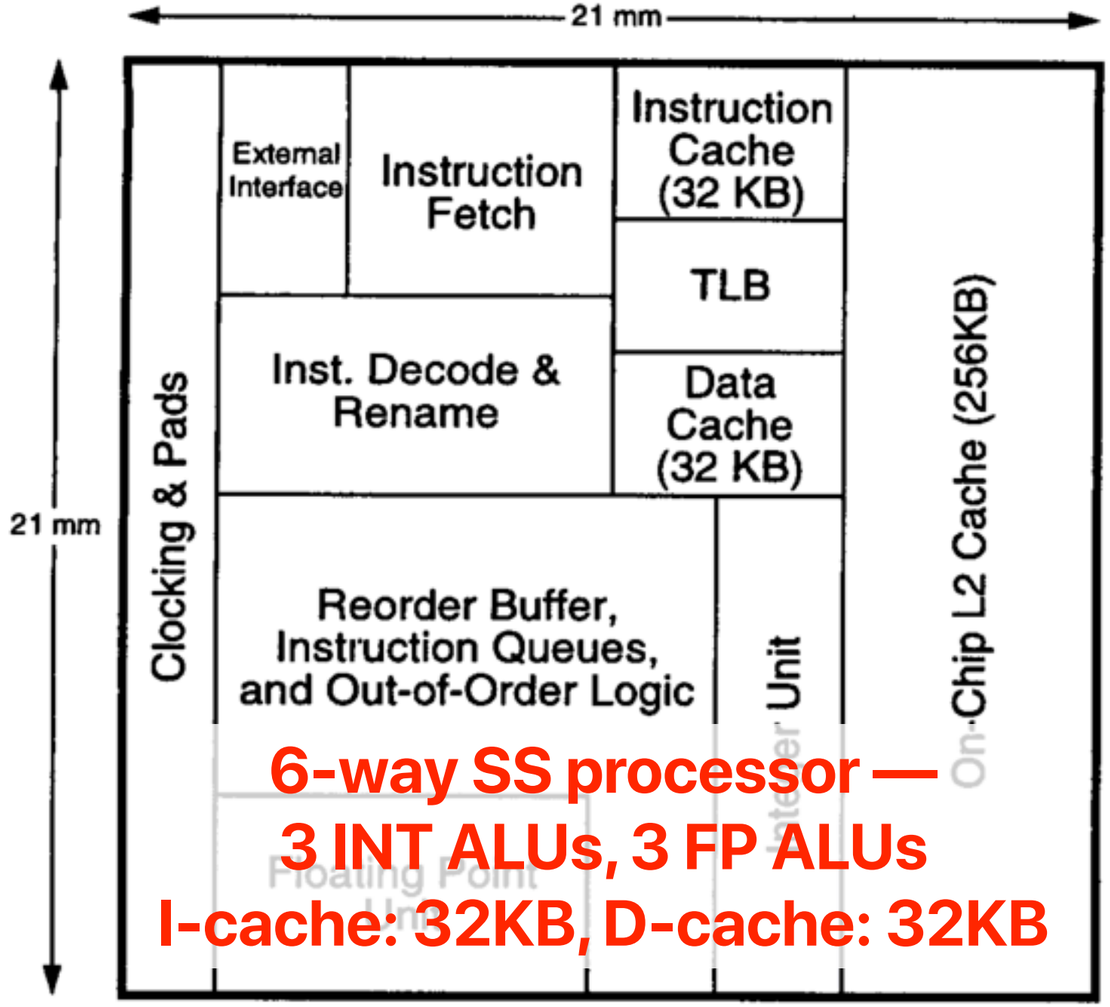



```
①  ld   X1, 0(X10)
②  addi X10, X10, 8
③  add  X20, X20, X1
④  bne  X10, X2, LOOP
⑤  ld   X1, 0(X10)
⑥  addi X10, X10, 8
⑦  add  X20, X20, X1
⑧  bne  X10, X2, LOOP
⑨  ld   X1, 0(X10)
⑩  addi X10, X10, 8
⑪  add  X20, X20, X1
⑫  bne  X10, X2, LOOP
```

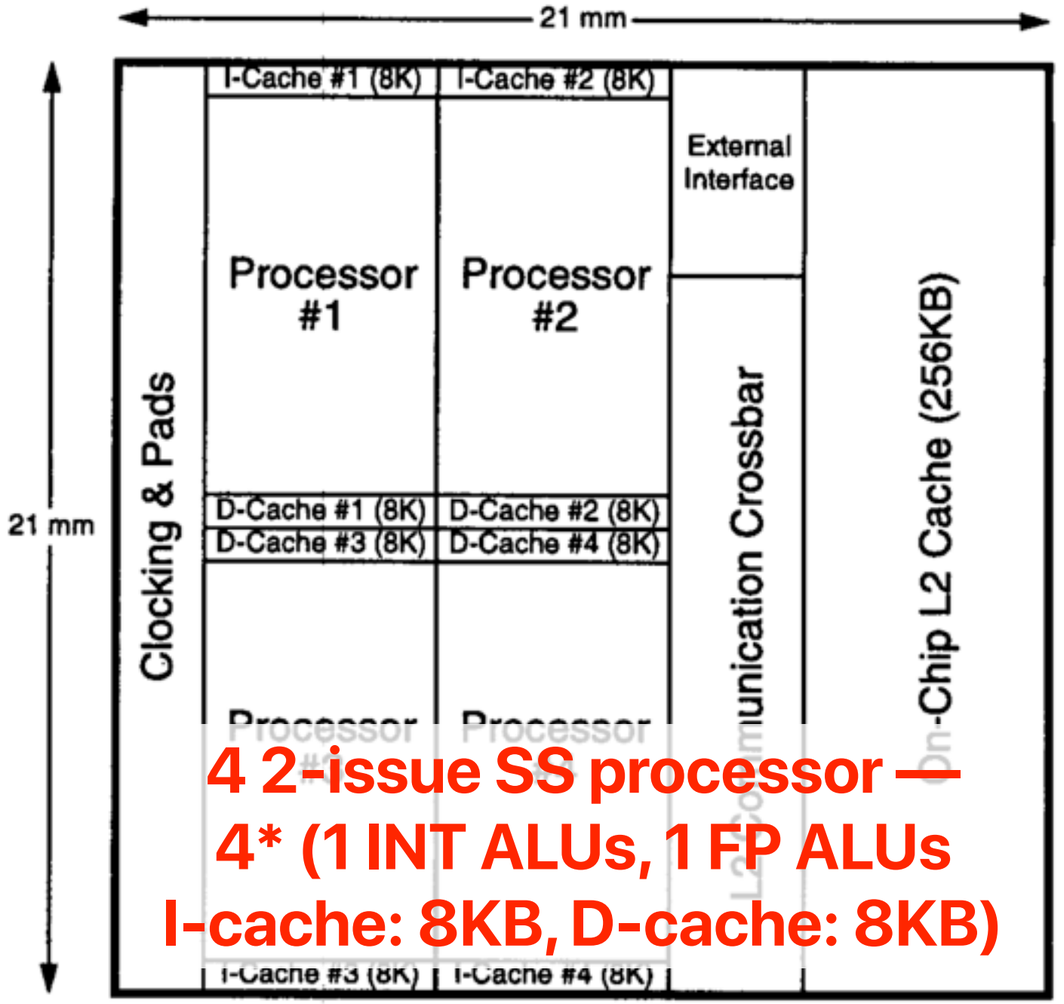
Recap: SMT

- Improve the throughput of execution
  - May increase the latency of a single thread
- Less branch penalty per thread
- Increase hardware utilization
- Simple hardware design: Only need to duplicate PC/Register Files
- Real Case:
  - Intel HyperThreading (supports up to two threads per core)
    - Intel Pentium 4, Intel Atom, Intel Core i7
  - AMD RyZen (Zen microarchitecture)

Wide-issue SS processor v.s. multiple narrower-issue SS processors

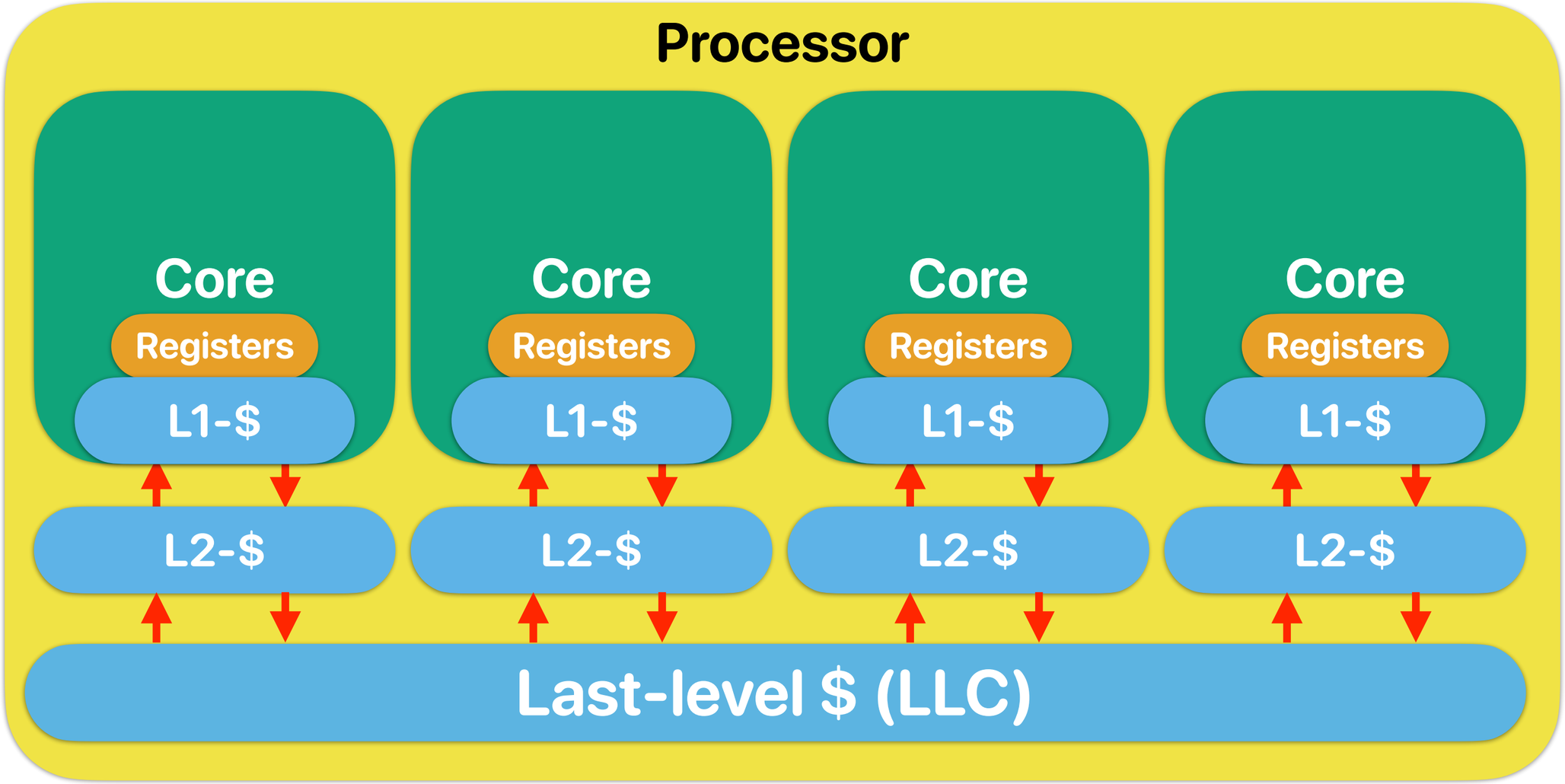


6-way SS processor —
3 INT ALUs, 3 FP ALUs
I-cache: 32KB, D-cache: 32KB



4 2-issue SS processor —
4* (1 INT ALUs, 1 FP ALUs
I-cache: 8KB, D-cache: 8KB)

Recap: Concept of CMP



Recap: SMT v.s. CMP

- An SMT processor is basically a SuperScalar processor with multiple instruction front-end. Assume within the same chip area, we can build an SMT processor supporting 4 threads, with 6-issue pipeline, 64KB cache or — a CMP with 4x 2-issue pipeline & 16KB cache in each core. Please identify how many of the following statements are/is correct when running programs on these processors.
  - ① If we are just running one program in the system, the program will perform better on an SMT processor **— you have more resources for the program**
  - ② If we are running 4 applications simultaneously, the cache miss rates will be higher in the SMT processor **— it depends!**
  - ③ If we are running 4 applications simultaneously, the branch mis-prediction will be higher in the SMT processor **— it depends!**
  - ④ If we are running one program with 4 parallel threads, the cache miss rates will be higher in the SMT processor **— it depends!**
  - ⑤ If we are running one program with 4 parallel threads simultaneously, the branch mis-prediction will be longer in the SMT processor **— it depends!**

  A. 1
  B. 2
  C. 3 **The only thing we know for sure — if we don't parallel the program, it won't get any faster on CMP**
  D. 4
  E. 5

Architectural Support for Parallel Programming

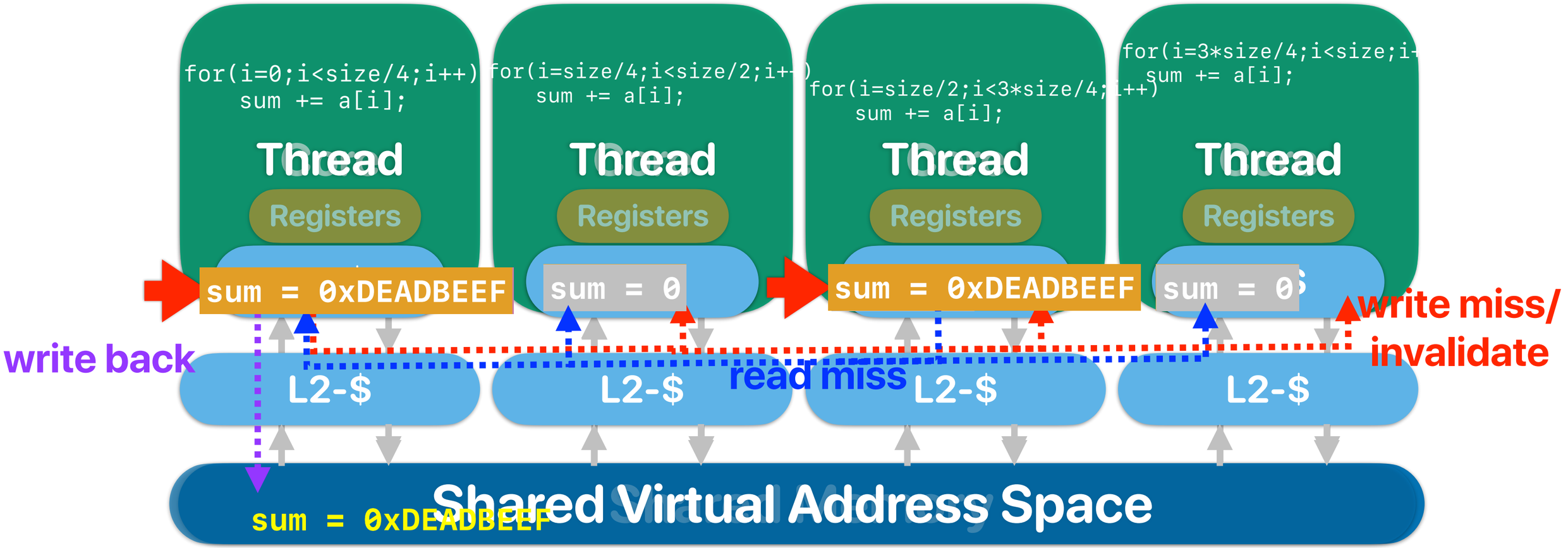
Coherency & Consistency

- Coherency — Guarantees all processors see the same value for a variable/memory address in the system when the processors need the value at the same time
  - What value should be seen
- Consistency — All threads see the change of data in the same order
  - When the memory operation should be done

Simple cache coherency protocol

- Snooping protocol
  - Each processor broadcasts / listens to cache misses
- State associate with each block (cacheline)
  - Invalid
    - The data in the current block is invalid
  - Shared
    - The processor can read the data
    - The data may also exist on other processors
  - Exclusive
    - The processor has full permission on the data
    - The processor is the only one that has up-to-date data

What happens when we write in coherent caches?



Observer

thread 1

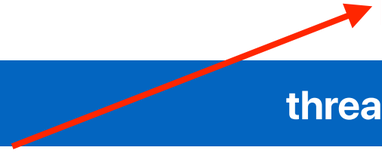
```
int loop;

int main()
{
  pthread_t thread;
  loop = 1;

  pthread_create(&thread, NULL, modifyloop,
NULL);
  while(loop == 1)
  {
    continue;
  }
  pthread_join(thread, NULL);
  fprintf(stderr,"User input: %d\n", loop);
  return 0;
}
```

thread 2

```
void* modifyloop(void *x)
{
  sleep(1);
  printf("Please input a number:\n");
  scanf("%d",&loop);
  return NULL;
}
```

Observer

prevents the compiler from putting the variable “loop" in the “register”



thread 1	thread 2

```
volatile int loop;

int main()
{
  pthread_t thread;
  loop = 1;

  pthread_create(&thread, NULL, modifyloop,
NULL);
  while(loop == 1)
  {
    continue;
  }
  pthread_join(thread, NULL);
  fprintf(stderr,"User input: %d\n", loop);
  return 0;
}
```

```
void* modifyloop(void *x)
{
  sleep(1);
  printf("Please input a number:\n");
  scanf("%d",&loop);
  return NULL;
}
```

Outline

- Parallel programming
- GPU

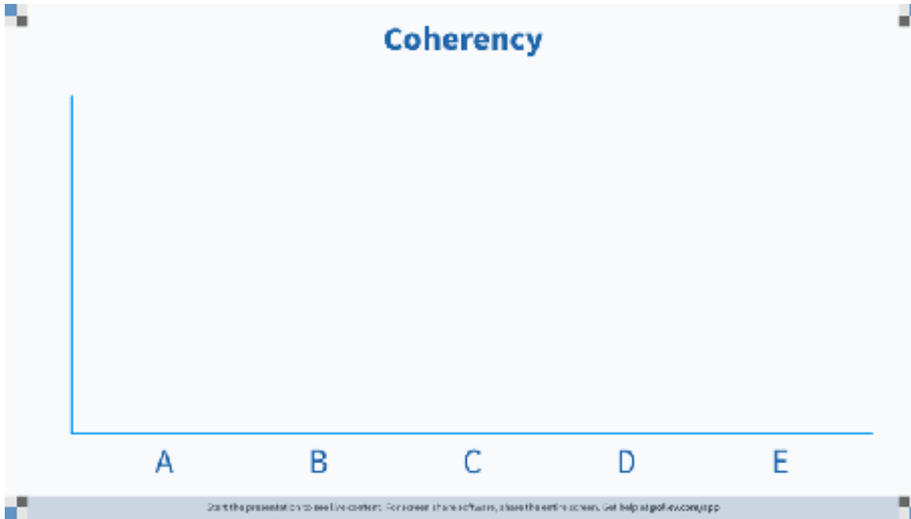
Cache coherency

- Assuming that we are running the following code on a CMP with a cache coherency protocol, how many of the following outputs are possible? (a is initialized to 0 as assume we will output more than 10 numbers)

thread 1	thread 2
`while(1)` `  printf("%d ",a);`	`while(1)` `  a++;`

① 0 1 2 3 4 5 6 7 8 9
② 1 2 5 9 3 6 8 10 12 13
③ 1 1 1 1 1 1 1 64 100
④ 1 1 1 1 1 1 1 1 1 100

A. 0
B. 1
C. 2
D. 3
E. 4



Cache coherency

- Assuming that we are running the following code on a CMP with a cache coherency protocol, how many of the following outputs are possible? (a is initialized to 0 as assume we will output more than 10 numbers)

thread 1	thread 2
`while(1)` `  printf("%d ",a);`	`while(1)` `  a++;`

① 0 1 2 3 4 5 6 7 8 9

② 1 2 5 9 3 6 8 10 12 13

③ 1 1 1 1 1 1 1 64 100

④ 1 1 1 1 1 1 1 1 1 100

A. 0

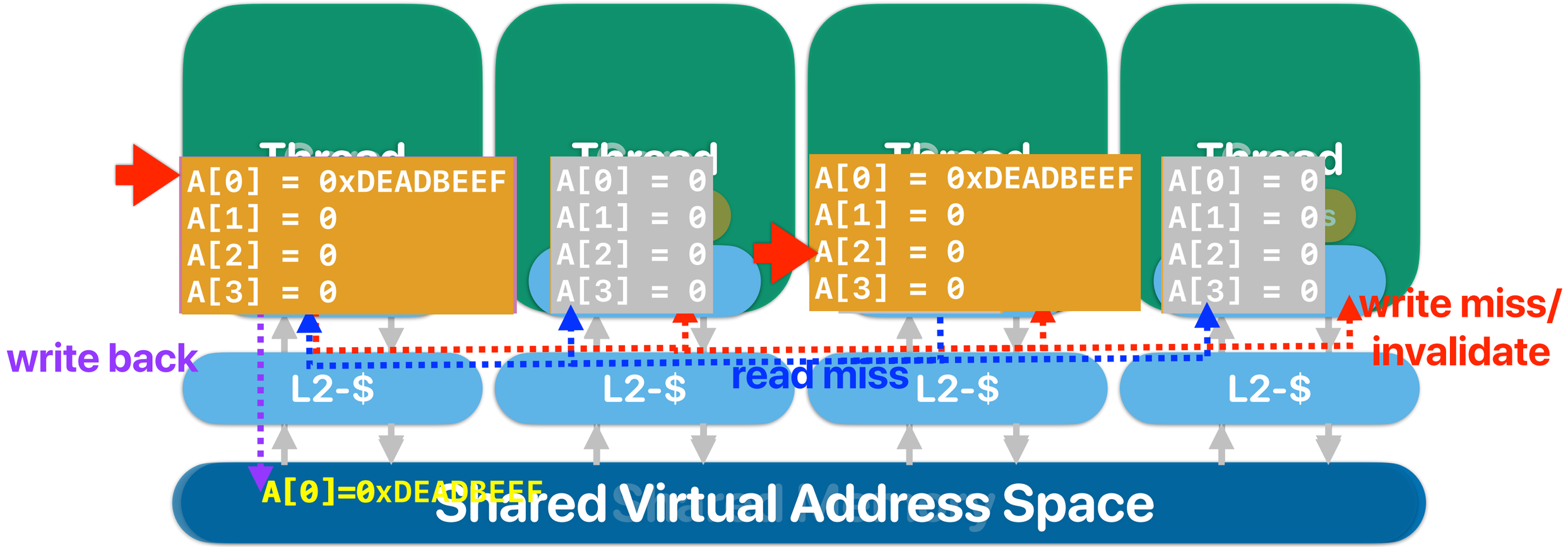
B. 1

C. 2

D. 3

E. 4

Cache coherency



Performance comparison

- Comparing implementations of thread_vadd — L and R, please identify which one will be performing better and why

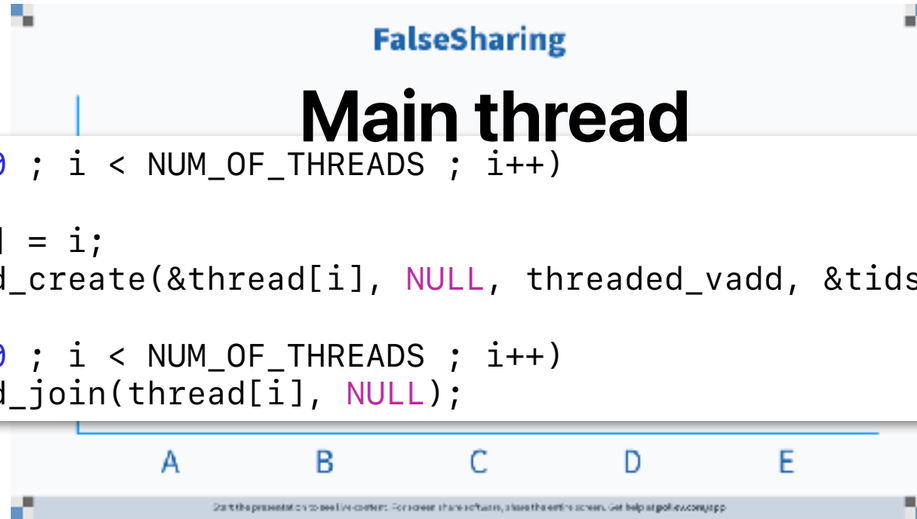
Version L

```
void *threaded_vadd(void *thread_id)
{
  int tid = *(int *)thread_id;
  int i;
  for(i=tid;i<ARRAY_SIZE;i+=NUM_OF_THREADS)
  {
        c[i] = a[i] + b[i];
  }
  return NULL;
}
```

Version R

```
void *threaded_vadd(void *thread_id)
{
  int tid = *(int *)thread_id;
  int i;
  for(i=tid*(ARRAY_SIZE/NUM_OF_THREADS);i<(tid+1)*(ARRAY_SIZE/NUM_OF_THREADS);i++)
  {
      c[i] = a[i] + b[i];
  }
  return NULL;
}
```

A. L is better, because the cache miss rate is lower
B. R is better, because the cache miss rate is lower
C. L is better, because the instruction count is lower
D. R is better, because the instruction count is lower
E. Both are about the same



L v.s. R

Version L

```
void *threaded_vadd(void *thread_id)
{
  int tid = *(int *)thread_id;
  int i;
  for(i=tid;i<ARRAY_SIZE;i+=NUM_OF_THREADS)
  {
        c[i] = a[i] + b[i];
  }
  return NULL;
}
```

C 

Version R

```
void *threaded_vadd(void *thread_id)
{
  int tid = *(int *)thread_id;
  int i;
  for(i=tid*(ARRAY_SIZE/NUM_OF_THREADS);i<(tid+1)*(ARRAY_SIZE/NUM_OF_THREADS);i++)
  {
      c[i] = a[i] + b[i];
  }
  return NULL;
}
```

C 

4Cs of cache misses

- 3Cs:
  - Compulsory, Conflict, Capacity
- Coherency miss:
  - A “block” invalidated because of the sharing among processors.

False sharing

- True sharing
  - Processor A modifies X, processor B also want to access X.
- False sharing
  - Processor A modifies X, processor B also want to access Y. However, Y is invalidated because X and Y are in the same block!

Performance comparison

- Comparing implementations of thread_vadd — L and R, please identify which one will be performing better and why

Version L

```
void *threaded_vadd(void *thread_id)
{
  int tid = *(int *)thread_id;
  int i;
  for(i=tid;i<ARRAY_SIZE;i+=NUM_OF_THREADS)
  {
        c[i] = a[i] + b[i];
  }
  return NULL;
}
```

Version R

```
void *threaded_vadd(void *thread_id)
{
  int tid = *(int *)thread_id;
  int i;
  for(i=tid*(ARRAY_SIZE/NUM_OF_THREADS);i<(tid+1)*(ARRAY_SIZE/NUM_OF_THREADS);i++)
  {
      c[i] = a[i] + b[i];
  }
  return NULL;
}
```

A. L is better, because the cache miss rate is lower
B. R is better, because the cache miss rate is lower
C. L is better, because the instruction count is lower
D. R is better, because the instruction count is lower
E. Both are about the same

Main thread

```
for(i = 0 ; i < NUM_OF_THREADS ; i++)
{
  tids[i] = i;
  pthread_create(&thread[i], NULL, threaded_vadd, &tids
}
for(i = 0 ; i < NUM_OF_THREADS ; i++)
  pthread_join(thread[i], NULL);
```

Again — how many values are possible?

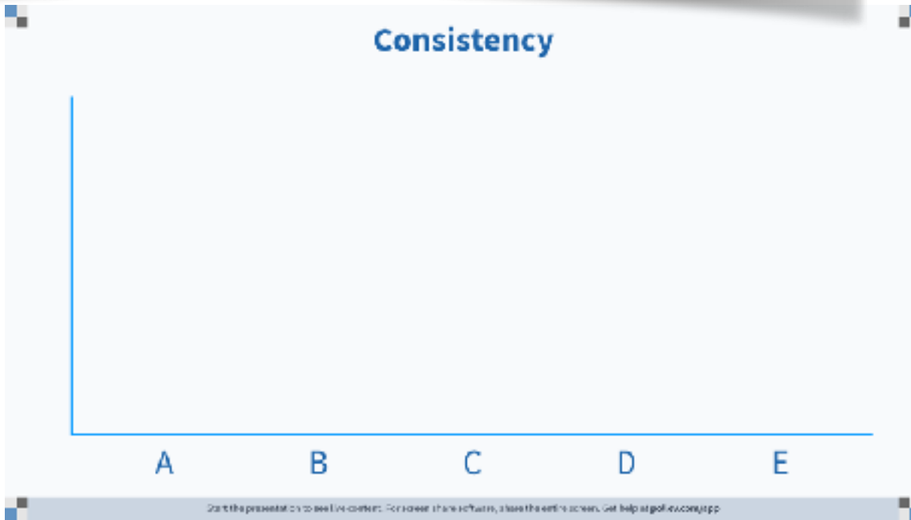
- Consider the given program. You can safely assume the caches are coherent. How many of the following outputs will you see?

① (0, 0)
② (0, 1)
③ (1, 0)
④ (1, 1)

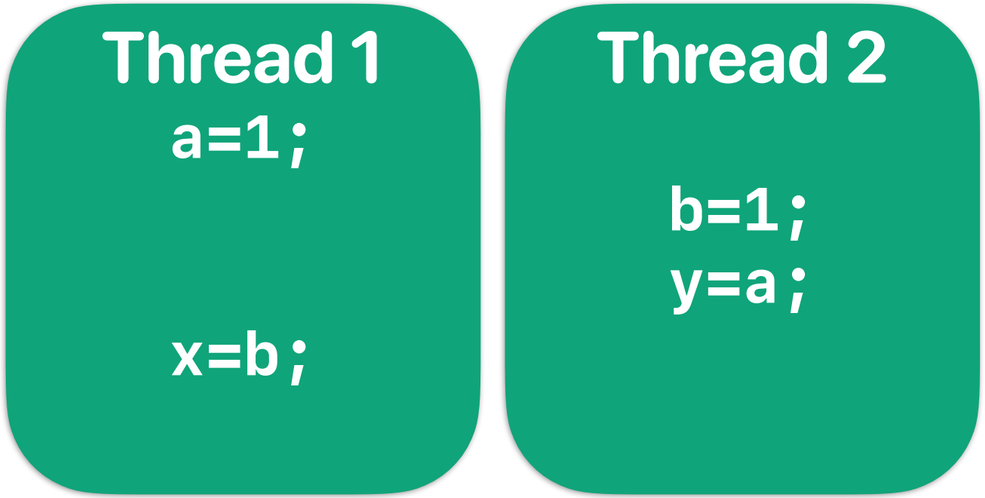
A. 0
B. 1
C. 2
D. 3
E. 4

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

volatile int a,b;
volatile int x,y;
volatile int f;
void* modifya(void *z) {
  a=1;
  x=b;
  return NULL;
}
void* modifyb(void *z) {
  b=1;
  y=a;
  return NULL;
}
```

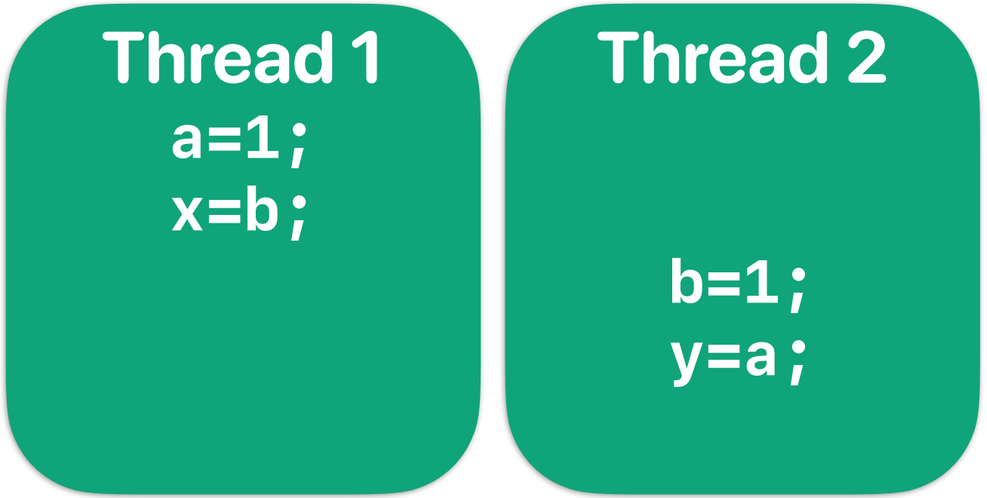
```c
int main() {
  int i;
  pthread_t thread[2];
  pthread_create(&thread[0], NULL, modifya, NULL);
  pthread_create(&thread[1], NULL, modifyb, NULL);
  pthread_join(thread[0], NULL);
  pthread_join(thread[1], NULL);
  fprintf(stderr,"(%d, %d)\n",x,y);
  return 0;
}
```



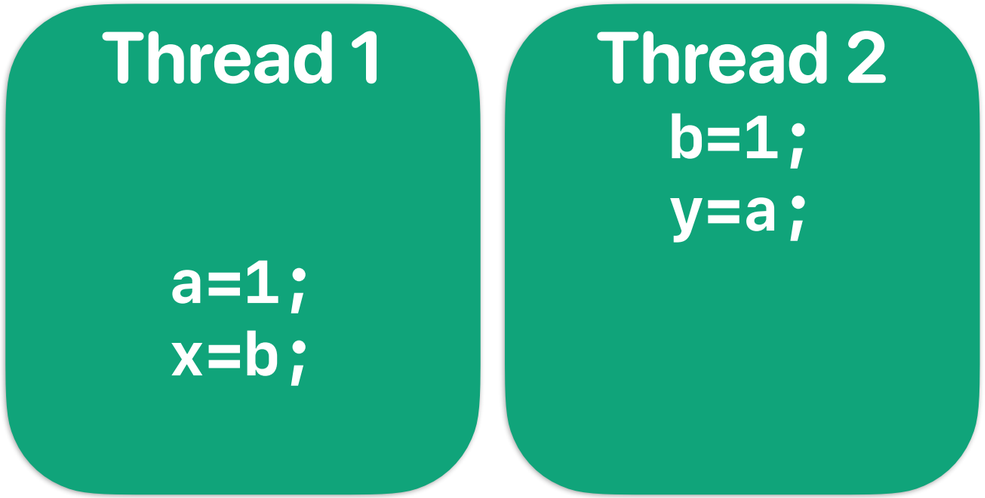
Possible scenarios



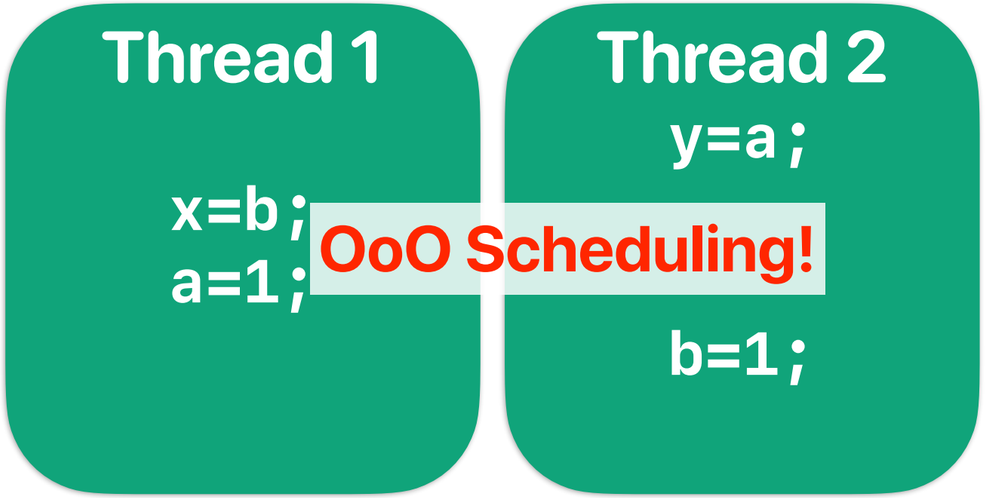
(1,1)



(0,1)



(1,0)



(0,0)

Why (0,0)?

- Processor/compiler may reorder your memory operations/ instructions
  - Coherence protocol can only guarantee the update of the same memory address
  - Processor can serve memory requests without cache miss first
  - Compiler may store values in registers and perform memory operations later
- Each processor core may not run at the same speed (cache misses, branch mis-prediction, I/O, voltage scaling and etc..)
- Threads may not be executed/scheduled right after it's spawned

Again — how many values are possible?

- Consider the given program. You can safely assume the caches are coherent. How many of the following outputs will you see?

① (0, 0)
② (0, 1)
③ (1, 0)
④ (1, 1)

A. 0
B. 1
C. 2
D. 3
E. 4

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

volatile int a,b;
volatile int x,y;
volatile int f;
void* modifya(void *z) {
  a=1;
  x=b;
  return NULL;
}
void* modifyb(void *z) {
  b=1;
  y=a;
  return NULL;
}
```

```c
int main() {
  int i;
  pthread_t thread[2];
  pthread_create(&thread[0], NULL, modifya, NULL);
  pthread_create(&thread[1], NULL, modifyb, NULL);
  pthread_join(thread[0], NULL);
  pthread_join(thread[1], NULL);
  fprintf(stderr,"(%d, %d)\n",x,y);
  return 0;
}
```

fence instructions

- x86 provides an "mfence" instruction to prevent reordering across the fence instruction
- x86 only supports this kind of "relaxed consistency" model. You still have to be careful enough to make sure that your code behaves as you expected

thread 1	thread 2

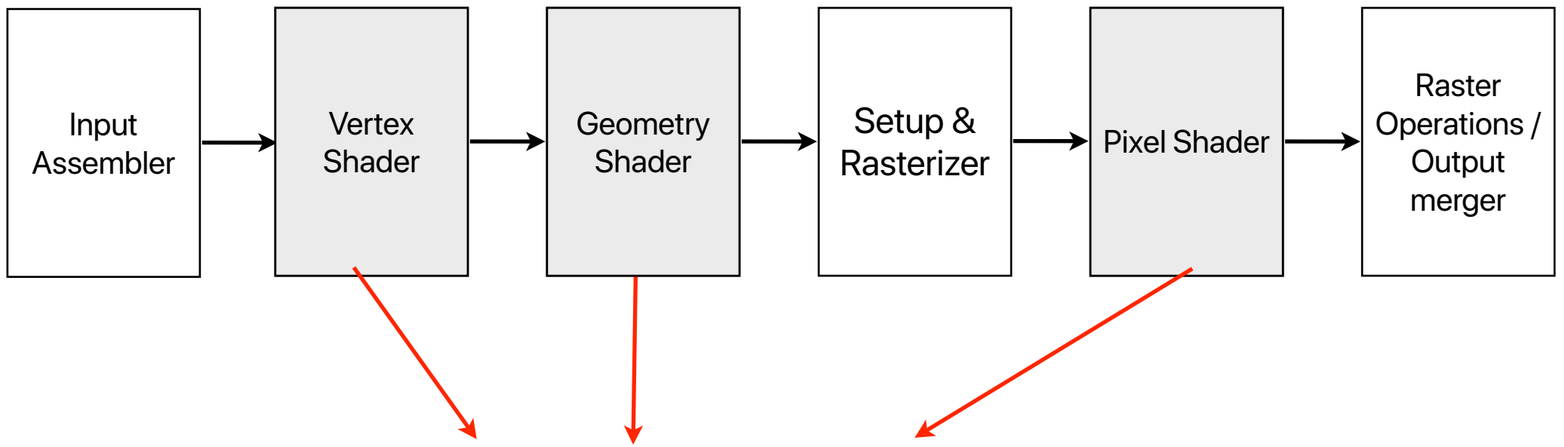
`a=1;`
`mfence` **a=1 must occur/update before mfence**
`x=b;`
	`b=1;`
	`mfence` **b=1 must occur/update before mfence**
	`y=a;`

Take-aways of parallel programming

- Processor behaviors are non-deterministic
  - You cannot predict which processor is going faster
  - You cannot predict when OS is going to schedule your thread
- Cache coherency only guarantees that everyone would eventually have a coherent view of data, but not when
- Cache consistency is hard to support

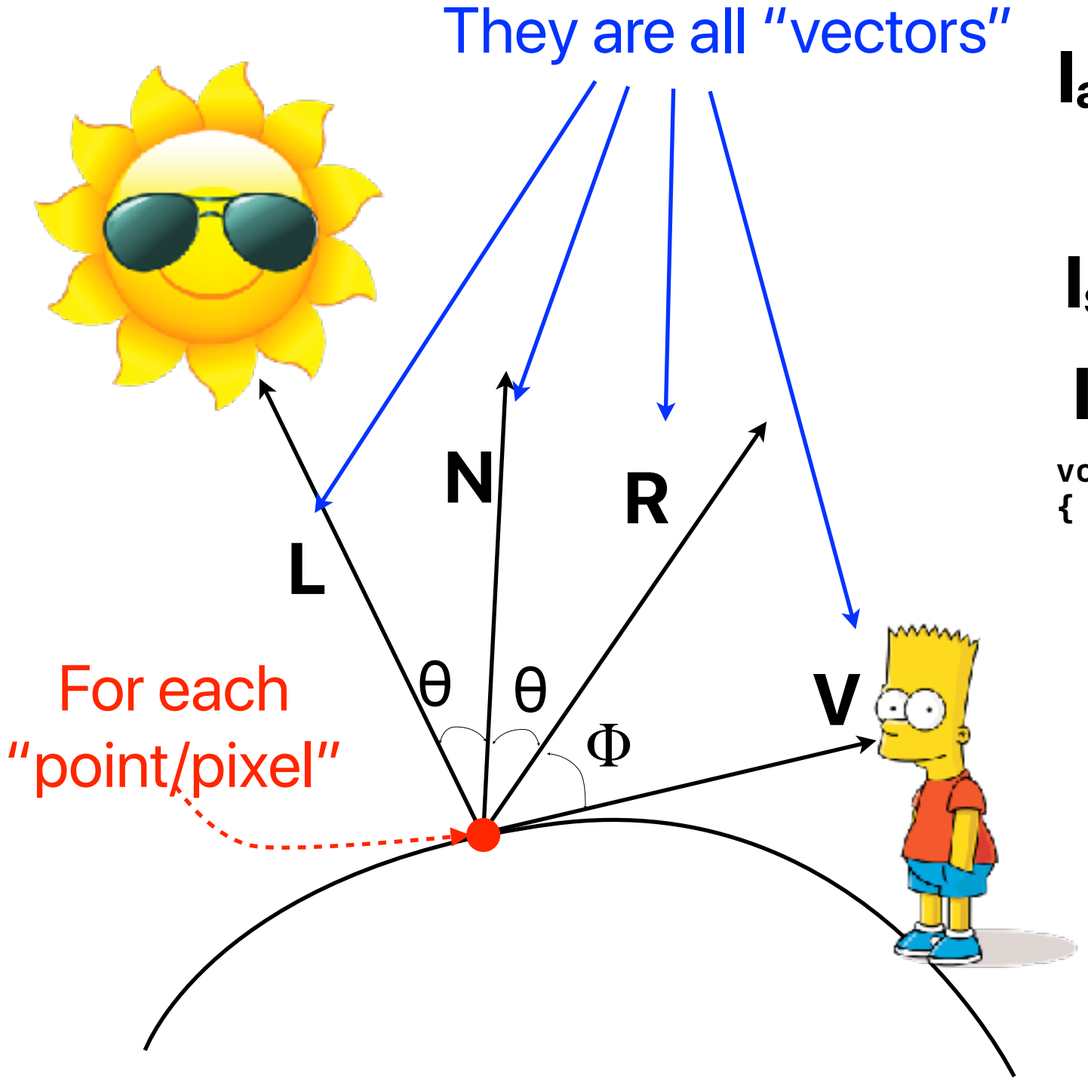
Alternative Parallel Architectures

GPU (Graphics Processing Unit)



These shaders need to be “programmable” to apply different rendering effects/algorithms (Phong shading, Gouraud shading, and etc…)

Basic concept of shading



$$I_{amb} = K_{amb} \cdot M_{amb}$$

$$I_{diff} = K_{diff} \cdot M_{diff} \cdot (N \cdot L)$$

$$I_{spec} = K_{spec} \cdot M_{spec} \cdot (R \cdot V)^n$$

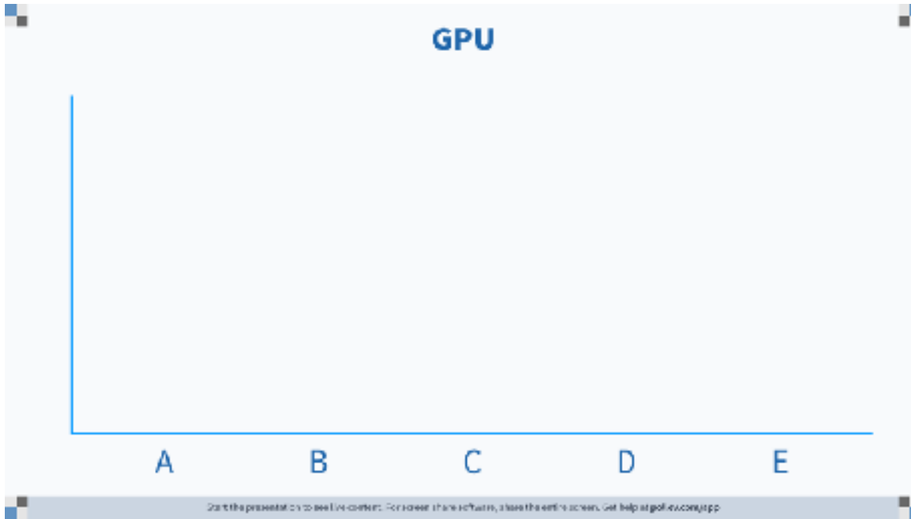
$$I_{total} = I_{amb} + I_{diff} + I_{spec}$$

```
void main(void)
{
    // normalize vectors after interpolation
    vec3 L = normalize(o_toLight);
    vec3 V = normalize(o_toCamera);
    vec3 N = normalize(o_normal);

    // get Blinn-Phong reflectance components
    float Iamb = ambientLighting();
    float Idif = diffuseLighting(N, L);
    float Ispe = specularLighting(N, L, V);

    // diffuse color of the object from texture
    vec3 diffuseColor = texture(u_diffuseTexture, o_texcoords).rgb;

    // combination of all components and diffuse color of the object
    resultingColor.xyz = diffuseColor * (Iamb + Idif + Ispe);
    resultingColor.a = 1;
}
```

What do you want from a GPU?

- Given the basic idea of shading algorithms, how many of the following statements would fit the agenda of designing a GPU?
  - ① Many ALUs to process multiple pixels simultaneously
  - ② Low latency memory bus to supply pixels, vectors and textures
  - ③ High performance branch predictors
  - ④ Powerful ALUs to process many different kinds of operators

A. 0
B. 1
C. 2
D. 3
E. 4



What do you want from a GPU?

- Given the basic idea of shading algorithms, how many of the following statements would fit the agenda of designing a GPU?
  - ① Many ALUs to process multiple pixels simultaneously
  - ② Low latency memory bus to supply pixels, vectors and textures
  - ③ ~~High performance branch predictors~~
  - ④ ~~Powerful ALUs to process many different kinds of operators~~

Each frame contains 1920*1080 pixels!

Actually, high bandwidth since each pixel requires different L, N, R, V and we need to feed thousands of pixels simultaneously

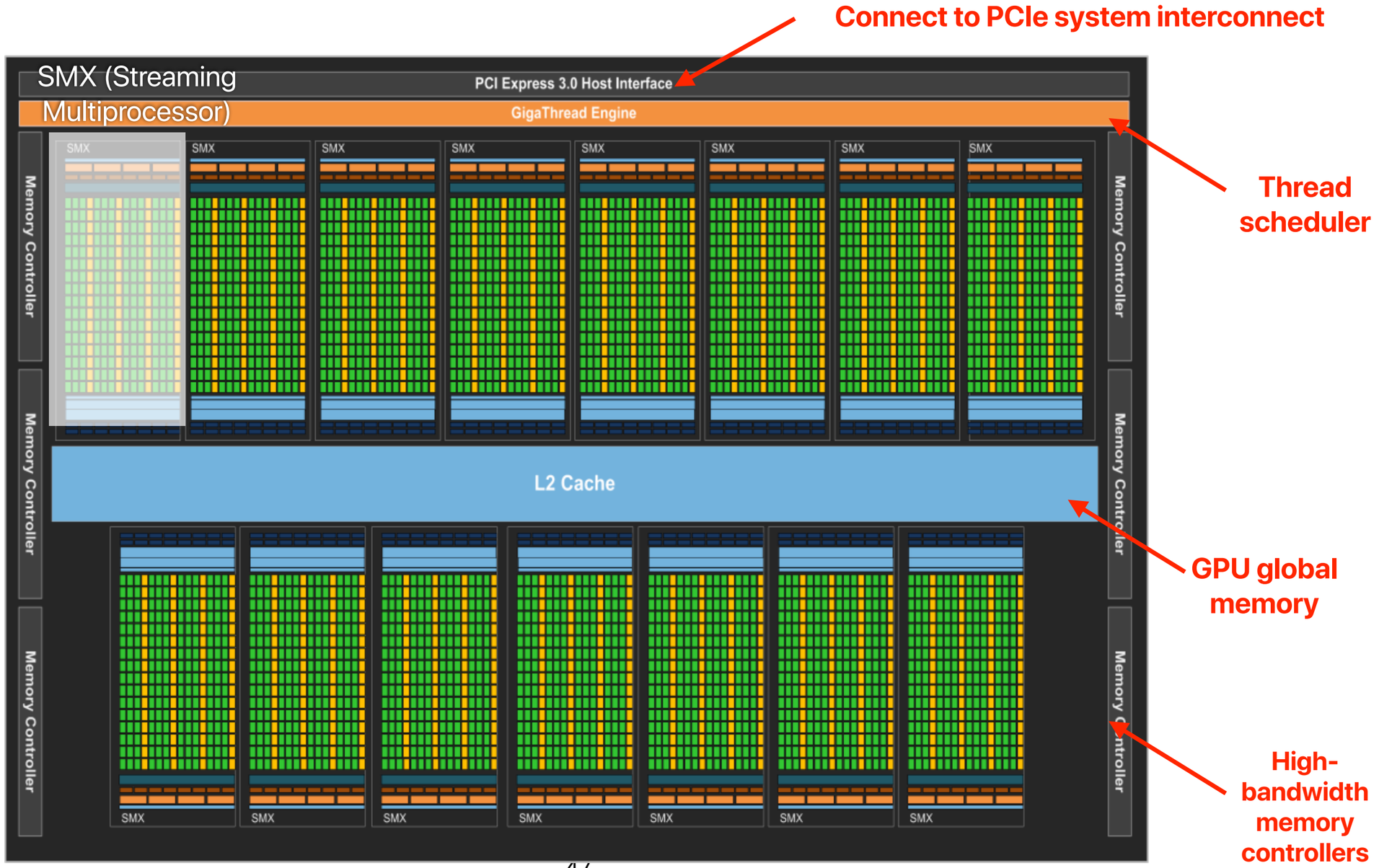
not really, the behavior is uniform across all pixels

not really, we only need vector add, vector mul, vector div. Low frequency is OK since we have many threads

In terms of latency, even for 120 frames, you still have 8ms latency to get everything done!

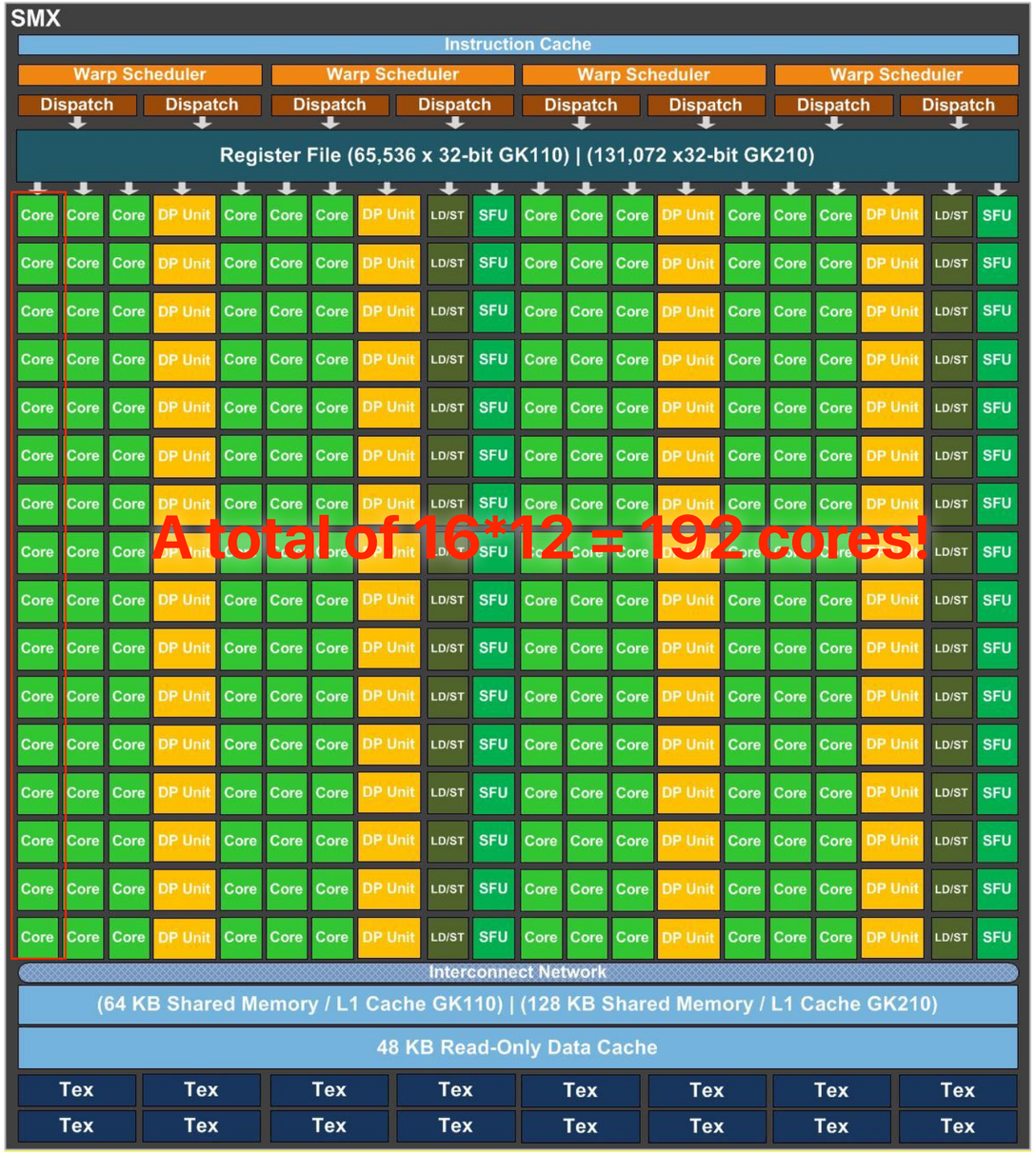
A. 0
B. 1
C. 2
D. 3
E. 4

Nvidia GPU architecture

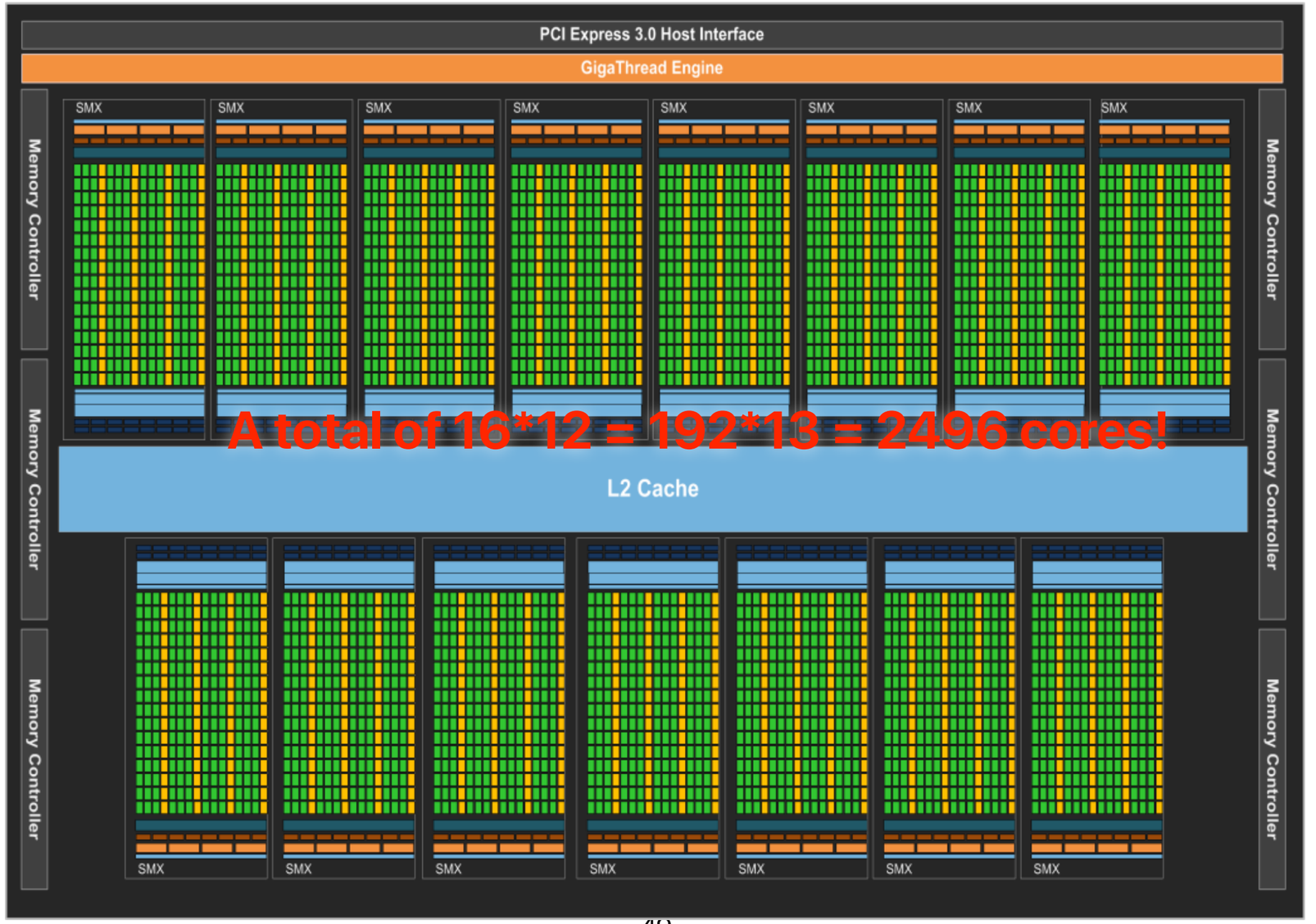


Inside each SMX

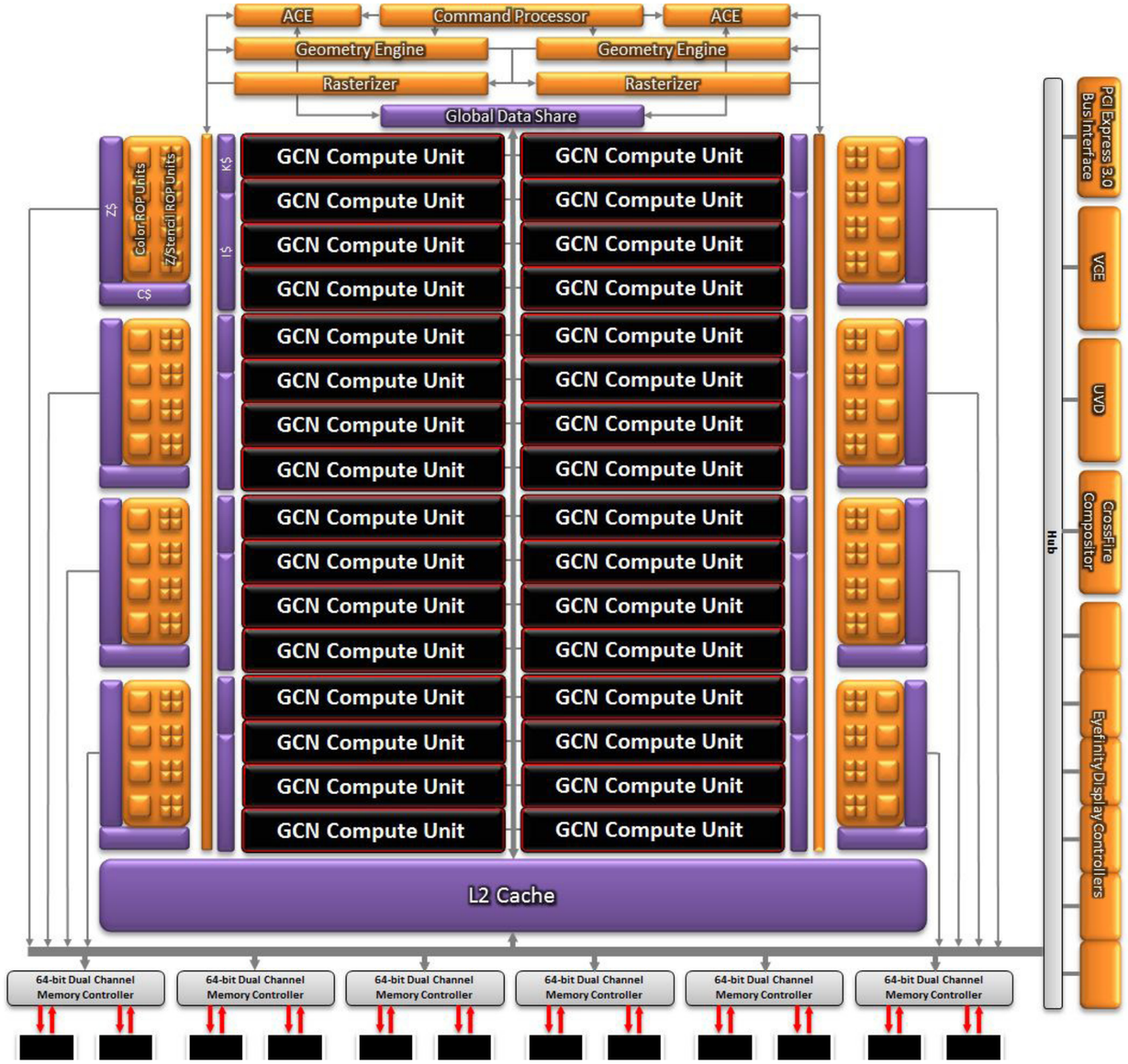
Each of these performs the same operation, but each of these is also a "thread"



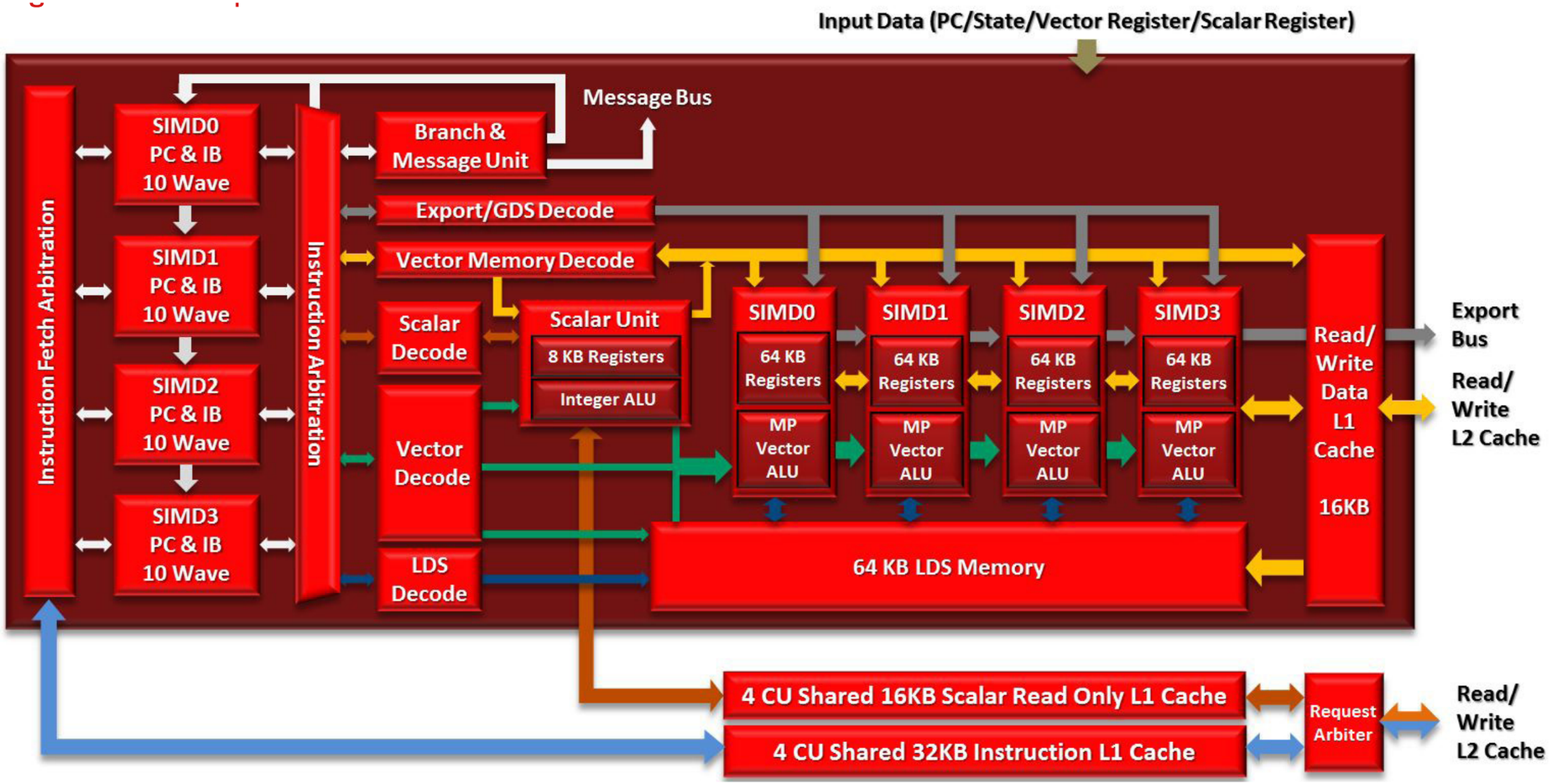
Nvidia GPU architecture



AMD GPU Architecture

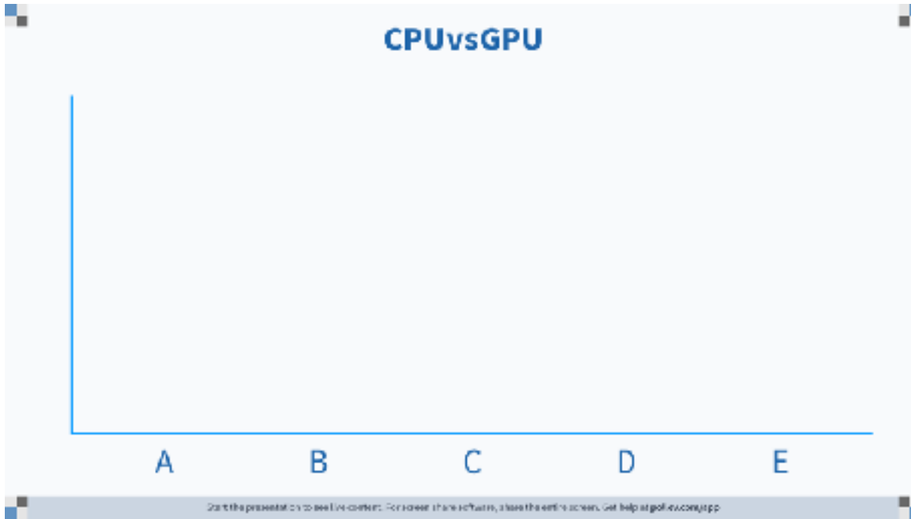


A CU in an AMD GPU



CPU v.s. GPU

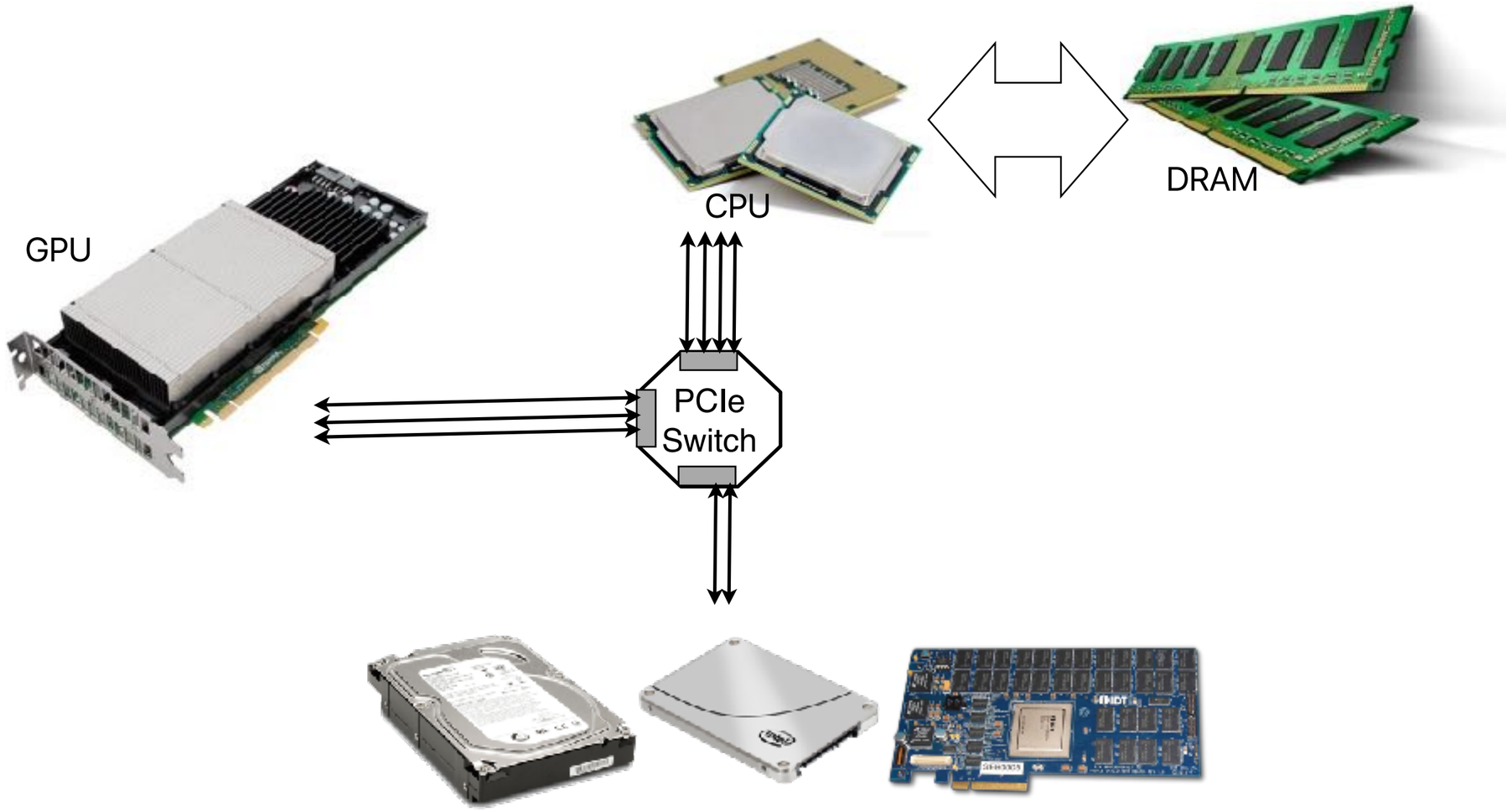
- Comparing the performance of solving the following set of problems using modern CPU and GPU architectures, how many can GPUs outperform CPUs?
  - ① Matrix multiplications
  - ② Minimum Spanning Trees
  - ③ Shortest Path Problems
  - ④ Gaussian Elimination
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4



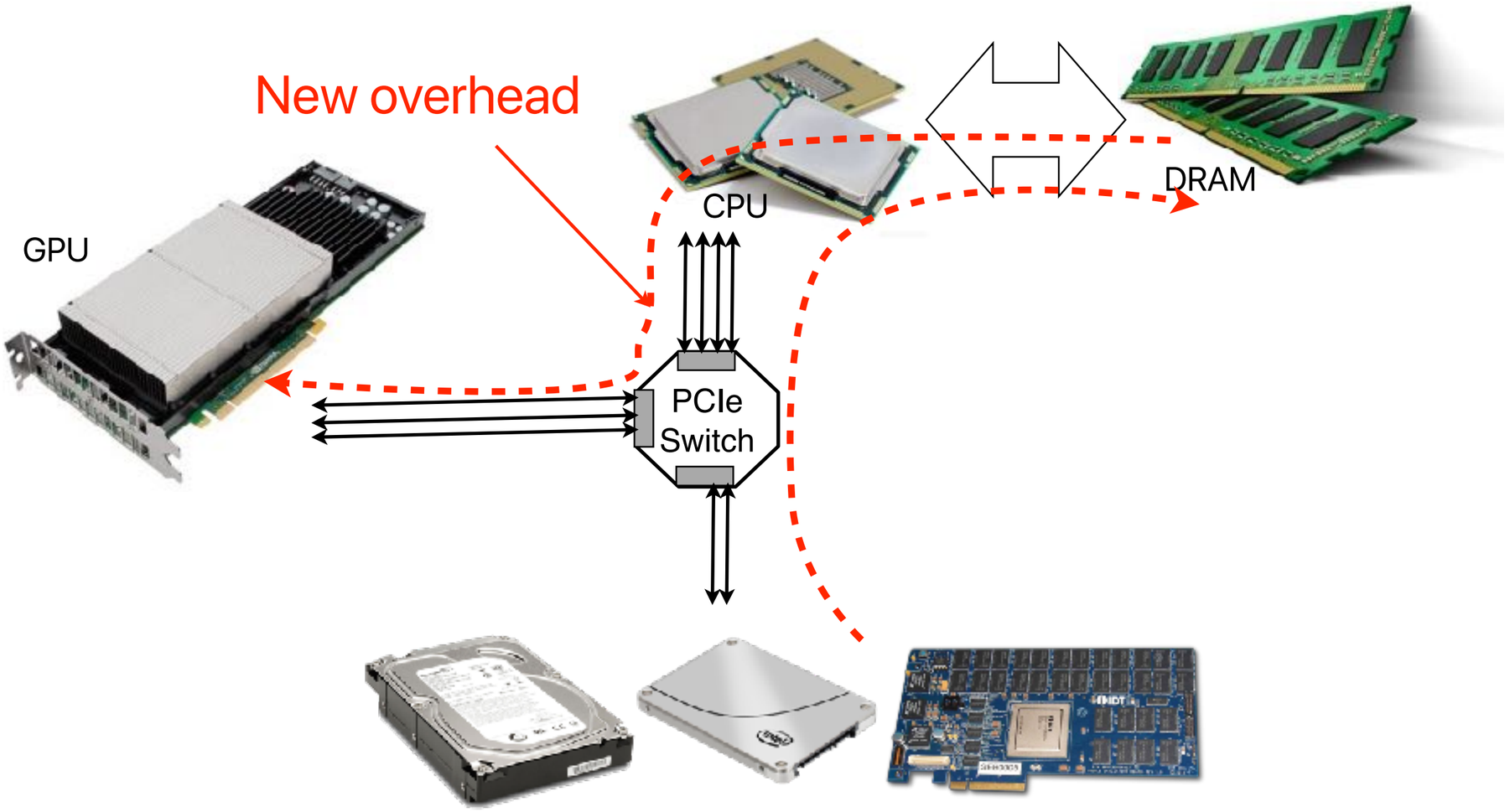
CPU v.s. GPU

- Comparing the performance of solving the following set of problems using modern CPU and GPU architectures, how many can GPUs outperform CPUs?
  - ① Matrix multiplications
  - ~~②~~ Minimum Spanning Trees
  - ~~③~~ Shortest Path Problems
  - ④ Gaussian Elimination
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

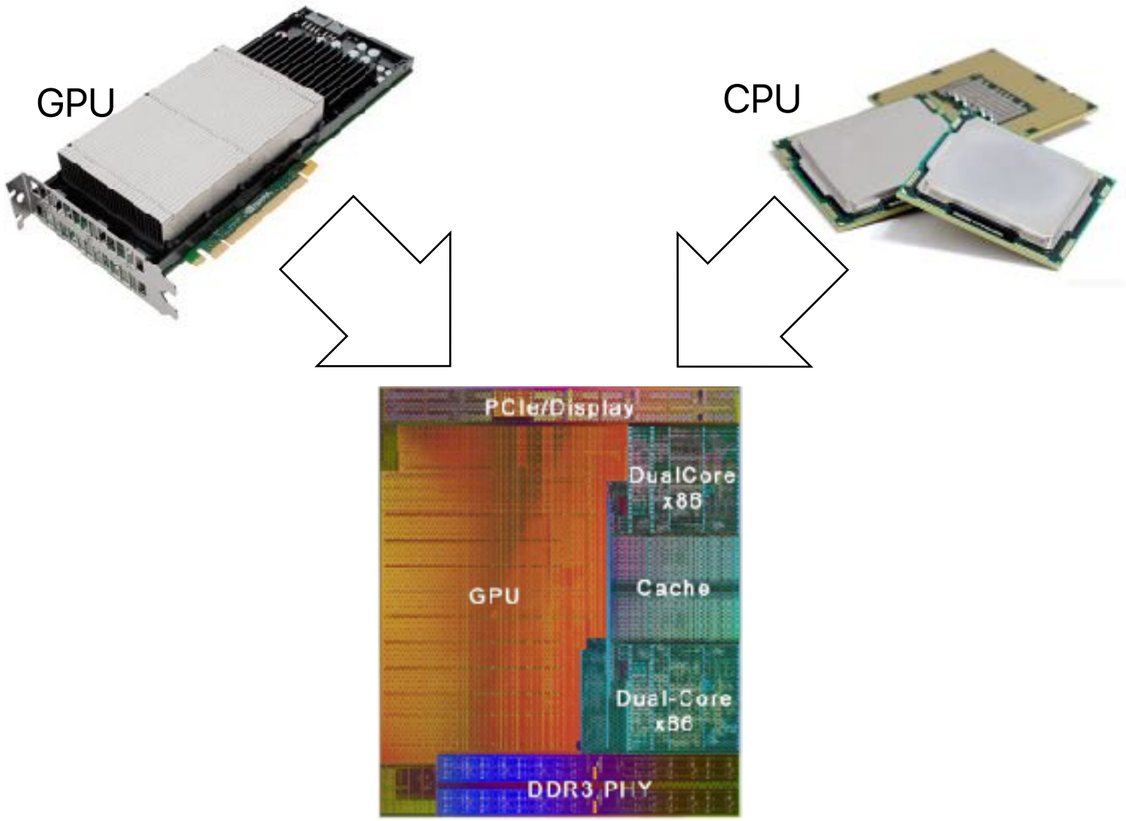
How things are connected



New overhead/bottleneck emerges



APU (Accelerated Processing Unit)

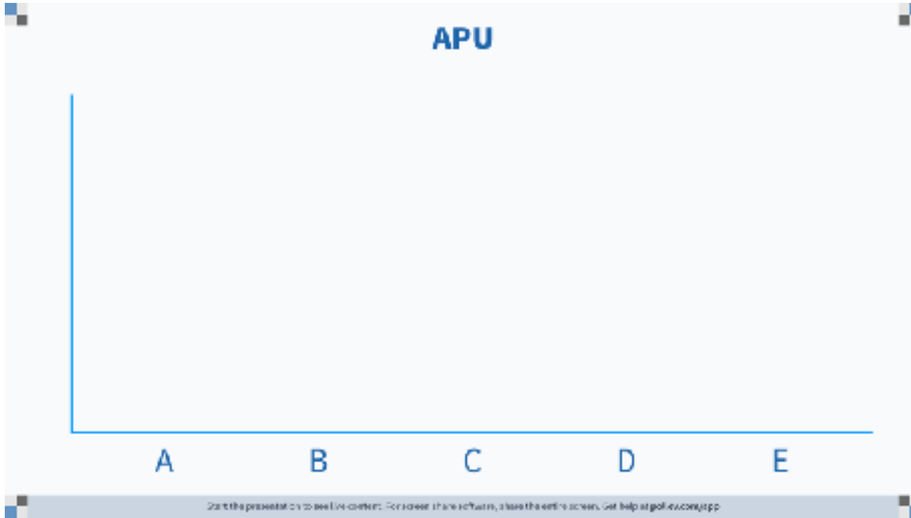


It’s now very common in intel and AMD lineups

APU

- Regarding the pros and cons of an APU, how many of the followings are correct
  - ① APU eliminates the need of moving data from DRAM to GPU device memory
  - ② The memory bandwidth of an APU is generally better than that of GPU device memory
  - ③ The total number of ALUs that an APU can provide must be fewer than a discrete GPU given the same power budget and chip area
  - ④ A memory intensive GPU kernel can slowdown the performance of another CPU program

A. 0
B. 1
C. 2
D. 3
E. 4



APU

- Regarding the pros and cons of an APU, how many of the followings are correct
  - ① APU eliminates the need of moving data from DRAM to GPU device memory
  - ② The memory bandwidth of an APU is generally better than that of GPU device memory
  - ③ The total number of ALUs that an APU can provide must be fewer than a discrete GPU given the same power budget and chip area
  - ④ A memory intensive GPU kernel can slowdown the performance of another CPU program

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

CPUs/GPUs in an APU are all connected to DRAM

Not true, APU uses CPU memory bus that is optimized for "latencies"

CPU cores are really big due to the dynamic scheduling logic

Because they have to compete with the DRAM bandwidth

Announcement

- Project due next Monday
- Last reading quiz due next Monday
- Assignment #4 due next Wednesday
- iEVAL, until 12/3
  - Please fill the survey to let us know your opinion!
  - Don’t forget to take a screenshot of your submission and submit through iLearn — it counts as a **full credit assignment**
  - **We will drop your lowest 2 assignment grades**
- Final Exam
  - Starting from 12/6 to 12/10 12:00pm, any consecutive 180 minutes you pick
  - Similar to the midterm, but more time and about 1.5x longer
  - Two of the questions will be comprehensive exam questions
  - Will release a sample final at the end of the last lecture

Computer Science & Engineering

203