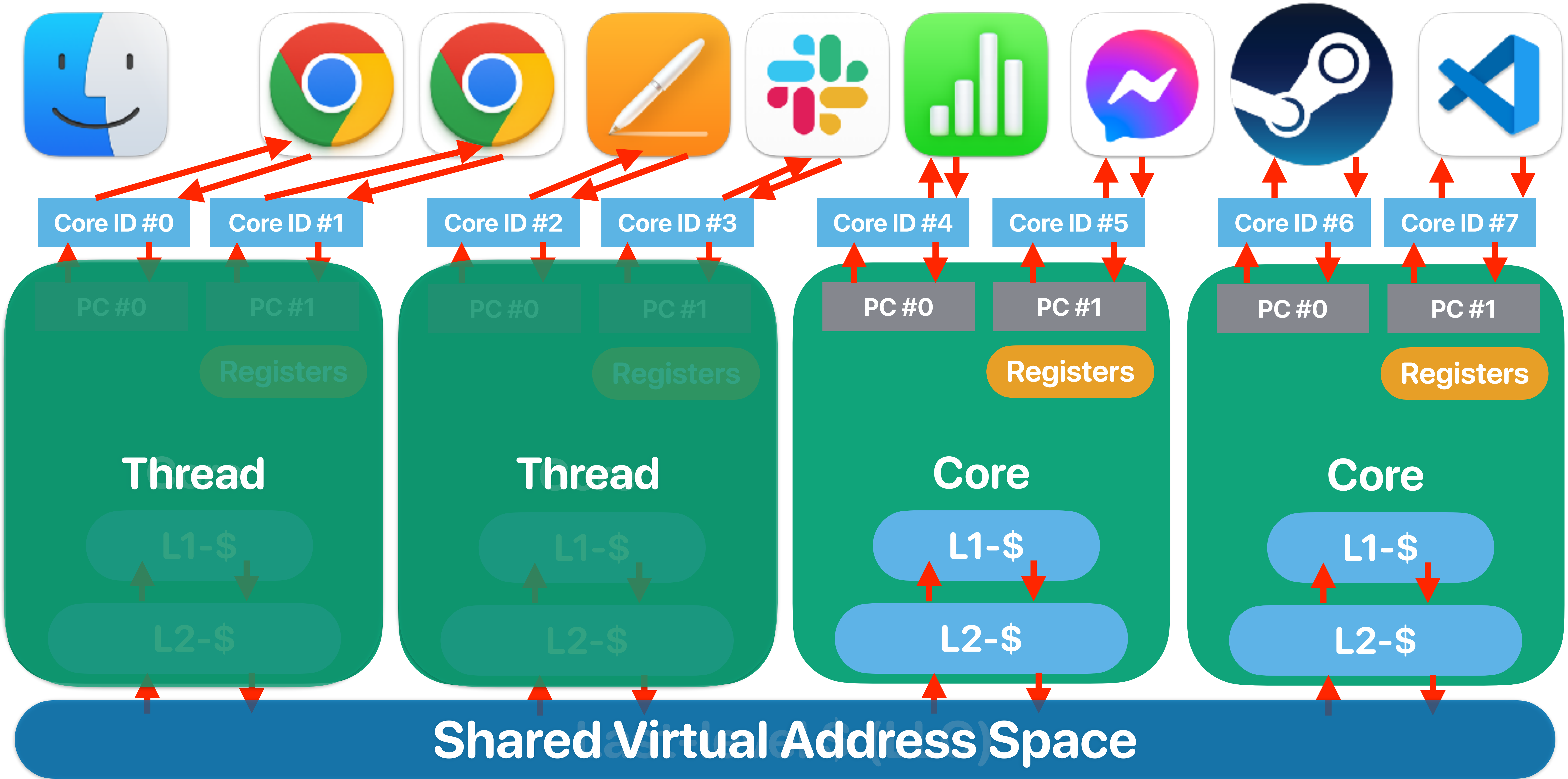
Multithreaded Architectures and Programming on Multithreaded Architectures (2): Never say never

Hung-Wei Tseng

Recap: Modern CMP/SMT and mis-aligned software abstractions



Recap: 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

Observer

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

thread 1	thread 2

```c
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",
```

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

Parallel Programming & Architectural Supports for Parallel Programming (cont.)

https://www.pollev.com/hungweitseng close in 1:30

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

- 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

Take-aways: parallel programming

- Cache coherency only guarantees that everyone would eventually have a coherent view of data, but not when

https://www.pollev.com/hungweitseng close in 1:30

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);
```

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

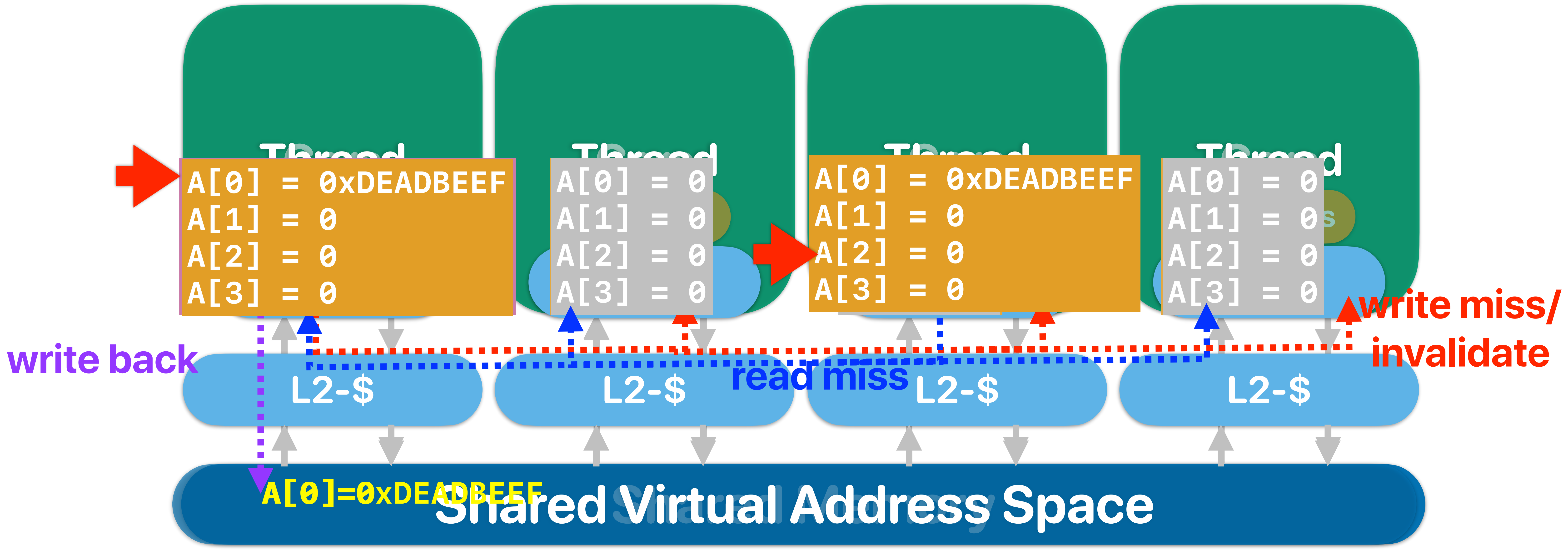
```
void *threaded_vadd(void *thread_id)
{
  int tid = *(int *)thread_id;
  int i;
  for(i=tid*(ARRAY_SIZE/NUM_OF_THREADS);i<(tid+1)*(ARRAY_SIZE/NUM_OF_THREADS);i++)
  {
      c[i] = a[i] + b[i];
  }
  return NULL;
}
```

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);
```

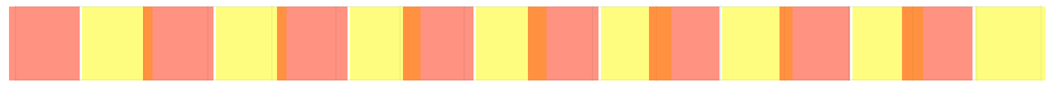
Cache coherency



L v.s. R

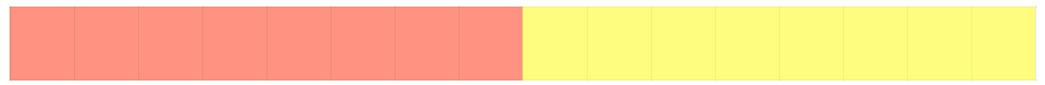
Version L

```c
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

```c
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);
```

Take-aways: parallel programming

- Cache coherency only guarantees that everyone would eventually have a coherent view of data, but not when
- Cache coherency may create unexpected cache invalidations/ misses if you do it wrong

https://www.pollev.com/hungweitseng close in 1:30

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;
}
```

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

```
Thread 1      Thread 2
a=1;
              b=1;
              y=a;
x=b;
```

(1,1)

```
Thread 1      Thread 2
a=1;
x=b;
              b=1;
              y=a;
```

(0,1)

```
Thread 1      Thread 2
              b=1;
              y=a;
a=1;
x=b;
```

(1,0)

```
Thread 1      Thread 2
              y=a;
x=b;
a=1;
              b=1;
```

OoO Scheduling!

(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;
}
```

Take-aways: parallel programming

- Cache coherency only guarantees that everyone would eventually have a coherent view of data, but not when
- Cache coherency may create unexpected cache invalidations/misses if you do it wrong
- Processor behaviors are non-deterministic
  - You cannot predict which processor is going faster
  - You cannot predict when OS is going to schedule your thread
  - You cannot predict when the processor is going to schedule an instruction

fence instructions

- x86 provides an "mfence" instruction to prevent reordering across the fence instruction
  - All updates prior to mfence must finish before the instruction can proceed
- 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

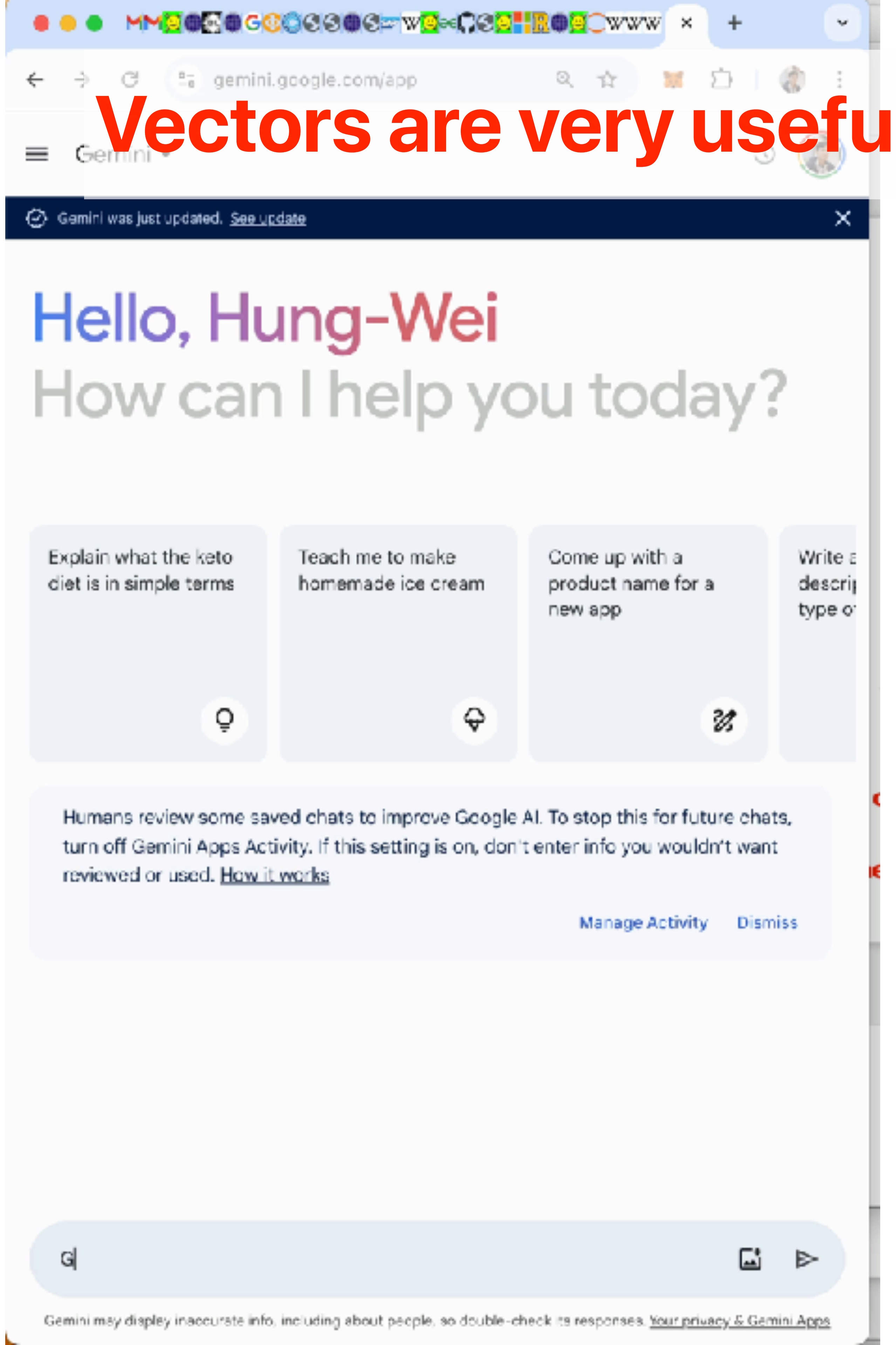
`a=1;`
`mfence` **a=1 must occur/update before mfence**
`x=b;`
	`b=1;`
	`mfence` **b=1 must occur/update before mfence**
	`y=a;`

Take-aways: parallel programming

- Cache coherency only guarantees that everyone would eventually have a coherent view of data, but not when
- Cache coherency may create unexpected cache invalidations/misses if you do it wrong
- Processor behaviors are non-deterministic
  - You cannot predict which processor is going faster
  - You cannot predict when OS is going to schedule your thread
  - You cannot predict when the processor is going to schedule an instruction
- Cache consistency is hard to support

Beyond “scalar”

Vectors are very useful data/mathematical representations



gemini.google.com/app

Gemini

Gemini was just updated. See update

Hello, Hung-Wei

How can I help you today?

Explain what the keto diet is in simple terms

Teach me to make homemade ice cream

Come up with a product name for a new app

Write a descri type o

Humans review some saved chats to improve Google AI. To stop this for future chats, turn off Gemini Apps Activity. If this setting is on, don't enter info you wouldn't want reviewed or used. How it works

Manage Activity Dismiss

G

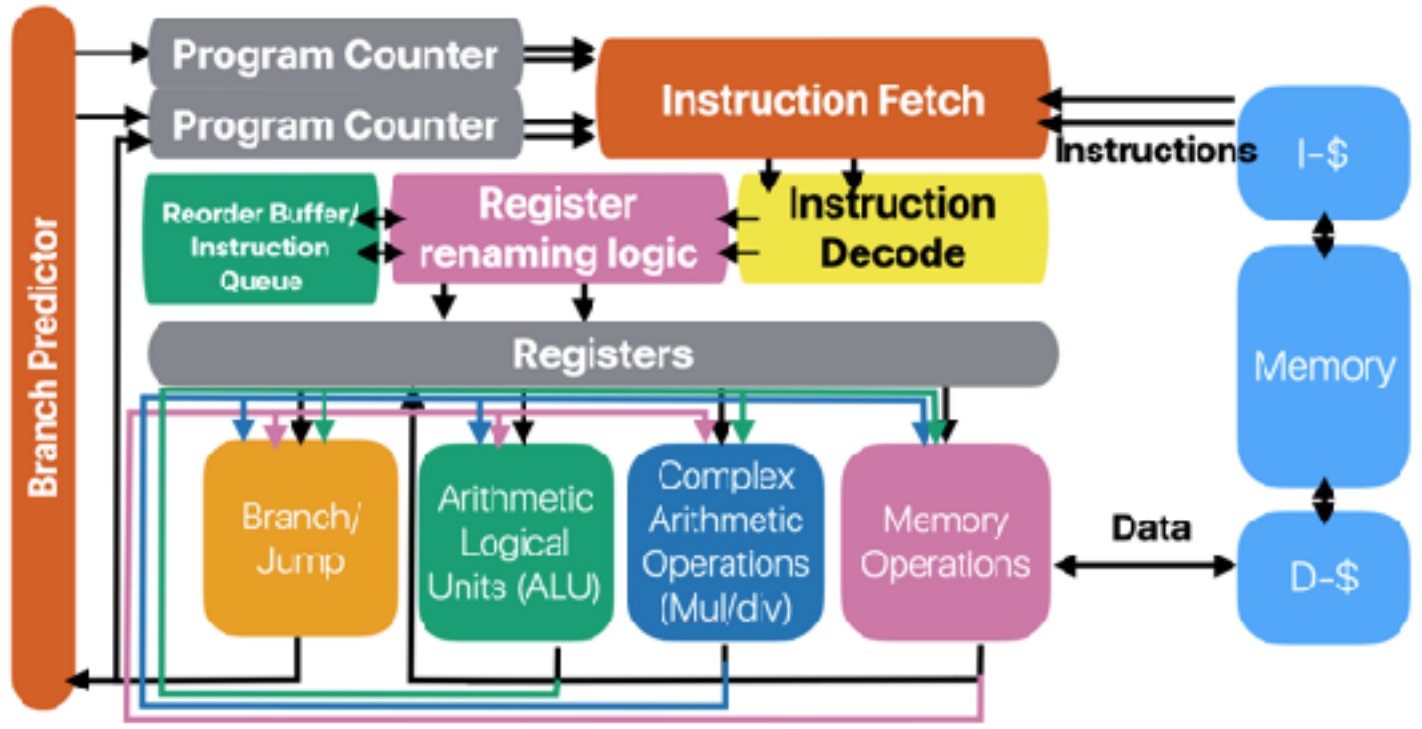
Gemini may display inaccurate info, including about people, so double-check its responses. Your privacy & Gemini Apps

- But how do we process them using "superscalar" processors?

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
    for(uint64_t j = 0; j < n; j++) {
           result += matrix[i][j]*vector[j];
    }
     output[i] = result;
}
```

How well does vector processing map to super "scalar" processors

```
for(uint64_t i = 0; i < m; i++) {
     result = 0;
    for(uint64_t j = 0; j < n; j++) {
           result += matrix[i][j]*vector[j];
    }
      output[i] = result;
}
```



Assume we have a 5-issue INT, 3-load, 4-store pipeline like Alder Lake

Performance is very limited by both the memory bandwidth and available functional units

```
load vector[0]       load matrix[0][1]              load vector[3]                 load matrix[0][4]
load matrix[0][0]    load vector[2]                 load matrix[0][3]              load vector[5]
load vector[1]       load matrix[0][2]              load vector[4]                 load matrix[0][5]

                     mul matrix[0][0], vector[0]    mul matrix[0][1], vector[1]    mul matrix[0][3], vector[3]
                                                    mul matrix[0][2], vector[2]
```

Characteristics of vector processing

- Their operations are uniform across all pairs of elements

— Can we have just one instruction to control all pair-wise operations?

- There are very limited vector operations with mathematical meanings

— Can we simplify the processing elements design and make space for more PEs?

- They have very good spatial locality

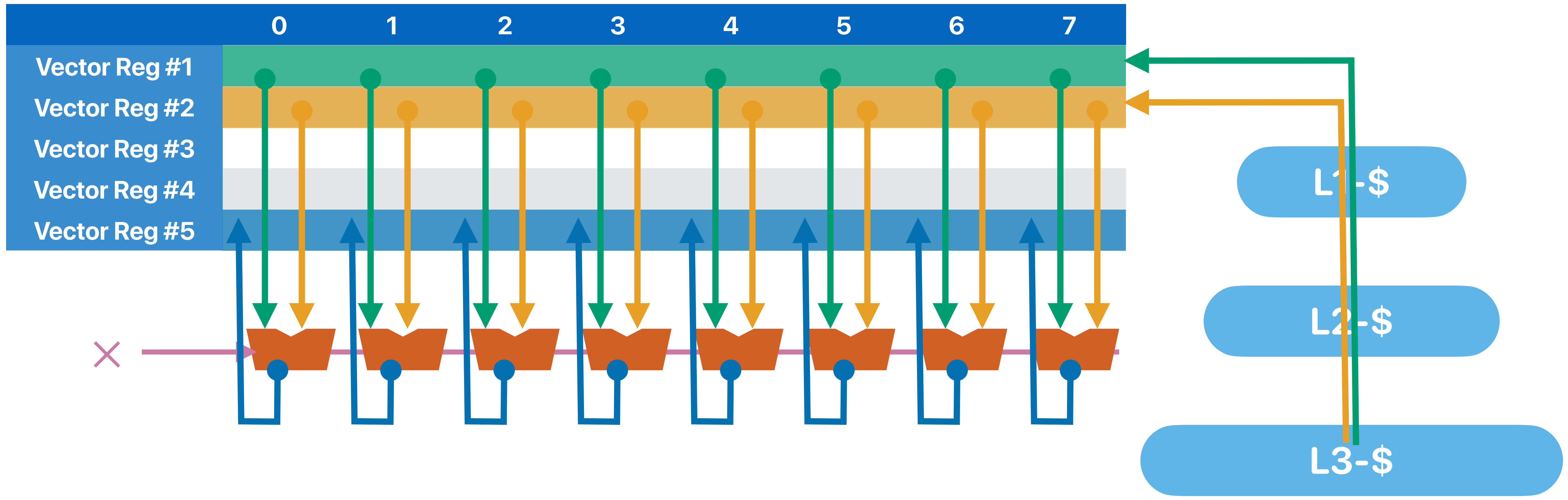
— Can we have fetch them once & put in wide registers?

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
    for(uint64_t j = 0; j < n; j++) {
          result += matrix[i][j]*vector[j];
    }
     output[i] = result;
}
```

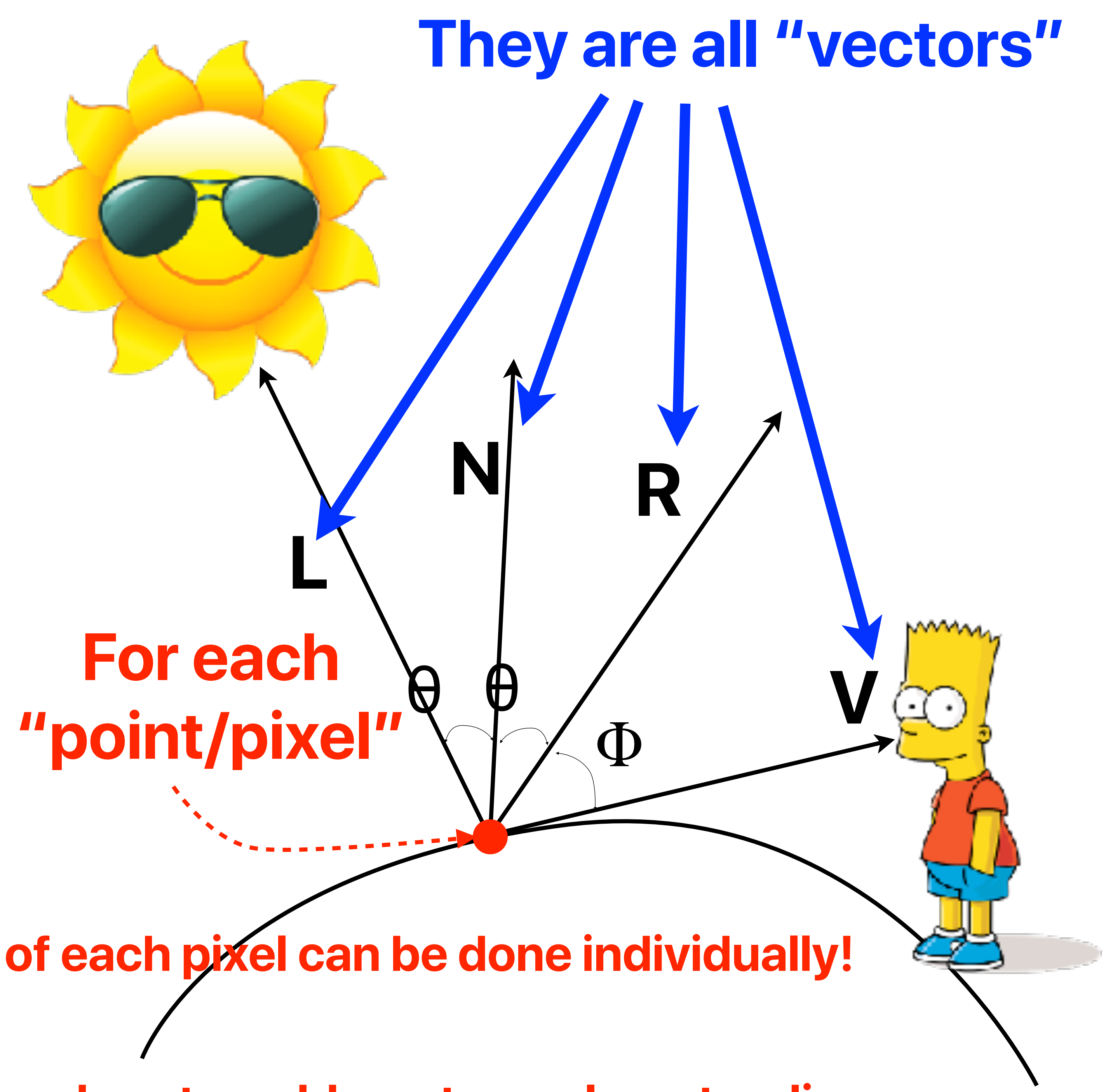
```
load vector[0]       load matrix[0][1]             load vector[3]              load matrix[0][4]
load matrix[0][0]    load vector[2]                load matrix[0][3]           load vector[5]
load vector[1]       load matrix[0][2]             load vector[4]              load matrix[0][5]

                     mul matrix[0][0], vector[0]   mul matrix[0][1], vector[1] mul matrix[0][3], vector[3]
                                                   mul matrix[0][2], vector[2]
```

Vector processing architecture



Basic concept of shading



The work of each pixel can be done individually!

We only need vector add, vector mul, vector div.

$$\boxed{\begin{aligned} I_{amb} &= K_{amb} \cdot M_{amb} \\ I_{diff} &= K_{diff} \cdot M_{diff} \cdot (N \cdot L) \\ I_{spec} &= K_{spec} \cdot M_{spec} \cdot (R \cdot V)^n \end{aligned}}$$

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

```
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)

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

https://www.pollev.com/hungweitseng close in 1:30

What GPU architectures should look like?

- Based on the original target of GPU design, what do you think would be reasonable design decisions that GPU architectures make?
  ① The architecture should render pixels as fast as possible
  ② The architecture does not need a branch unit
  ③ The architecture should contain an array of powerful ALUs and functional units that can perform a rich set of operations
  ④ The architecture should offer very large bandwidth of memory accesses
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

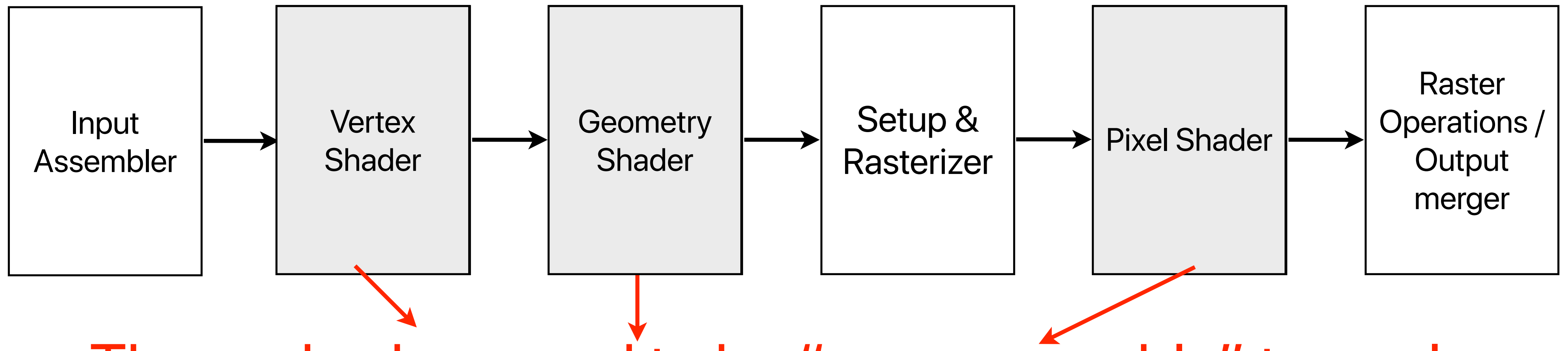
What GPU architectures should look like?

- Based on the original target of GPU design, what do you think would be reasonable design decisions that GPU architectures make?
  ① The architecture should render pixels as fast as possible
  ② The architecture does not need a branch unit
  ③ The architecture should contain an array of powerful ALUs and functional units that can perform a rich set of operations
  ④ The architecture should offer very large bandwidth of memory accesses
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

GPU (Graphics Processing Unit)

- Originally for displaying images
- HD video: 1920*1080 pixels * 60 frames per second
  - Therefore, GPU is not latency-oriented by design!
  - Even for 120 frames, you still have 8ms latency to get everything done!
- Graphics processing pipeline

1 GHz can give you 8000000 cycles!!!



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

What GPU architectures should look like?

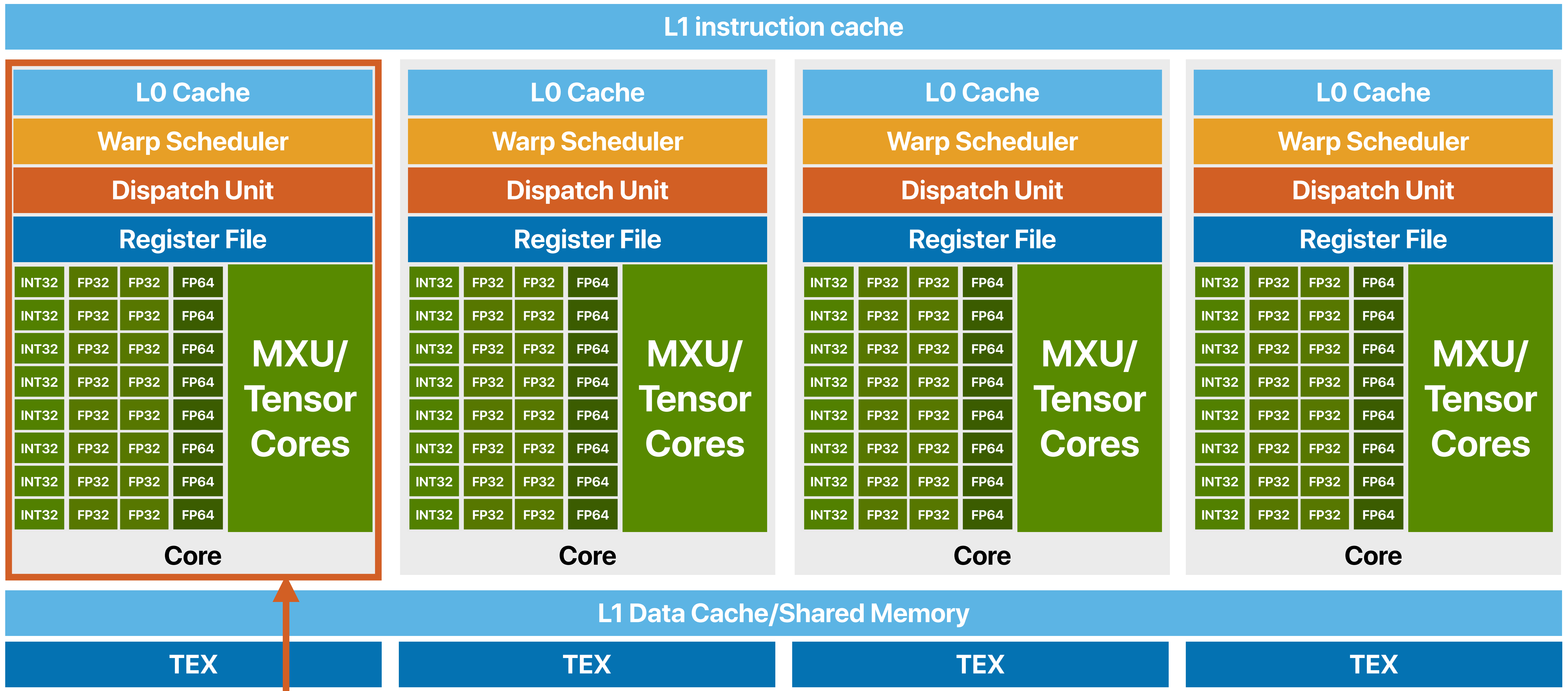
- Based on the original target of GPU design, what do you think would be reasonable design decisions that GPU architectures make?
  ① The architecture should render pixels as fast as possible
  [✓ ② The architecture does not need a branch unit]
  ③ The architecture should contain an array of powerful ALUs and functional units that can perform a rich set of operations
  [✓ ④ The architecture should offer very large bandwidth of memory accesses]

  A. 0
  B. 1
  [✓ C. 2]
  D. 3
  E. 4

What's the "appropriate" GPU architecture

- Lots of ALUs to process pixels in parallel — 2M pixels in HD resolution, very regular workloads
  - Vector processing model
- Simple operations
  - The ALUs only supports very few instructions
  - Almost no branches
- Deadline driven and throughput-oriented rather than latency oriented
  - High-bandwidth but also "higher-latency" memory
  - ALUs can be slower

GPU Architecture

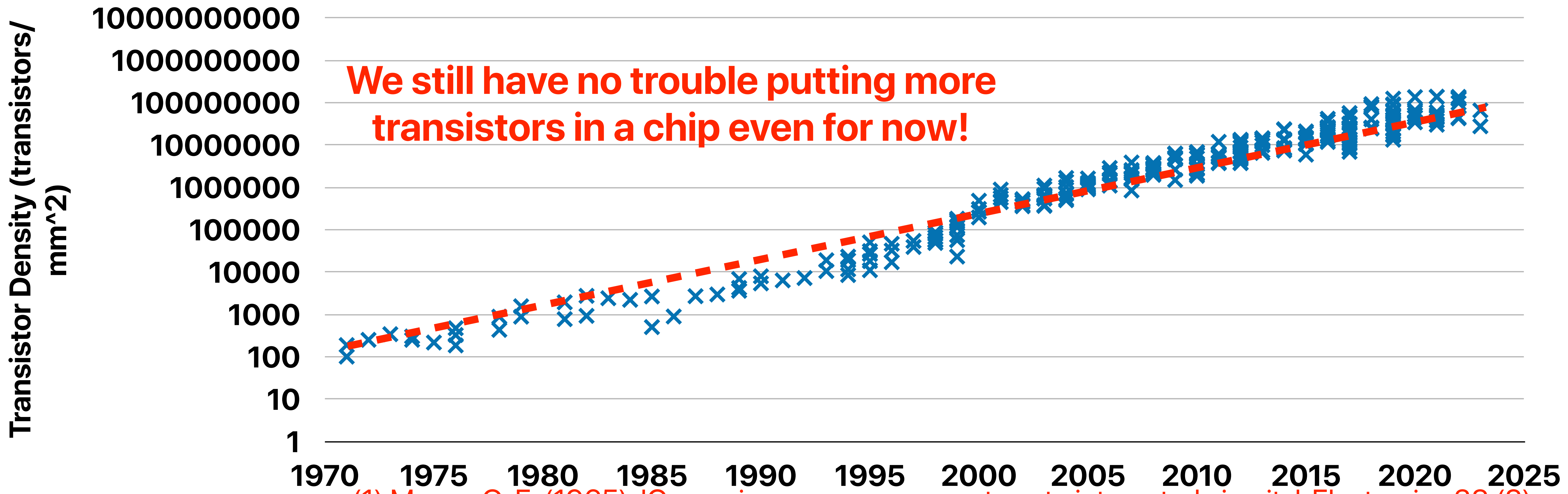


Each core is a "vector processing" unit

Dark silicon problem

Moore's Law[1]

- The number of transistors we can build in a fixed area of silicon doubles every 12 ~ 24 months.
- Moore's Law "was" the most important driver for historic CPU performance gains



(1) Moore, G. E. (1965), 'Cramming more components onto integrated circuits', Electronics 38 (8) .

Plot based on https://en.wikipedia.org/wiki/Transistor_count by Hung-Wei Tseng

https://www.pollev.com/hungweitseng close in 1:30

What happens if power doesn't scale with process technologies?

- If we are able to cram more transistors within the same chip area (Moore's law continues), but the power consumption per transistor remains the same. Right now, if put more transistors in the same area because the technology allows us to. How many of the following statements are true?
  ① The power consumption per chip will increase
  ② The power density of the chip will increase
  ③ Given the same power budget, we may not able to power on all chip area if we maintain the same clock rate
  ④ Given the same power budget, we may have to lower the clock rate of circuits to power on all chip area
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

Power consumption & power density

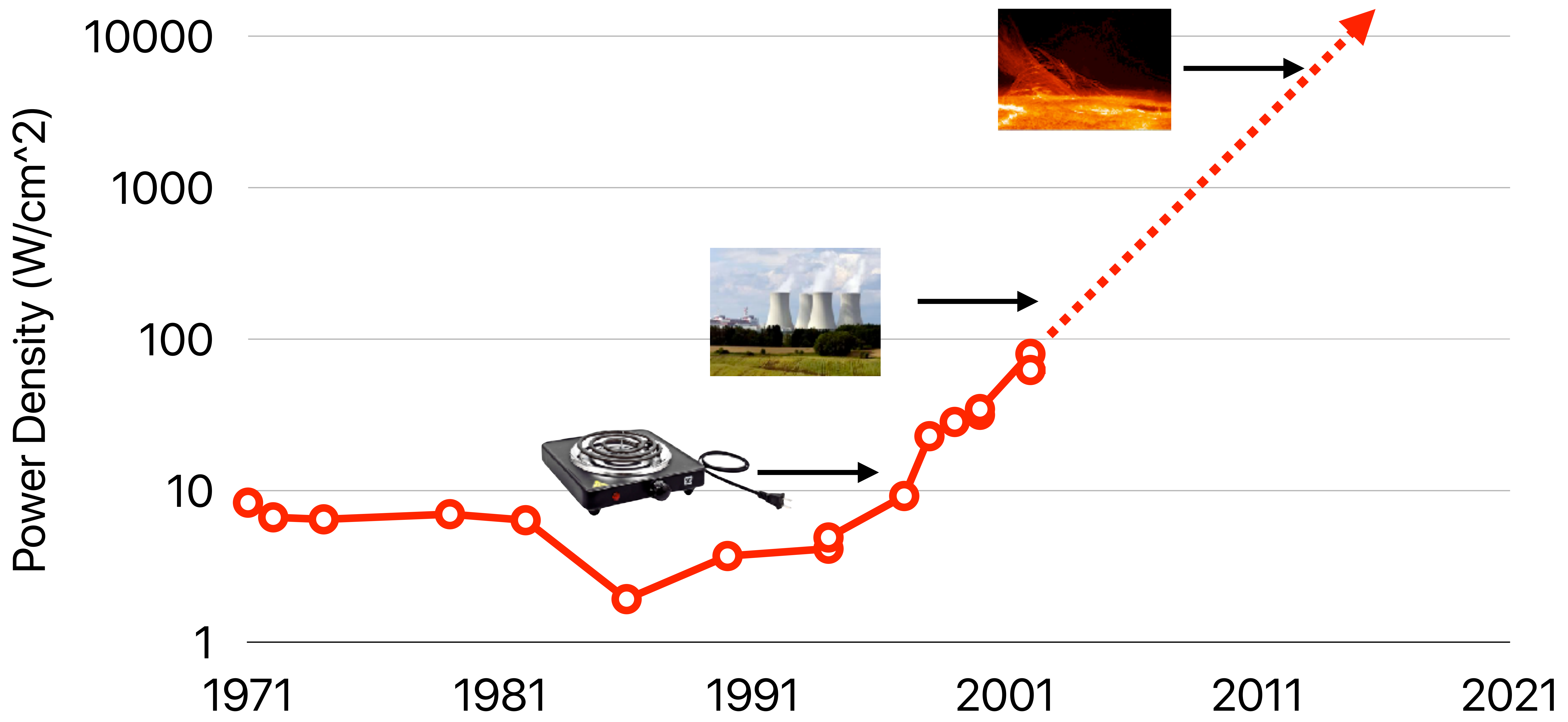
- $P_{dynamic} \sim \alpha \times C \times \boxed{V^2} \times f \times N$
  - α: average switches per cycle
  - C: capacitance
  - V: voltage
  - f: frequency, usually linear with V
  - N: the number of transistors
- $P_{leakage} \sim N \times \boxed{V} \times e^{-V_t}$
  - N: number of transistors
  - V: voltage
  - V_t: threshold voltage where transistor conducts (begins to switch)
- Power density:

$$P_{density} = \frac{P}{area}$$

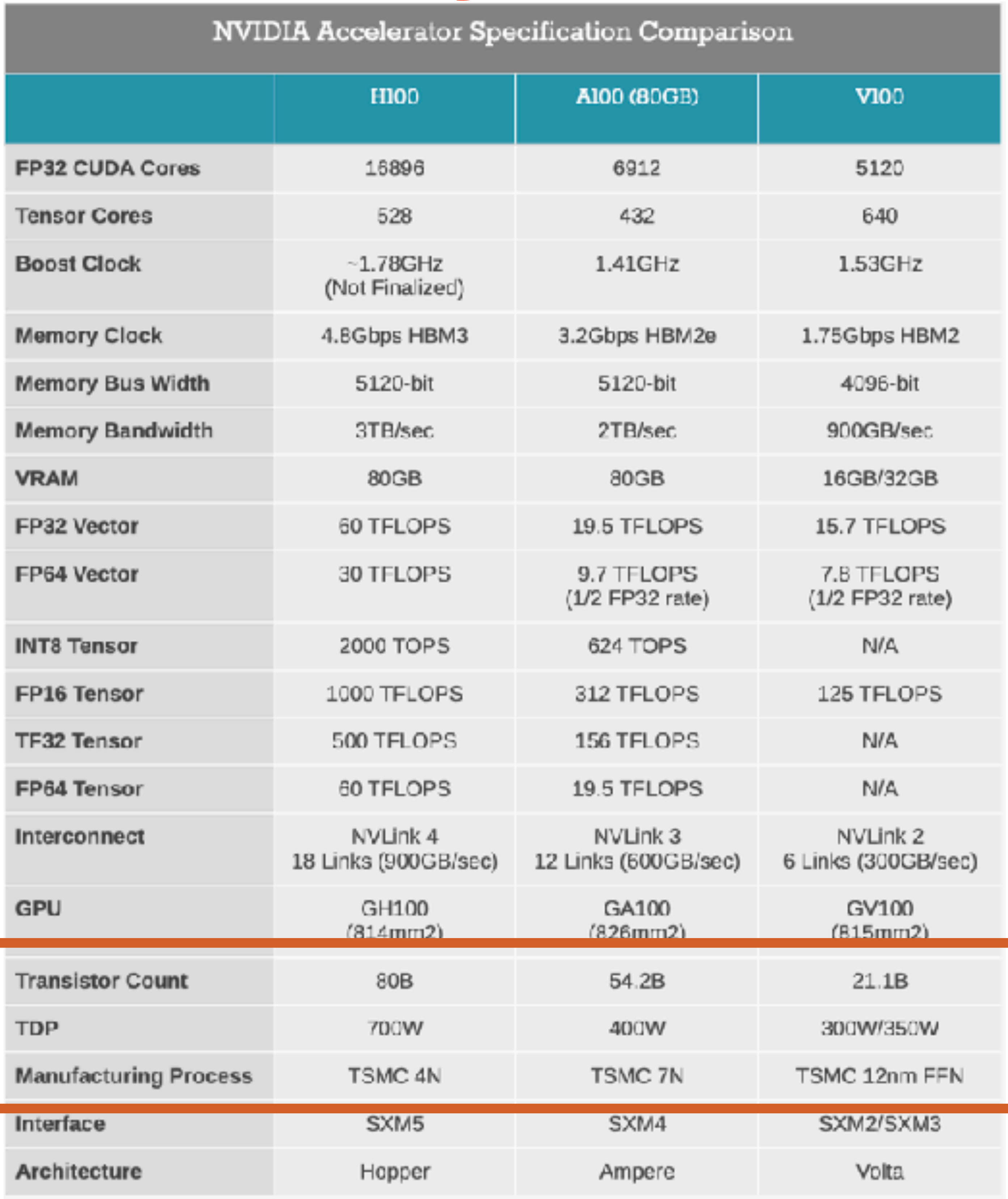
Voltage is a key factor of power consumption

Frequency is depending on the supply voltage

Power Density of Processors



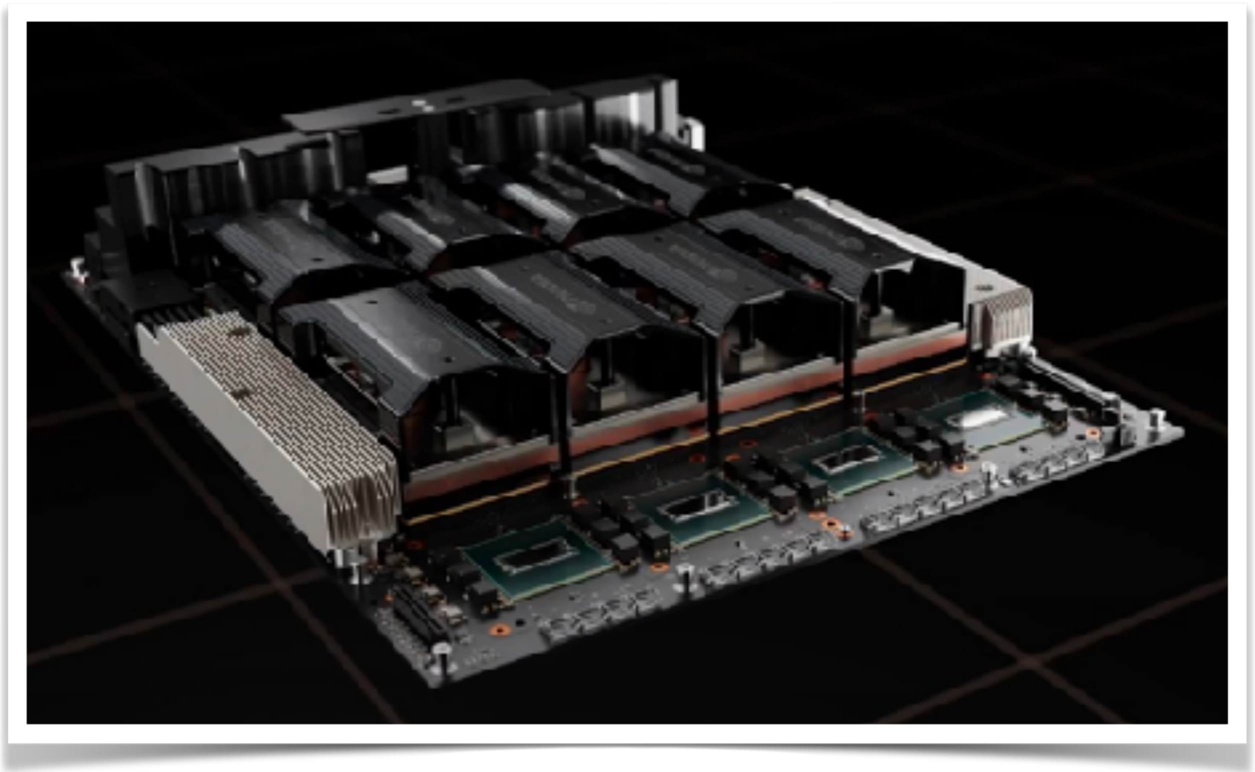
If you can add power budget...



NVIDIA Accelerator Specification Comparison

	H100	A100 (80GB)	V100
FP32 CUDA Cores	16896	6912	5120
Tensor Cores	528	432	640
Boost Clock	~1.78GHz (Not Finalized)	1.41GHz	1.53GHz
Memory Clock	4.8Gbps HBM3	3.2Gbps HBM2e	1.75Gbps HBM2
Memory Bus Width	5120-bit	5120-bit	4096-bit
Memory Bandwidth	3TB/sec	2TB/sec	900GB/sec
VRAM	80GB	80GB	16GB/32GB
FP32 Vector	60 TFLOPS	19.5 TFLOPS	15.7 TFLOPS
FP64 Vector	30 TFLOPS	9.7 TFLOPS (1/2 FP32 rate)	7.8 TFLOPS (1/2 FP32 rate)
INT8 Tensor	2000 TOPS	624 TOPS	N/A
FP16 Tensor	1000 TFLOPS	312 TFLOPS	125 TFLOPS
TF32 Tensor	500 TFLOPS	156 TFLOPS	N/A
FP64 Tensor	60 TFLOPS	19.5 TFLOPS	N/A
Interconnect	NVLink 4 18 Links (900GB/sec)	NVLink 3 12 Links (600GB/sec)	NVLink 2 6 Links (300GB/sec)
GPU	GH100 (814mm2)	GA100 (826mm2)	GV100 (815mm2)
Transistor Count	80B	54.2B	21.1B
TDP	700W	400W	300W/350W
Manufacturing Process	TSMC 4N	TSMC 7N	TSMC 12nm FFN
Interface	SXM5	SXM4	SXM2/SXM3
Architecture	Hopper	Ampere	Volta

https://www.workstationspecialist.com/product/nvidia-tesla-a100/



https://www.servethehome.com/wp-content/uploads/2022/03/NVIDIA-GTC-2022-H100-in-HGX-H100.jpg

Power consumption to light on all transistors

Chip

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1

=49W

Dennardian Scaling

Chip

0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

=50W

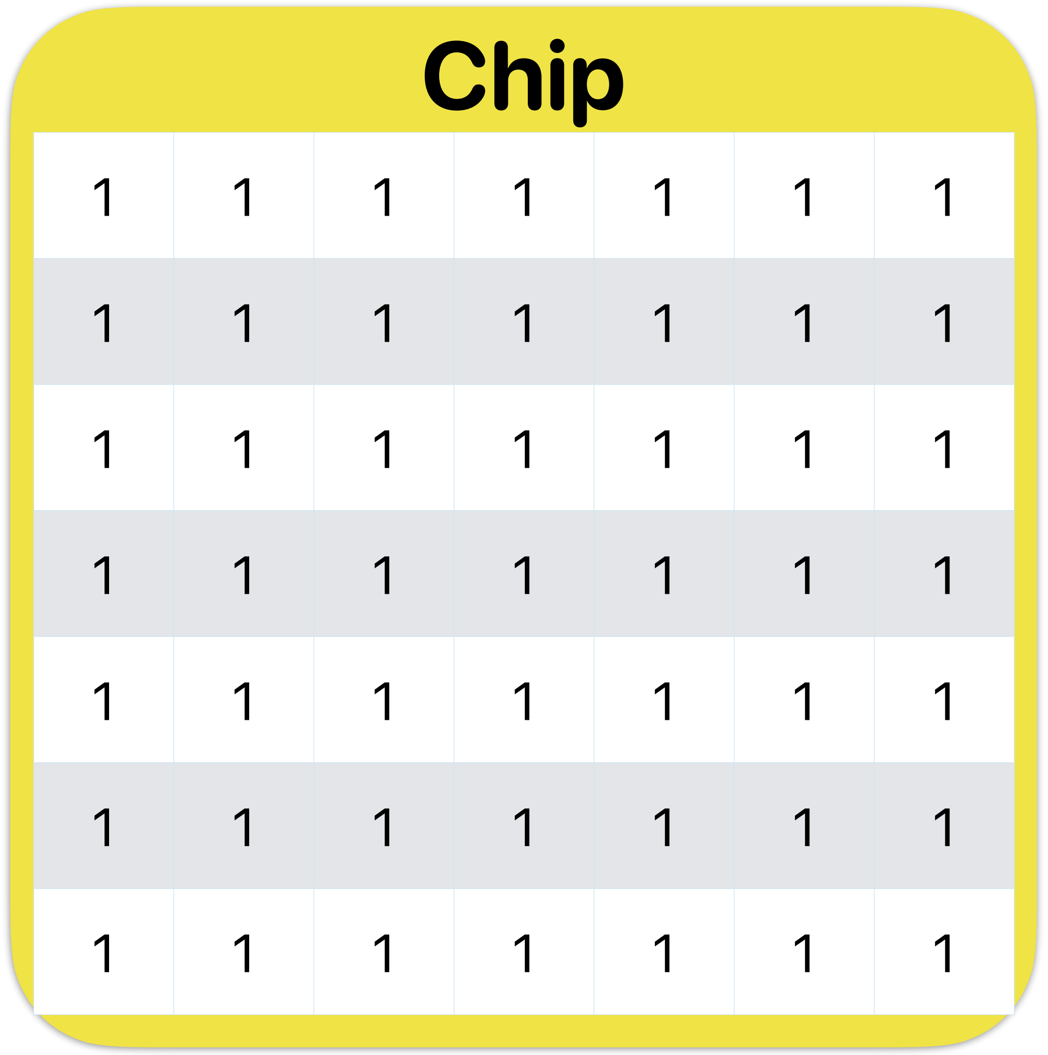
Dennardian Broken

Chip

1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

=100W!

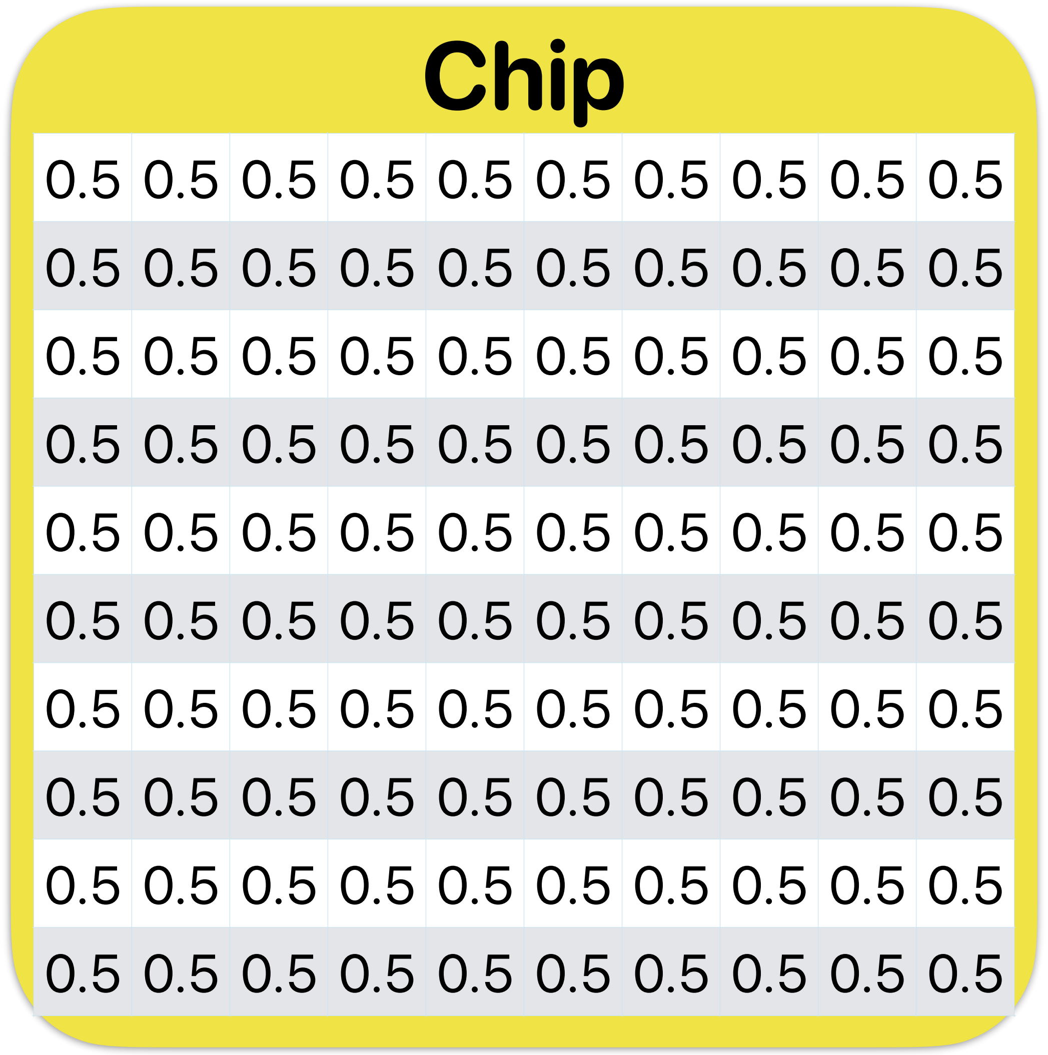
Power consumption to light on all transistors



1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1

=49W

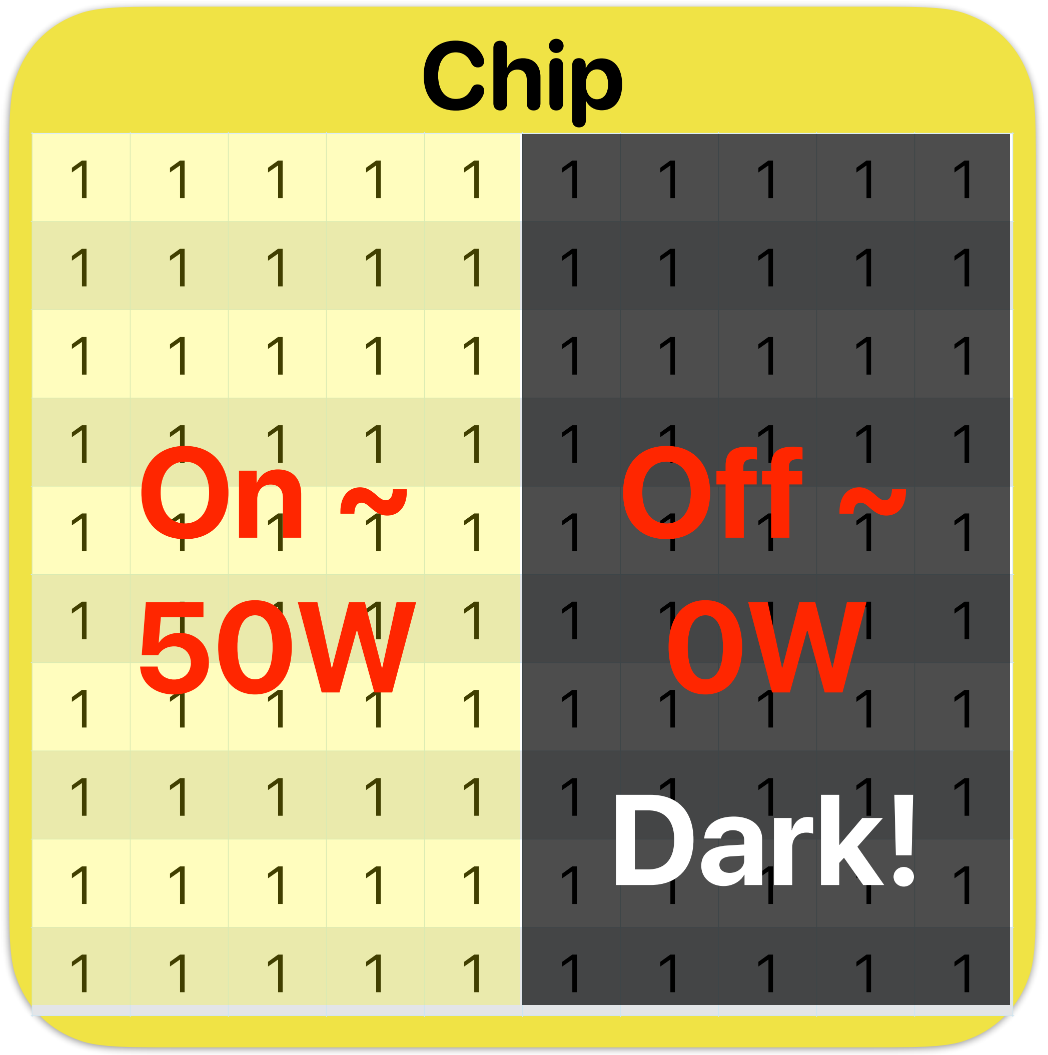
Dennardian Scaling



0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

=50W

Dennardian Broken



1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

=100W!

What happens if power doesn't scale with process technologies?

- If we are able to cram more transistors within the same chip area (Moore's law continues), but the power consumption per transistor remains the same. Right now, if put more transistors in the same area because the technology allows us to. How many of the following statements are true?
  - ① The power consumption per chip will increase
  - ② The power density of the chip will increase
  - ③ Given the same power budget, we may not able to power on all chip area if we maintain the same clock rate
  - ④ Given the same power budget, we may have to lower the clock rate of circuits to power on all chip area

  A. 0
  B. 1
  C. 2
  D. 3
  [✓ E. 4]

Announcements

- **Course evaluation** started and ends on 9/05/2024
  - Submit the prove of your participation in course evaluation through Gradescope
  - It can become a full credit reading quiz (it helps to amortize the penalty of another least performing one) — we will drop a total of three now
- **Assignment 5** is **due 9/05/2024**
  - The due date is earlier to allow publication of solutions before the deadline
- **Final exam**
  - 9/6 3p-6p @ **PCH 121**
  - Closed book, no cheatsheet — the same rules as the midterm
  - Pizza party after the examine

Computer
Science &
Engineering

つづく