Synchronization 3 and GPUs

CS 3410, Spring 2014

Computer Science

Cornell University

See P&H Chapter: 6.7

Administrivia

Next 3 weeks

- Prelim2 Thu May 1st : 7:30-9:30
  - Olin 155: Netid [a-g]*
  - Uris G01: Netid [h-z]*
- Proj3 tournament: Mon May 5 5pm-7pm (Pizza!)
- Proj4 design doc meetings May 5-7 (doc ready for mtg)

Final Project for class

- Proj4 due Wed May 14
- Proj4 demos: May 13 and 14
- Proj 4 release: in labs this week
- Remember: No slip days for PA4

Academic Integrity

All submitted work must be your own

- OK to study together, but do not share soln's
- Cite your sources

Project groups submit joint work

- Same rules apply to projects at the group level
- Cannot use of someone else's soln

Closed-book exams, no calculators

- Stressed? Tempted? Lost?
  - Come see us before due date!

Plagiarism in any form will not be tolerated

Synchronization

- Threads
- Critical sections, race conditions, and mutexes
- Atomic Instructions
  - HW support for synchronization
  - Using sync primitives to build concurrency-safe data structures
- Example: thread-safe data structures
- Language level synchronization
- Threads and processes

Synchronization in MIPS

Load linked: `LL rt, offset(rs)`

Store conditional: `SC rt, offset(rs)`

- Succeeds if location not changed since the LL
  - Returns 1 in rt
- Fails if location is changed
  - Returns 0 in rt

Any time a processor intervenes and modifies the value in memory between the LL and SC instruction, the SC returns 0 in $t0

Use this value 0 to try again

Mutex from LL and SC

Linked load / Store Conditional

m = 0; // 0 means lock is free; otherwise, if m ==1, then lock locked

```
mutex_lock(int m) {
   while(test_and_set(&m)){}
}

int test_and_set(int *m) {
     old = *m;     LL     Atomic
     *m = 1;       SC
     return old;
}
```

Mutex from LL and SC

Linked load / Store Conditional

```
m = 0;
mutex_lock(int *m) {
   while(test_and_set(m)){}
}
int test_and_set(int *m) {
 try:
      LI $t0, 1
      LL $t1, 0($a0)
      SC $t0, 0($a0)
      BEQZ $t0, try
      MOVE $v0, $t1
}
```

Mutex from LL and SC

```
m = 0;
mutex_lock(int *m) {
   test_and_set:
           LI $t0, 1
           LL $t1, 0($a0)
           BNEZ $t1, test_and_set
           SC $t0, 0($a0)
           BEQZ $t0, test_and_set
}
mutex_unlock(int *m) {
     *m = 0;
}
```

Mutex from LL and SC

```
m = 0;
mutex_lock(int *m) {
   test_and_set:
           LI $t0, 1
           LL $t1, 0($a0)
           BNEZ $t1, test_and_set
           SC $t0, 0($a0)
           BEQZ $t0, test_and_set
}
mutex_unlock(int *m) {
      SW $zero, 0($a0)
}
```

This is called a
Spin lock
Aka spin waiting

Mutex from LL and SC

```
m = 0;
mutex_lock(int *m) {
```

Time Step	Thread A	Thread B	Thread A $t0	Thread A $t1	Thread B $t0	Thread B $t1	Mem M[$a0]
0							0
1	try: LI $t0, 1	try: LI $t0, 1					
2	LL $t1, 0($a0)	LL $t1, 0($a0)					
3	BNEZ $t1, try	BNEZ $t1, try					
4	SC $t0, 0($a0)	SC $t0, 0 ($a0)					
5	BEQZ $t0, try	BEQZ $t0, try					
6							

Mutex from LL and SC

m = 0;

```
mutex_lock(int *m) {
```

Time Step	Thread A	Thread B	Thread A $t0	Thread A $t1	Thread B $t0	Thread B $t1	Mem M[$a0]
0							0
1	try: LI $t0, 1	try: LI $t0, 1	1		1		0
2	LL $t1, 0($a0)	LL $t1, 0($a0)	1	0	1	0	0
3	BNEZ $t1, try	BNEZ $t1, try	1	0	1	0	0
4	SC $t0, 0($a0)	SC $t0, 0 ($a0)	0	0	1	0	1
5	BEQZ $t0, try	BEQZ $t0, try	0	0	1	0	1
6							

Mutex from LL and SC

m = 0;

```
mutex_lock(int *m) {
```

Time Step	Thread A	Thread B	Thread A $t0	Thread A $t1	Thread B $t0	Thread B $t1	Mem M[$a0]
0							0
1	try: LI $t0, 1	try: LI $t0, 1	1		1		0
2	LL $t1, 0($a0)	LL $t1, 0($a0)	1	0	1	0	0
3	BNEZ $t1, try	BNEZ $t1, try	1	0	1	0	0
4	SC $t0, 0($a0)	SC $t0, 0 ($a0)	0	0	1	0	1
5	BEQZ $t0, try	BEQZ $t0, try	0	0	1	0	1
6	try: LI $t0, 1	Critical section					

Summary

Need parallel abstraction like for multicore

Writing correct programs is hard

Need to prevent data races

Need critical sections to prevent data races

Mutex, mutual exclusion, implements critical section

Mutex often implemented using a lock abstraction

Hardware provides synchronization primitives such as LL and SC (load linked and store conditional) instructions to efficiently implement locks

Topics

Synchronization

- Threads
- Critical sections, race conditions, and mutexes
- Atomic Instructions
  - HW support for synchronization
  - Using sync primitives to build concurrency-safe data structures
- Example: thread-safe data structures
- Language level synchronization
- Threads and processes

Next Goal

How do we use synchronization primitives to build concurrency-safe data structure?

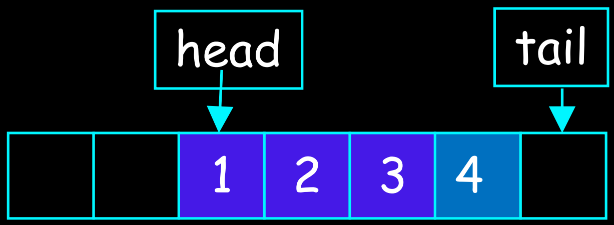
Let’s look at a ring buffer

Attempt#1: Producer/Consumer

Access to shared data must be synchronized

- goal: enforce datastructure invariants

```
// invariant:
// data is in A[h … t-1]
char A[100];
int h = 0, t = 0;
```




```
// producer: add to list tail
void put(char c) {
  // Need: check if list full
  A[t] = c;
  t = (t+1)%n;
}
```

```
// consumer: take from list head
char get() {
  while (empty()) { };
  char c = A[h];
  h = (h+1)%n;
  return c;
}
```

Testing the invariant

Various ways to implement empty()

1. h==t

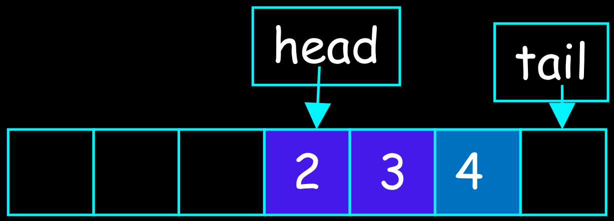
   but then the put has to be careful

2. could be a separate boolean

…

Attempt#1: Producer/Consumer

```
// invariant:
// data is in A[h … t-1]
char A[100];
int h = 0, t = 0;
```




```
// producer: add to list tail
void put(char c) {
  A[t] = c;
  t = (t+1)%n;
}
```

```
// consumer: take from list head
char get() {
  while (empty()) { };
  char c = A[h];
  h = (h+1)%n;
  return c;
}
```

Error: could miss an update to ***t*** or ***h*** due to lack of synchronization

Current implementation will **break invariant:**

only produce if not full and only consume if not empty

Need to synchronize access to shared data

Attempt#2: Protecting an invariant

```
// invariant: (protected by mutex m)
// data is in A[h … t-1]
pthread_mutex_t *m = pthread_mutex_create();
char A[100];
int h = 0, t = 0;
```

```
// consumer: take from list head
char get() {
  pthread_mutex_lock(m);
  while(empty()) {}
  char c = A[h];
  h = (h+1)%n;
  pthread_mutex_unlock(m);
  return c;
}
```

Rule of thumb: all access and updates that can affect invariant become critical sections

Attempt#2: Protecting an invariant

```
// invariant: (protected by mutex m)
// data is in A[h … t-1]
pthread_mutex_t *m = pthread_mutex_create();
char A[100];
int h = 0, t = 0;
```

BUG: Can’t wait while holding lock

```
// consumer: take from list head
char get() {
  pthread_mutex_lock(m);
  while(empty()) {}
  char c = A[h];
  h = (h+1)%n;
  pthread_mutex_unlock(m);
  return c;
}
```

Rule of thumb: all access and updates that can affect invariant become critical sections

Guidelines for successful mutexing

Insufficient locking can cause races

- Skimping on mutexes? Just say no!

But poorly designed locking can cause deadlock

```
P1: lock(m1);     P2: lock(m2);
    lock(m2);         lock(m1);
```

Circular Wait

- Know why you are using mutexes!
- Acquire locks in a consistent order to avoid cycles
- Use lock/unlock like braces (match them lexically)
  - lock(&m); …; unlock(&m)
  - Watch out for return, goto, and function calls!
  - Watch out for exception/error conditions!

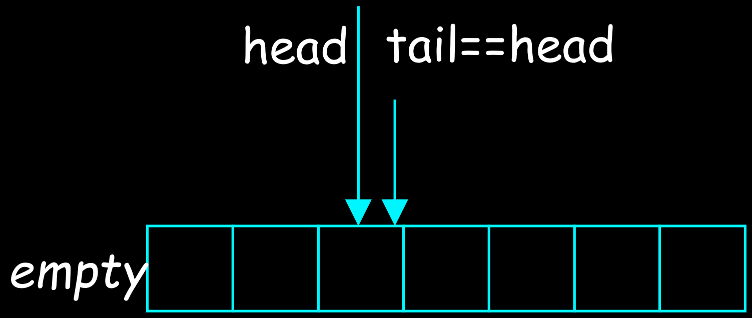
Attempt#3: Beyond mutexes

Writers must check for full buffer
& Readers must check if for empty buffer

- ideal: don’t busy wait… go to sleep instead

```
char get() {
  while (empty()) { };
  lock (L);
  char c = A[h];
  h = (h+1)%n;
  unlock (L);
  return c;
}
```

Cannot check condition while
Holding the lock,
BUT, empty condition may no
longer hold in critical section



Dilemma: Have to check while holding lock

Attempt#3: Beyond mutexes

Writers must check for full buffer
& Readers must check if for empty buffer

- ideal: don't busy wait… go to sleep instead

```
char get() {
  lock (L);
  while (empty()) { };
  char c = A[h];
  h = (h+1)%n;
  unlock (L);
  return c;
}
```

Dilemma: Have to check while holding lock,
but cannot wait while holding lock

Attempt#4: Beyond mutexes

Writers must check for full buffer
& Readers must check if for empty buffer

- ideal: don't busy wait… go to sleep instead

```
char get() {
  do {
      lock (L);
      if (!empty()) {
          c = A[h];
          h = (h+1)%n;
      }
      unlock (L);
  } while (empty);
  return c;
}
```

Language-Level Synchronization

Condition variables

- Wait for condition to be true
- Thread sleeps while waiting
- Can wake up one thread or all threads

Monitors, …

Summary

Hardware Primitives: test-and-set, LL/SC, barrier, ...

... used to build ...

Synchronization primitives: mutex, semaphore, ...

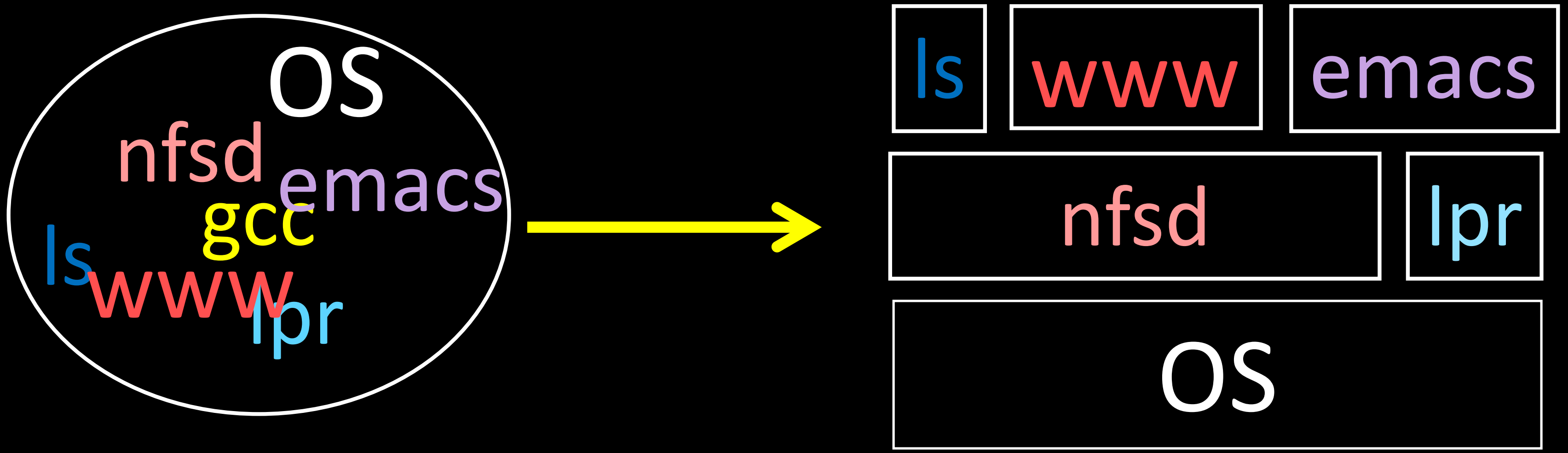
... used to build ...

Language Constructs: monitors, signals, ...

PRELIM 2 CONTENT TILL HERE

Abstraction of Processes

How do we cope with lots of activity?



Simplicity? Separation into processes

Reliability? Isolation

Speed? Program-level parallelism

Process and Program

Process

OS abstraction of a running computation

- The unit of execution
- The unit of scheduling
- Execution state
  + address space

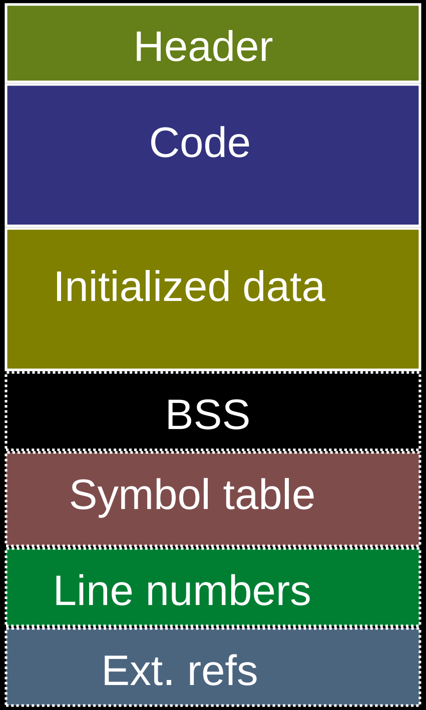
From process perspective

- a virtual CPU
- some virtual memory
- a virtual keyboard, screen, …

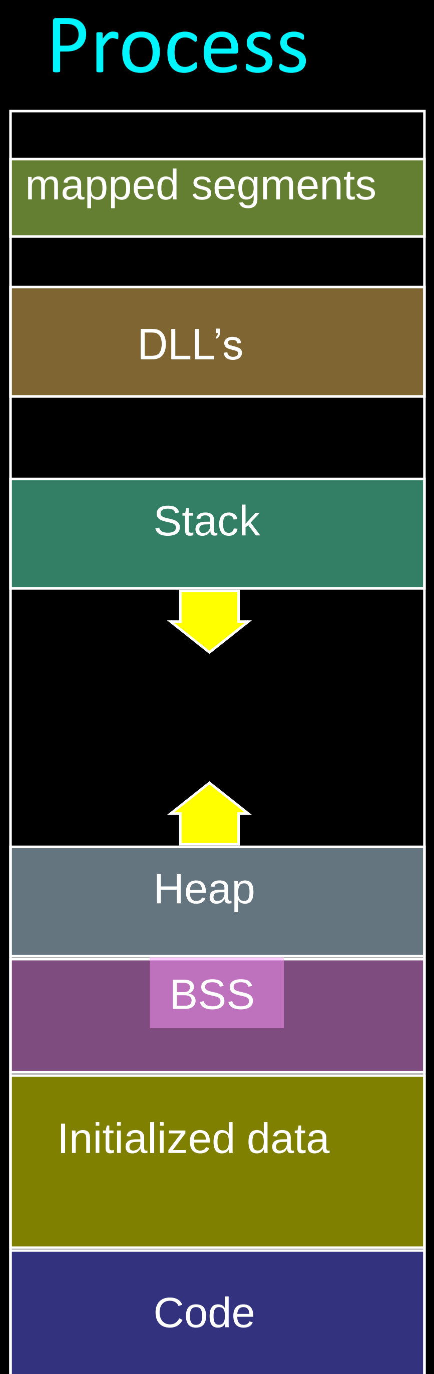
Program

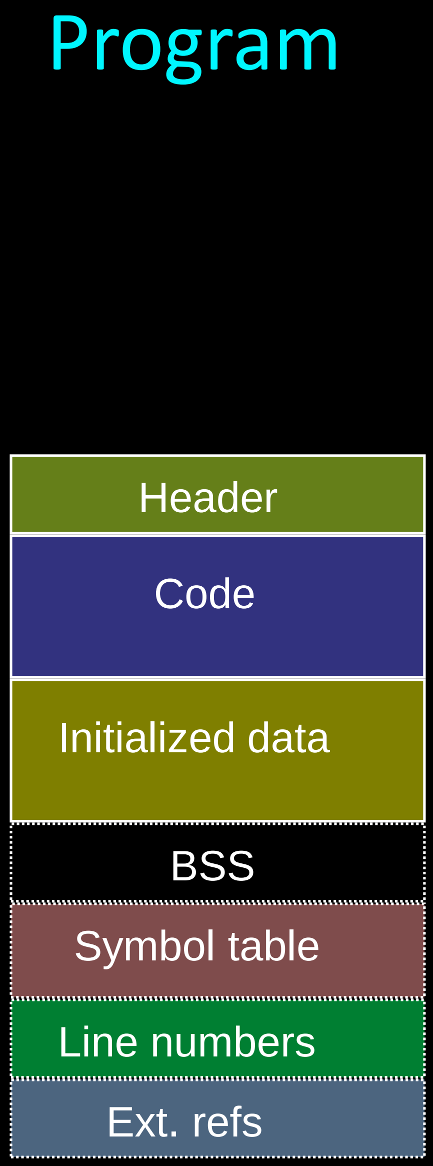
"Blueprint" for a process

- Passive entity (bits on disk)
- Code + static data



Process and Program





Role of the OS

Role of the OS

Context Switching

- Provides illusion that every process owns a CPU

Virtual Memory

- Provides illusion that process owns some memory

Device drivers & system calls

- Provides illusion that process owns a keyboard, …

To do:

How to start a process?

How do processes communicate / coordinate?

How to create a process?

Q: How to create a process?

A: Double click

After boot, OS starts the first process

…which in turn creates other processes

- parent / child → the process tree

pstree example

```
$ pstree | view -
init-+-NetworkManager-+-dhclient
     |-apache2
     |-chrome-+-chrome
     |        `-chrome
     |-chrome---chrome
     |-clementine
     |-clock-applet
     |-cron
     |-cupsd
     |-firefox---run-mozilla.sh---firefox-bin-+-plugin-cont
     |-gnome-screensaver
     |-grep
     |-in.tftpd
     |-ntpd
     `-sshd---sshd---sshd---bash-+-gcc---gcc---cc1
                                 |-pstree
                                 |-vim
                                 `-view
```

Processes Under UNIX

Init is a special case. For others…

Q: How does parent process create child process?

A: fork() system call

int fork() returns TWICE!

Example

```
main(int ac, char **av) {
      int x = getpid(); // get current process ID from OS
      char *hi = av[1]; // get greeting from command line
      printf("I'm process %d\n", x);
      int id = fork();
      if (id == 0)
            printf("%s from %d\n", hi, getpid());
      else
      printf("%s from %d, child is %d\n", hi, getpid(), id);
}
$ gcc -o strange strange.c
$ ./strange "Hey"
I'm process 23511
Hey from 23512
Hey from 23511, child is 23512
```

Inter-process Communication

Parent can pass information to child

- In fact, *all parent data* is passed to child
- But isolated after (copy-on-write ensures changes are invisible)

Q: How to continue communicating?

A: Invent OS “IPC channels” : send(msg), recv(), …

Inter-process Communication

Parent can pass information to child

- In fact, *all parent data* is passed to child
- But isolated after (C-O-W ensures changes are invisible)

Q: How to continue communicating?

A: Shared (Virtual) Memory!

Processes and Threads

Processes and Threads

Process

OS abstraction of a running computation

- The unit of execution
- The unit of scheduling
- Execution state
  + address space

From process perspective

- a virtual CPU
- some virtual memory
- a virtual keyboard, screen, …

Thread

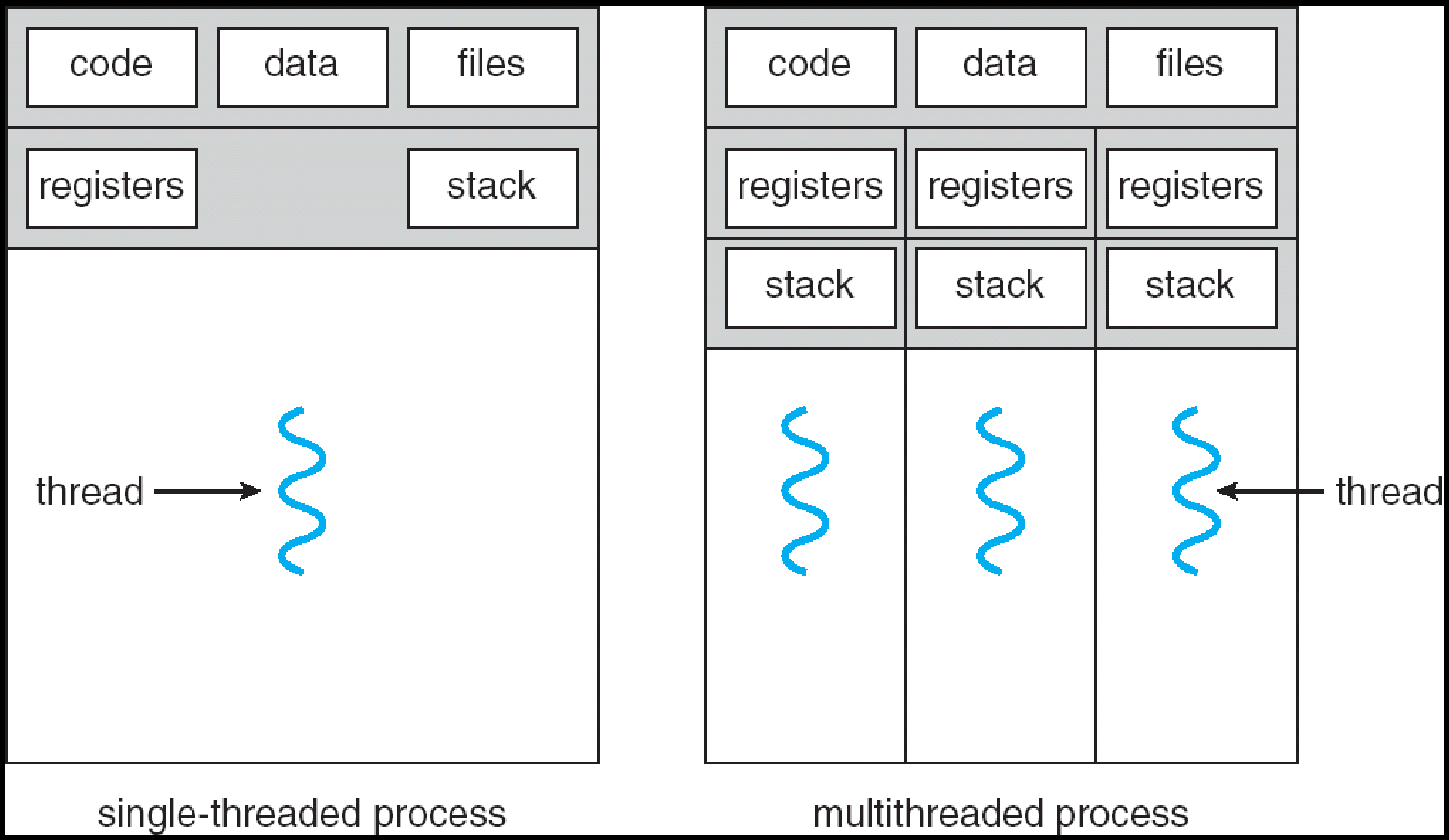
OS abstraction of a single thread of control

- The unit of scheduling
- Lives in one single process

From thread perspective

- one virtual CPU core on a virtual multi-core machine

Multithreaded Processes



Threads

```
#include <pthread.h>
int counter = 0;

void PrintHello(int arg) {
      printf("I'm thread %d, counter is %d\n", arg, counter++);
      ... do some work ...
      pthread_exit(NULL);
}

int main () {
   for (t = 0; t < 4; t++) {
      printf("in main: creating thread %d\n", t);
      pthread_create(NULL, NULL, PrintHello, t);
   }
   pthread_exit(NULL);
}
```

Threads versus Fork

```
in main: creating thread 0
I’m thread 0, counter is 0
in main: creating thread 1
I’m thread 1, counter is 1
in main: creating thread 2
in main: creating thread 3
I’m thread 3, counter is 2
I’m thread 2, counter is 3
```

Summary

Processes and Threads are the abstraction that we use to write parallel programs

Fork and Join and Interprocesses communication (IPC) can be used to coordinate processes

Threads are used to coordinate use of shared memory within a process

GPUs

The supercomputer in your laptop

GPU: Graphics processing unit

Very basic till about 1999

Specialized device to accelerate display

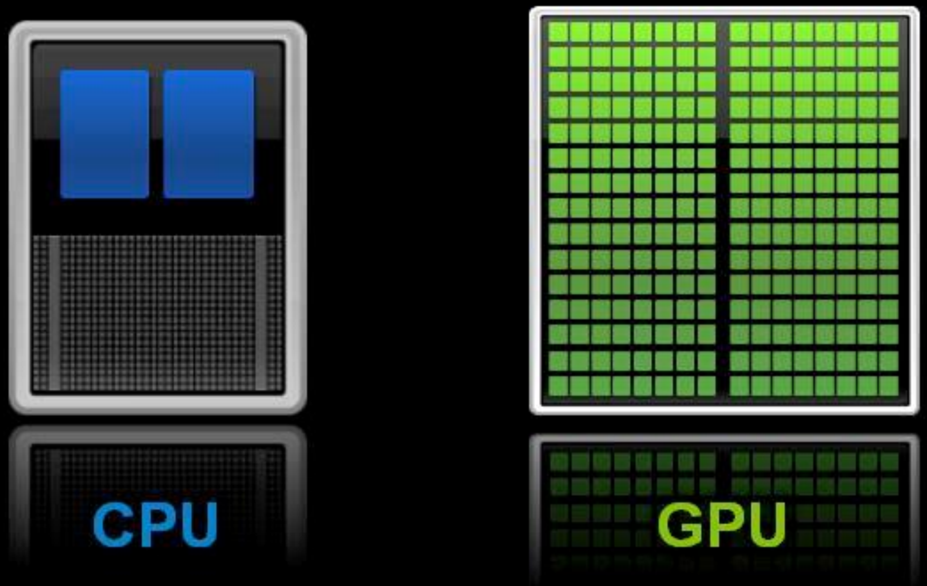
Then started changing into a full processor

2000-…: Frontier times

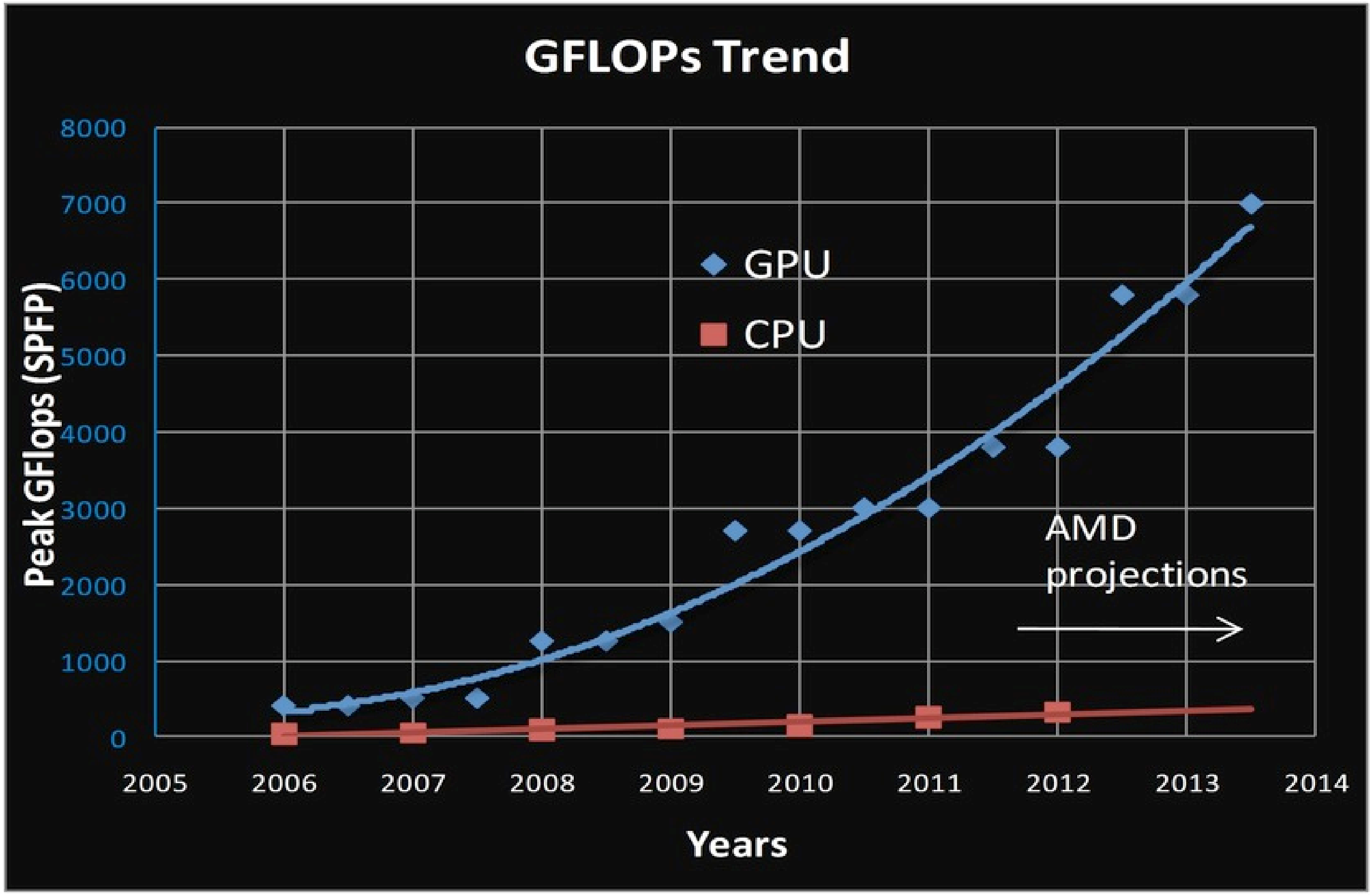
Parallelism

CPU: Central Processing Unit

GPU: Graphics Processing Unit



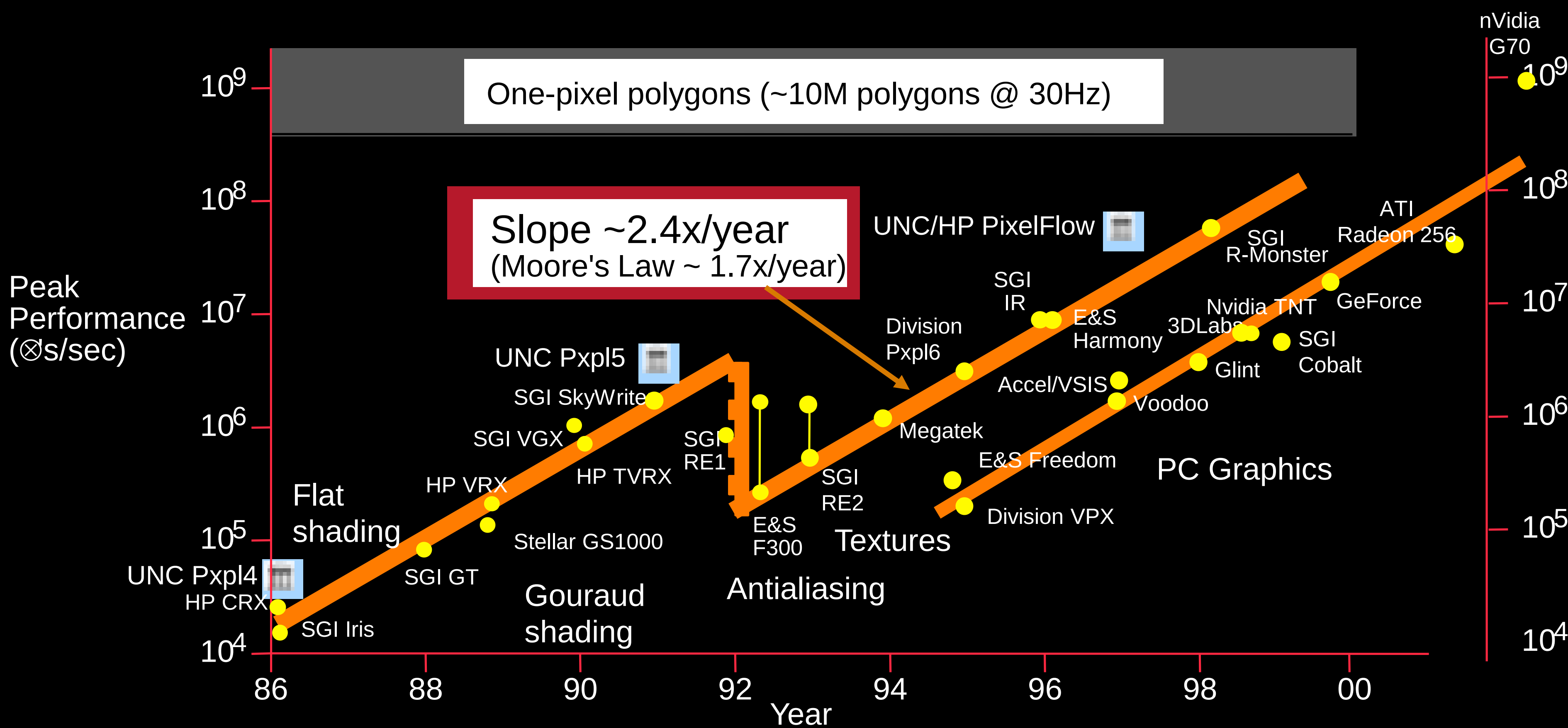
GPU-type computation offers higher GFlops



(Source: Sam Naffziger, AMD)

Fig. 4

GPUs: Faster than Moore's Law
Moore's Law is for Wimps?!

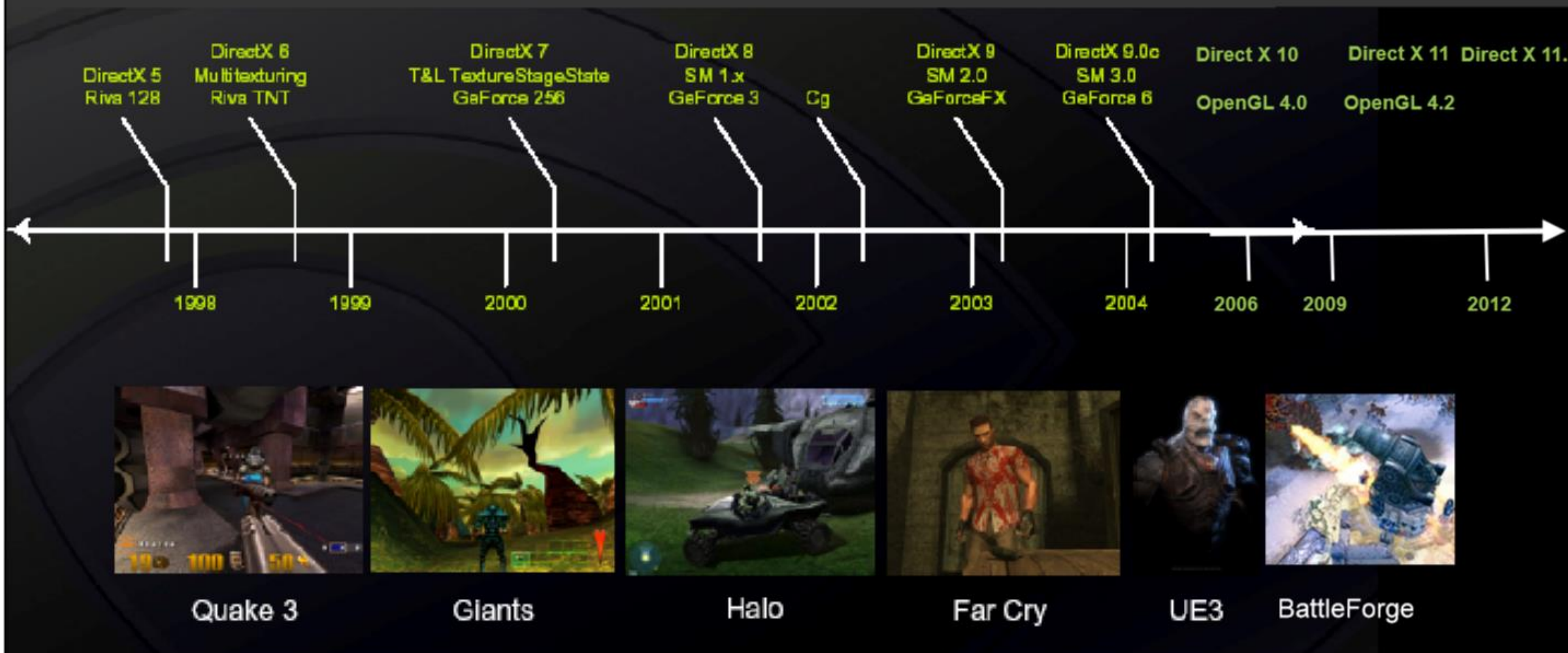


Graph courtesy of Professor John Poulton (from Eric Haines)

Programmable Hardware

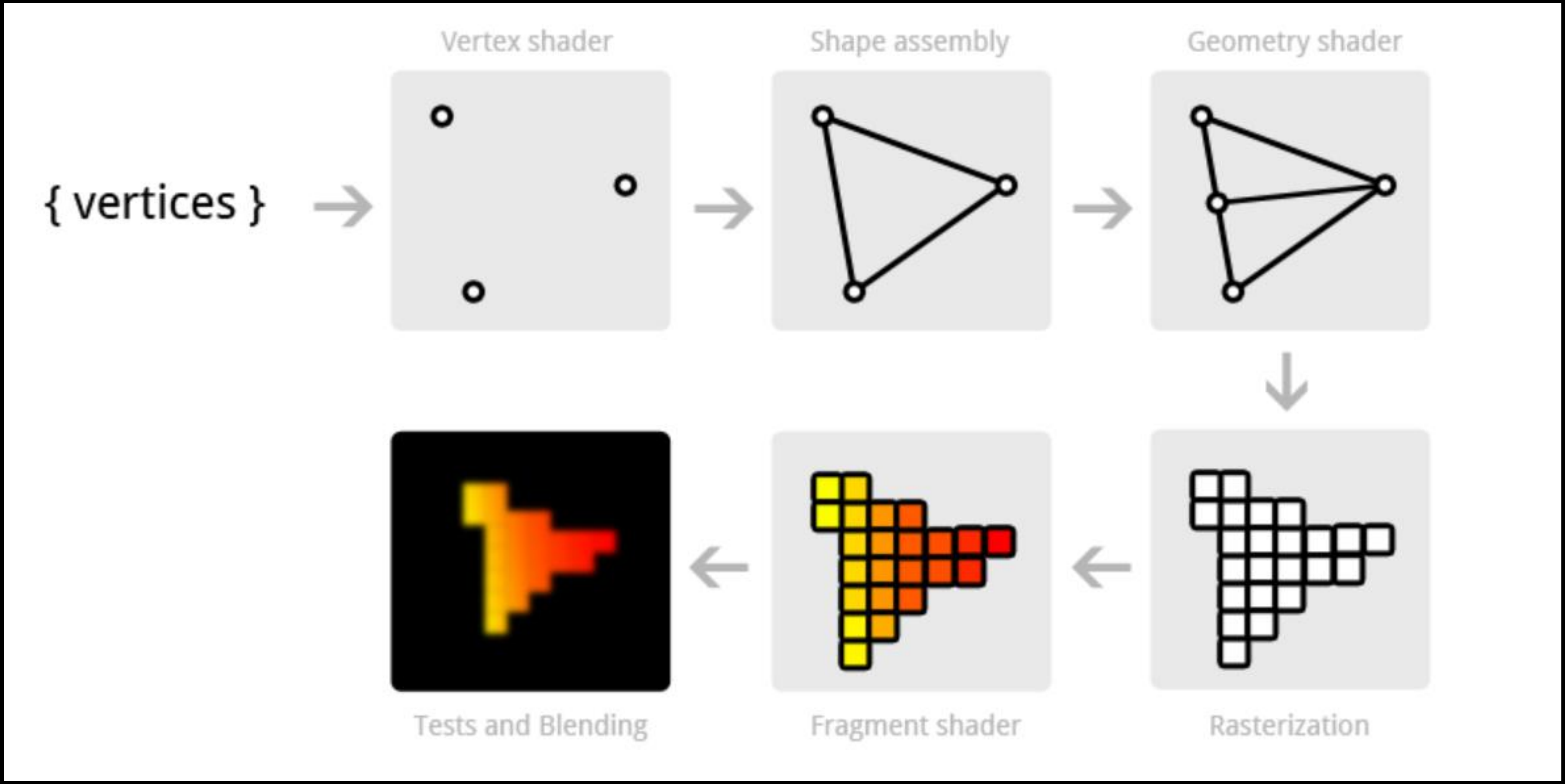
- Started in 1999
- Flexible, programmable
  - Vertex, Geometry, Fragment Shaders
- And much faster, of course
  - 1999 GeForce256: 0.35 Gigapixel peak fill rate
  - 2001 GeForce3: 0.8 Gigapixel peak fill rate
  - 2003 GeForceFX Ultra: 2.0 Gigapixel peak fill rate
  - ATI Radeon 9800 Pro : 3.0 Gigapixel peak fill rate
  - 2006 NV60: ... Gigapixel peak fill rate
  - 2009 GeForce GTX 285: 10 Gigapixel peak fill rate
  - 2011
    - GeForce GTC 590: 56 Gigapixel peak fill rate
    - Radeon HD 6990: 2x26.5
  - 2012
    - GeForce GTC 690: 62 Gigapixel/s peak fill rate

Evolution of GPU



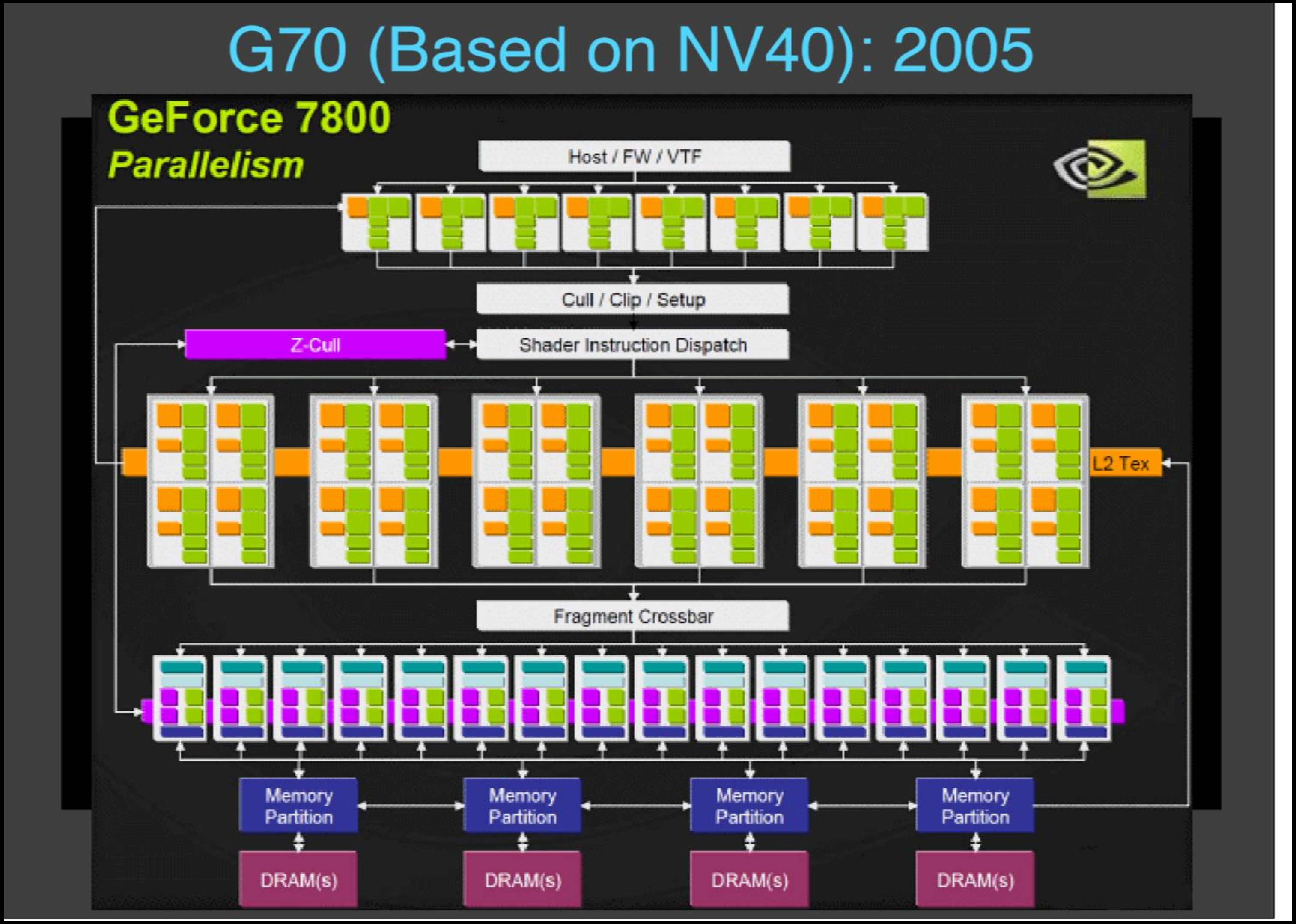
Around 2000

Fixed function pipeline



Around 2005

Programmable vertex and pixel processors



Post 2006: Unified Architecture

Why?

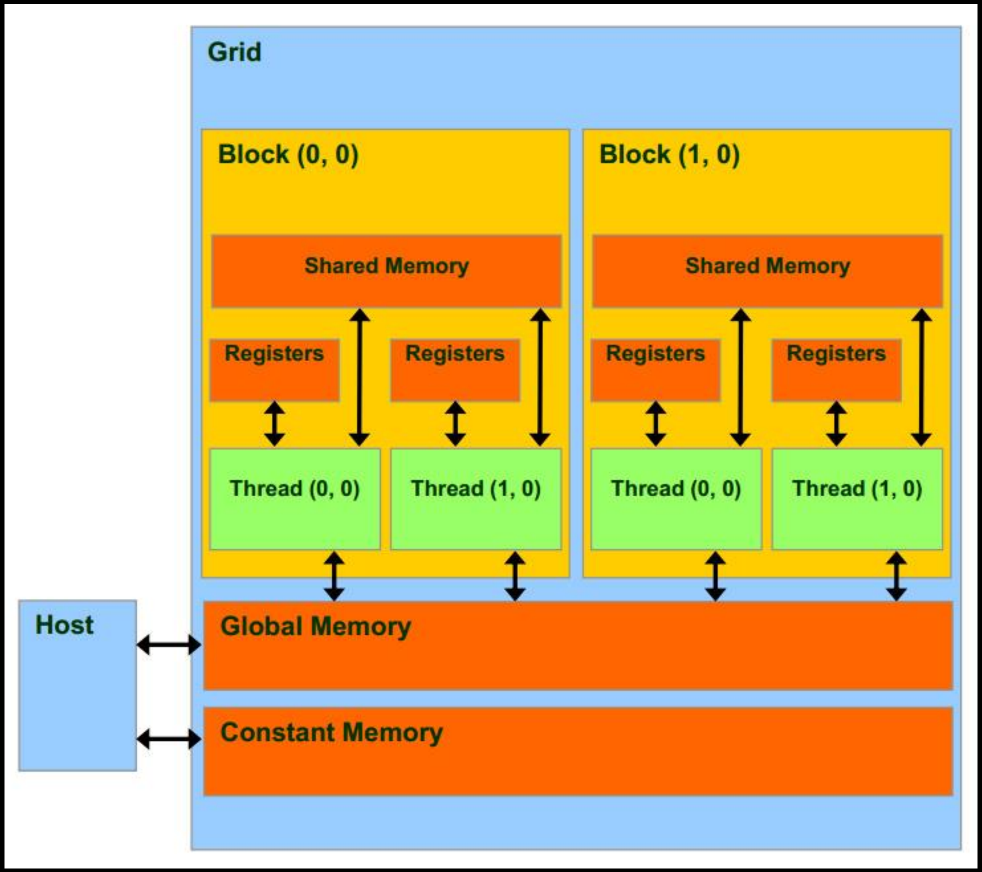
- Parallelism: thousands of cores
- Pipelining
- Hardware multithreading
- Not multiscale caching
    - Streaming caches
- Throughput, not latency

Flynn's Taxonomy

Single Instruction Single Data (SISD)	**Multiple Instruction Single Data (MISD)**
Single Instruction Multiple Data (SIMD)	Multiple Instruction Multiple Data (MIMD)

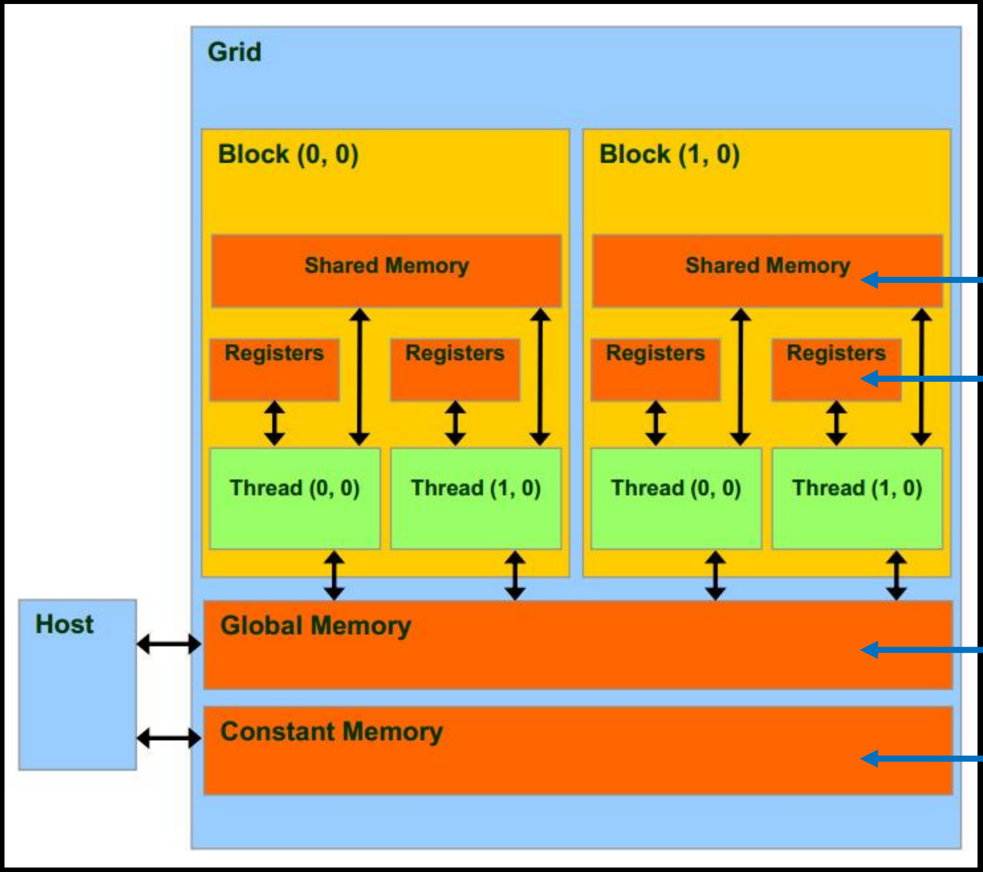
MIMD array of SIMD procs

Grids, Blocks, and Threads



Shuang Zhao, Cornell University, 2014

CUDA Memory



Faster, per-block

Fastest, per-thread

Slower, global

Read-only, cached

Heterogeneous Computing

Host: the CPU and its memory

Device: the GPU and its memory

Programming using CUDA

Compute Unified Device Architecture

```
do_something_on_host();
kernel<<<nBlk, nTid>>>(args);
cudaDeviceSynchronize();
do_something_else_on_host();
```

…

Highly parallel

Hardware Thread Organization

Threads in a block are partitioned into warps

- All threads in a warp execute in a **S**ingle **I**nstruction **M**ultiple **D**ata, or SIMD, fashion
- All paths of conditional branches will be taken
- Warp size varies, many graphics cards have 32

NO guaranteed execution ordering between warps

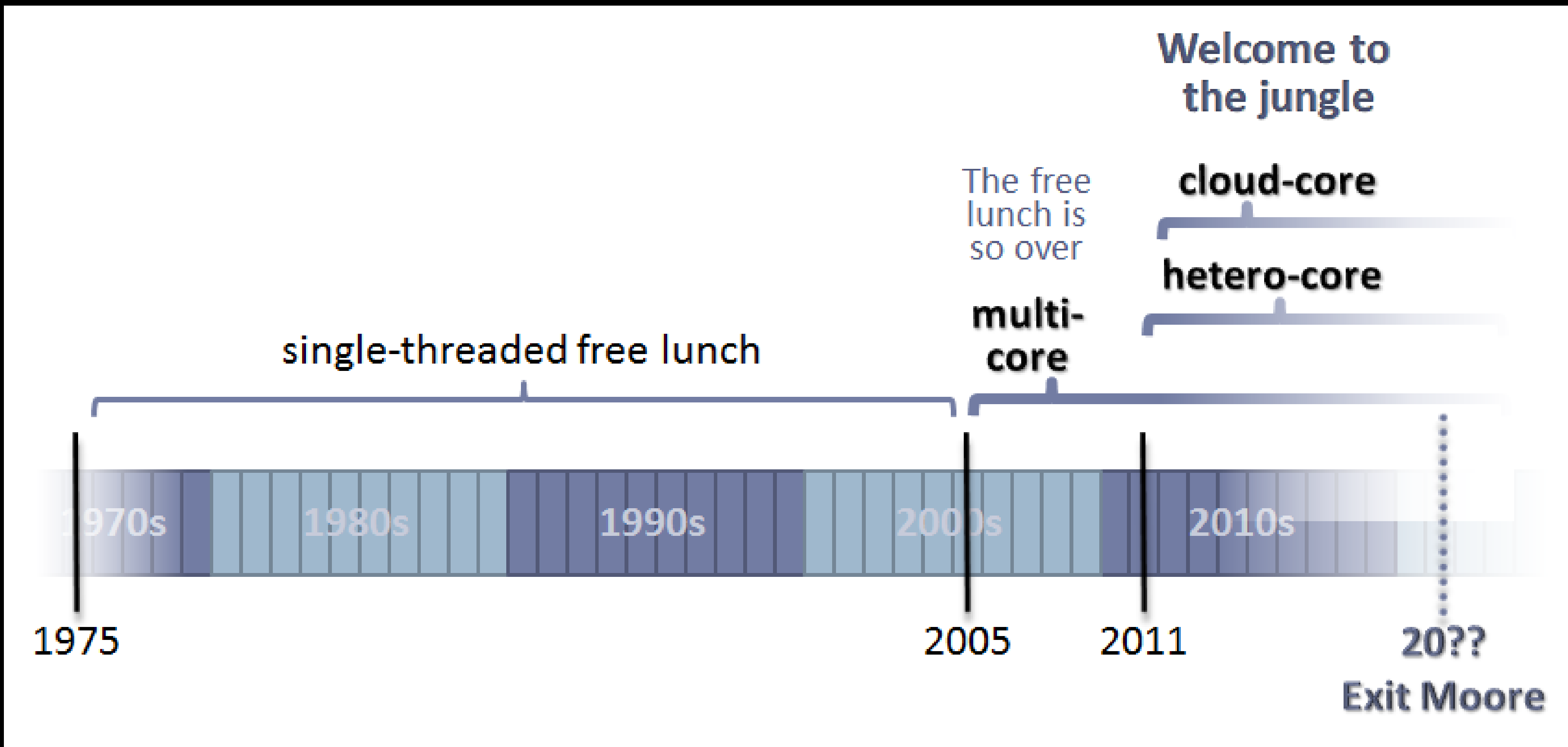
Branch Divergence

Threads in one warp execute very different branches

Significantly harms the performance!

Simple solution:

- Reordering the threads so that all threads in each block are more likely to take the same branch
- Not always possible



Welcome to the jungle
cloud-core
The free lunch is so over
hetero-core
multi-core
single-threaded free lunch
1970s
1980s
1990s
2000s
2010s
1975
2005
2011
20??
Exit Moore