Introduction to Computer Systems

15-213/18-243, fall 2009

24th Lecture, Dec 1

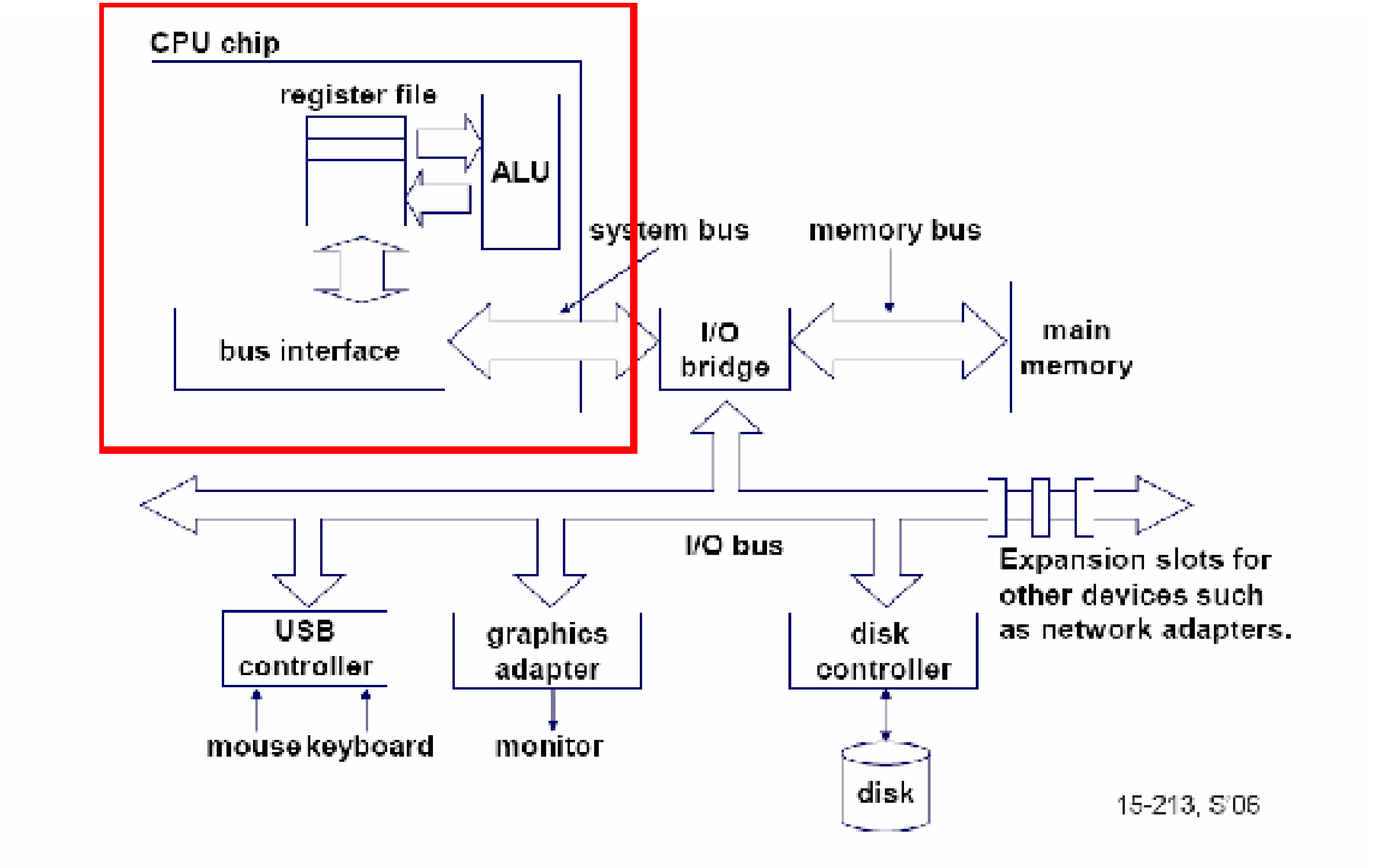
Instructors:

Roger B. Dannenberg and Greg Ganger

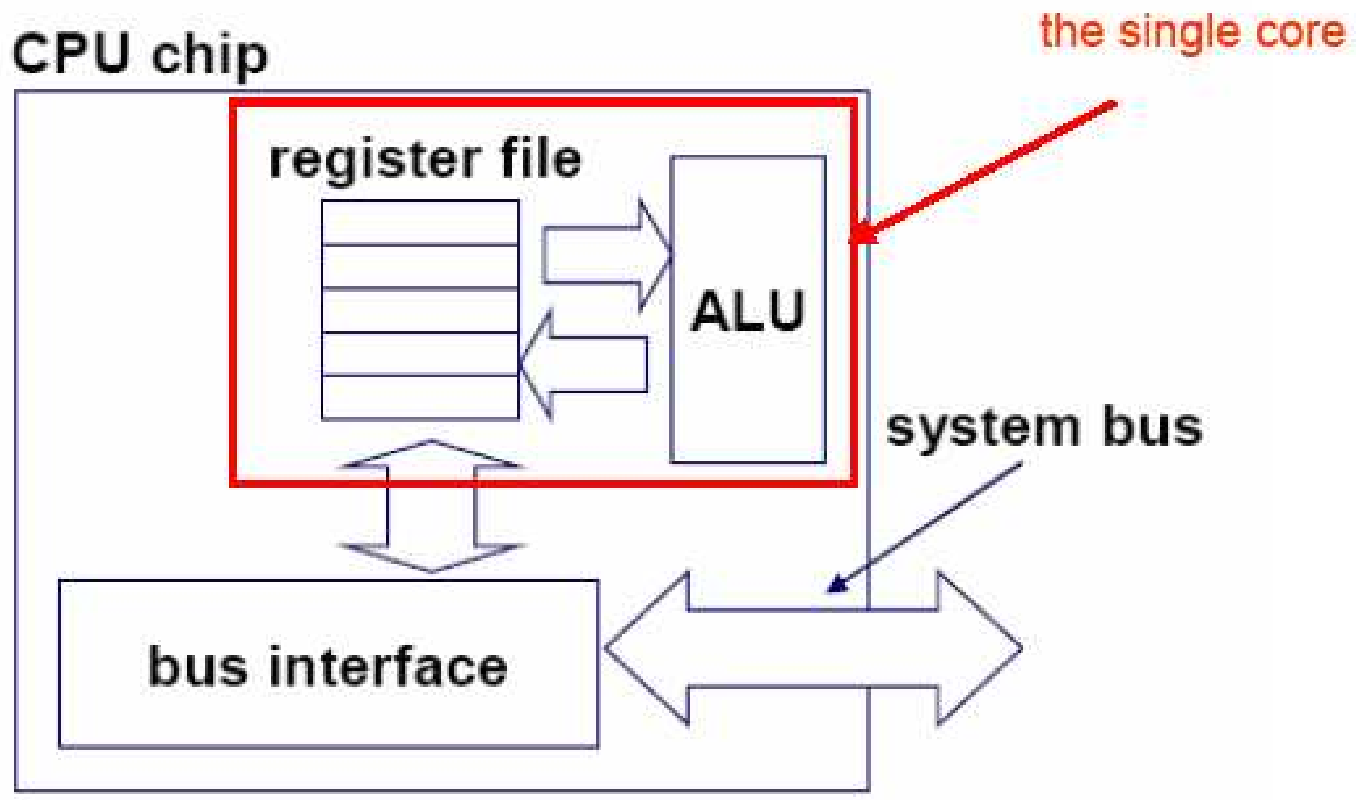
Today

- **Multi-core**
- **Thread Level Parallelism (TLP)**
- **Simultaneous Multi -Threading (SMT)**
- **Parallel Programming Paradigms**
  - OpenMP
  - Functional Programming
  - Thread pools, message passing, pipeline processing
  - SIMD, Vectors, and CUDA

Single-core computer

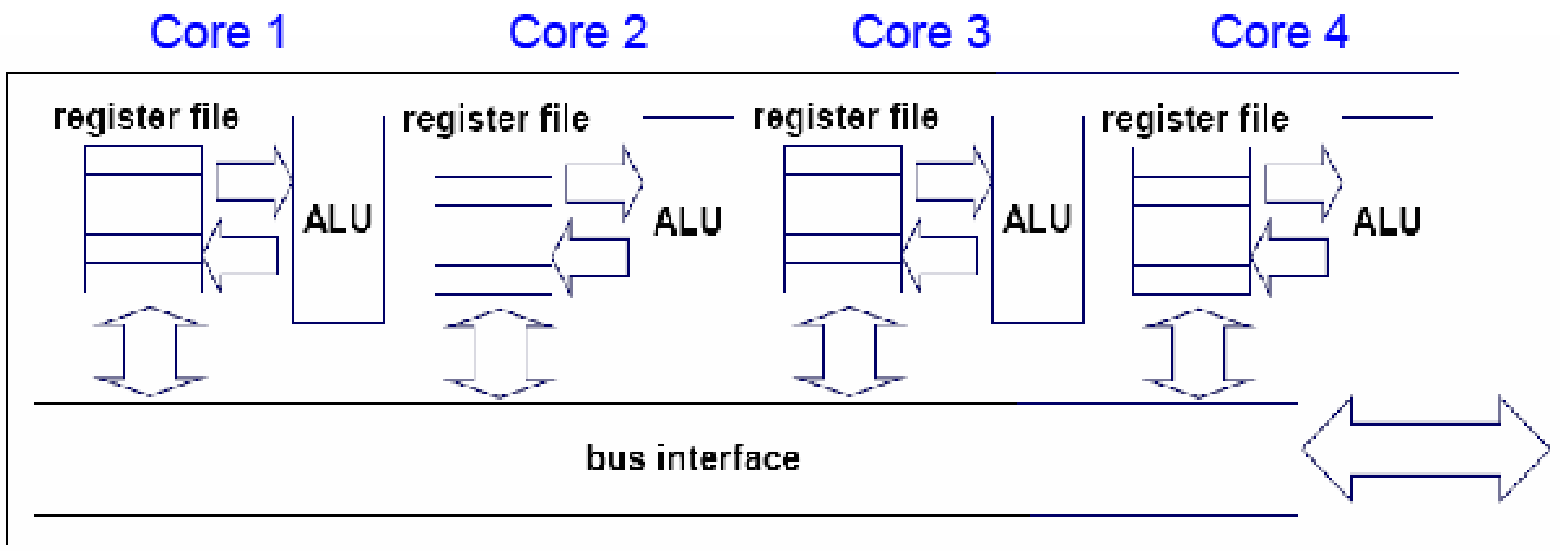


Single-core CPU chip



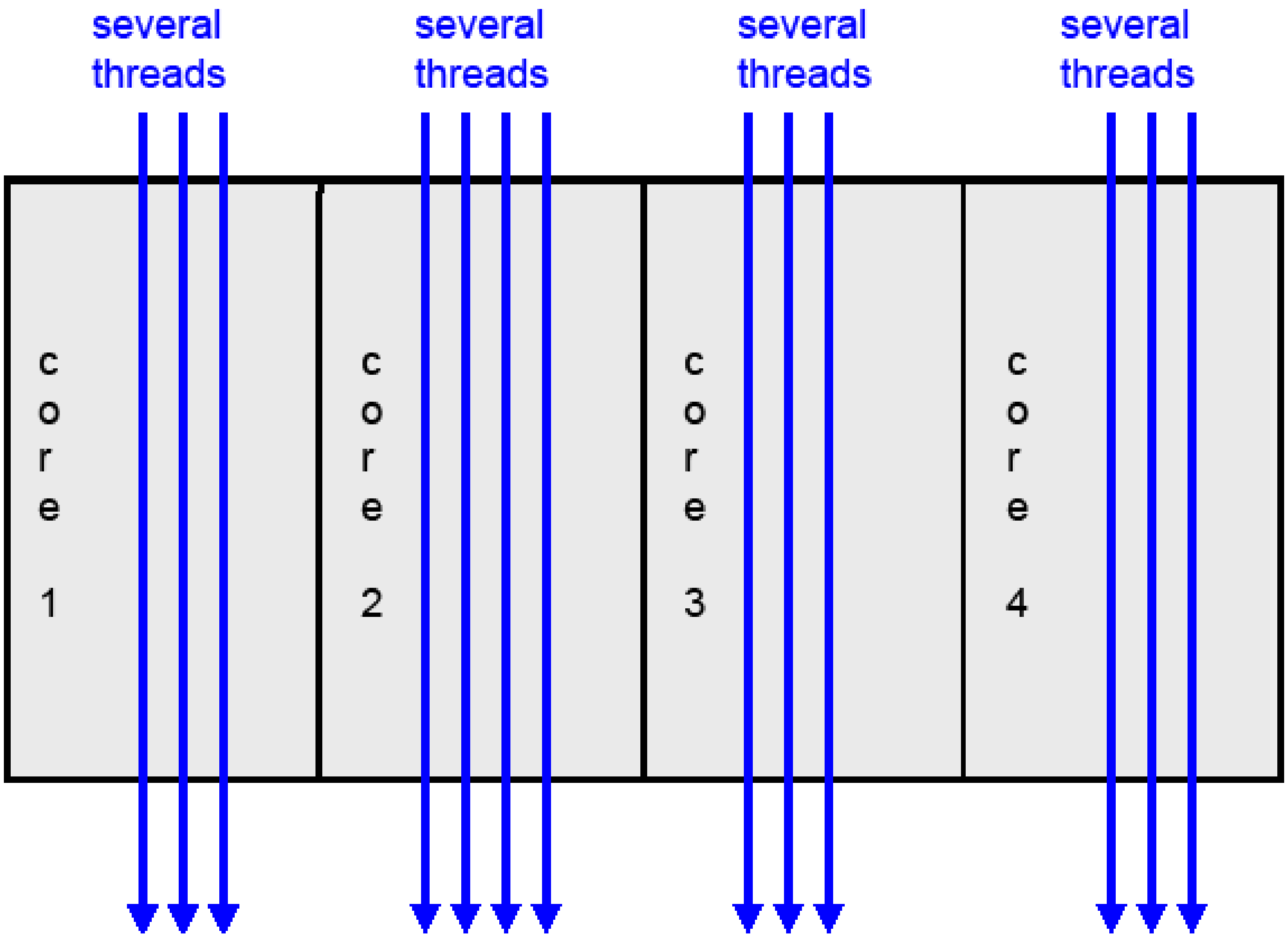
Multi-core architectures

- This lecture is about a new trend in computer architecture:
  Replicate multiple processor cores on a single die.



Multi-core CPU chip

Within each core, threads are time-sliced (just like on a uniprocessor)



Interaction With the Operating System

- OS perceives each core as a separate processor
- OS scheduler maps threads/processes to different cores
- Most major OS support multi-core today: Windows, Linux, Mac OS X, …

Why multi-core ?

- Difficult to make single-core clock frequencies even higher
- Deeply pipelined circuits:
  - heat problems
  - speed of light problems
  - difficult design and verification
  - large design teams necessary
  - server farms need expensive air-conditioning
- Many new applications are multithreaded
- General trend in computer architecture (shift towards more parallelism)

Instruction-level parallelism

- Parallelism at the machine-instruction level
- The processor can re-order, pipeline instructions, split them into microinstructions, do aggressive branch prediction, etc.
- Instruction-level parallelism enabled rapid increases in processor speeds over the last 15 years

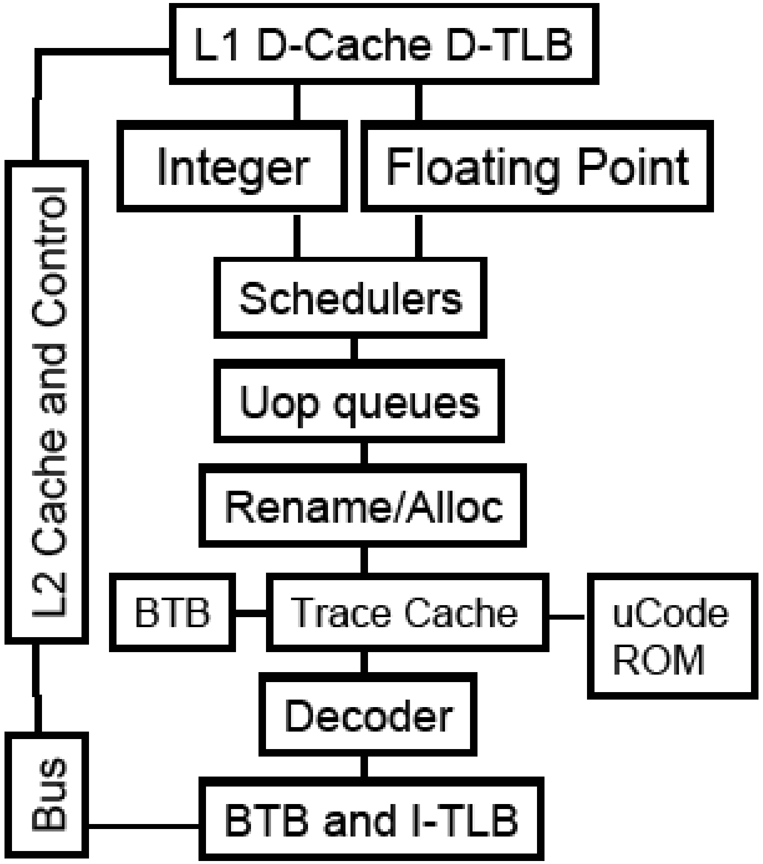
Thread-level parallelism (TLP)

- This is parallelism on a more coarser scale
- Server can serve each client in a separate thread (Web server, database server)
- A computer game can do AI, graphics, and physics in three separate threads
- Single-core superscalar processors cannot fully exploit TLP
- Multi-core architectures are the next step in processor evolution: explicitly exploiting TLP

A technique complementary to multi-core: Simultaneous multithreading

- Problem addressed: The processor pipeline can get stalled:
  - Waiting for the result of a long floating point (or integer) operation
  - Waiting for data to arrive from memory

  Other execution units wait unused

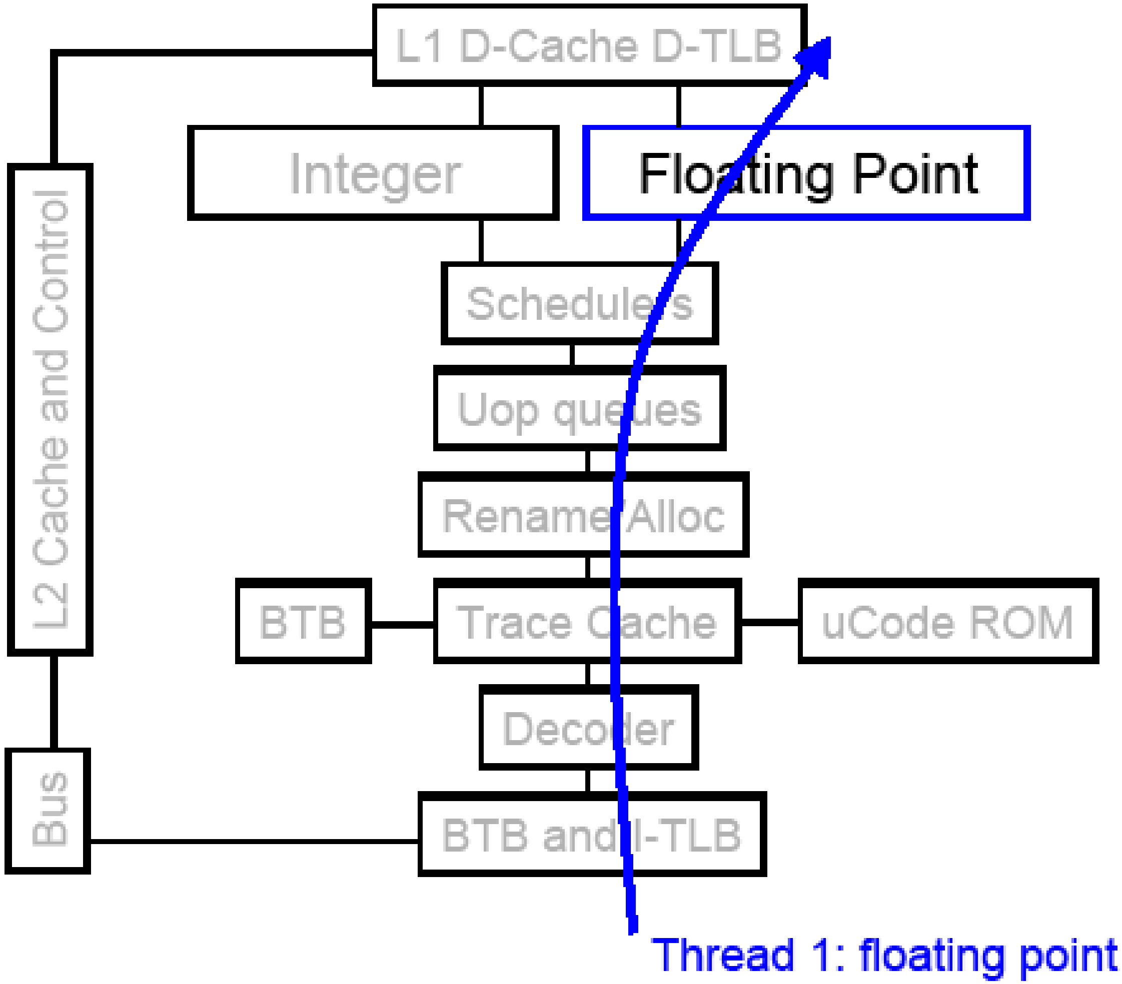


Source: Intel

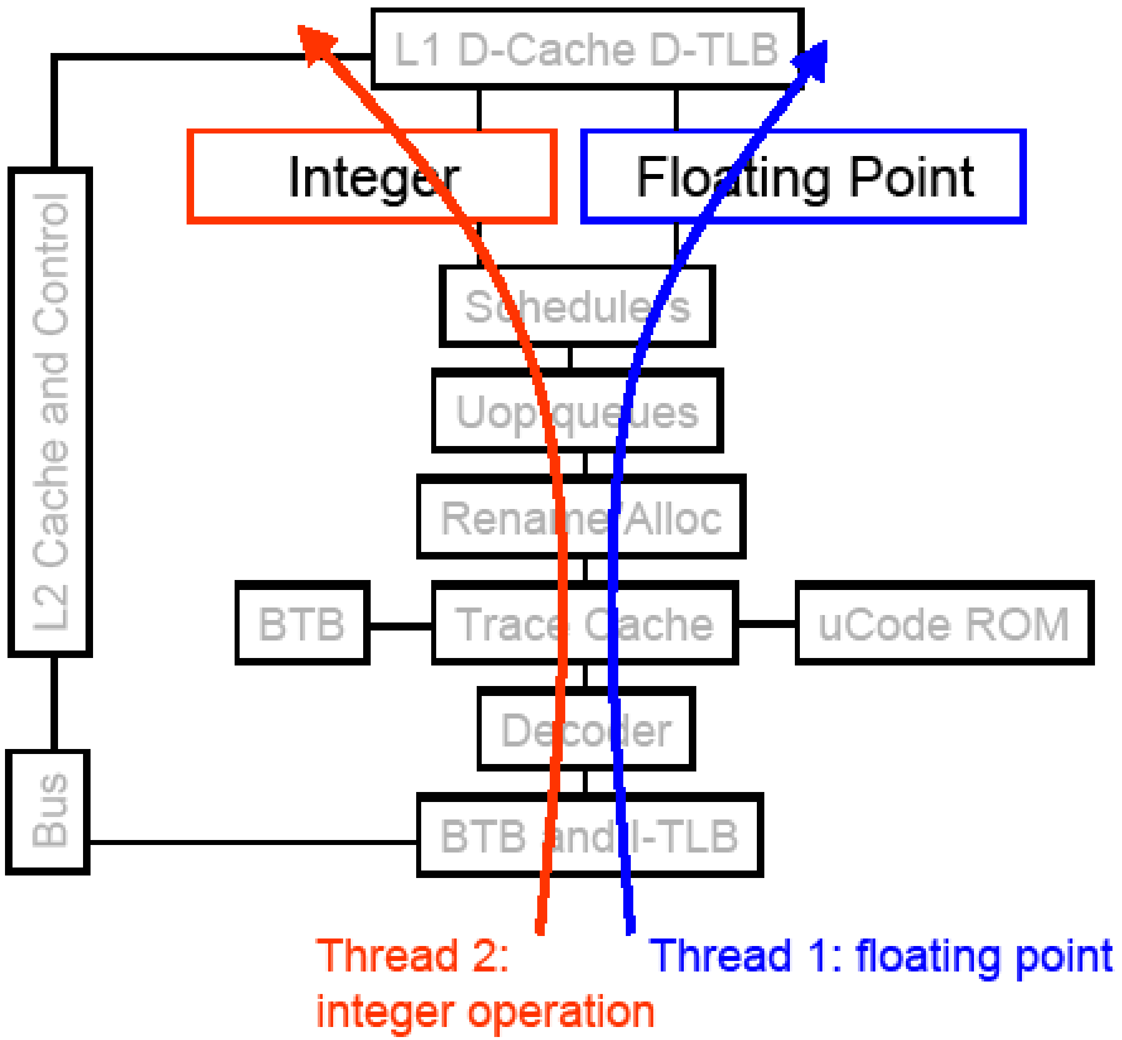
Simultaneous multithreading (SMT)

- Permits multiple independent threads to execute SIMULTANEOUSLY on the SAME core
- Weaving together multiple "threads" on the same core
- Example: if one thread is waiting for a floating point operation to complete, another thread can use the integer units

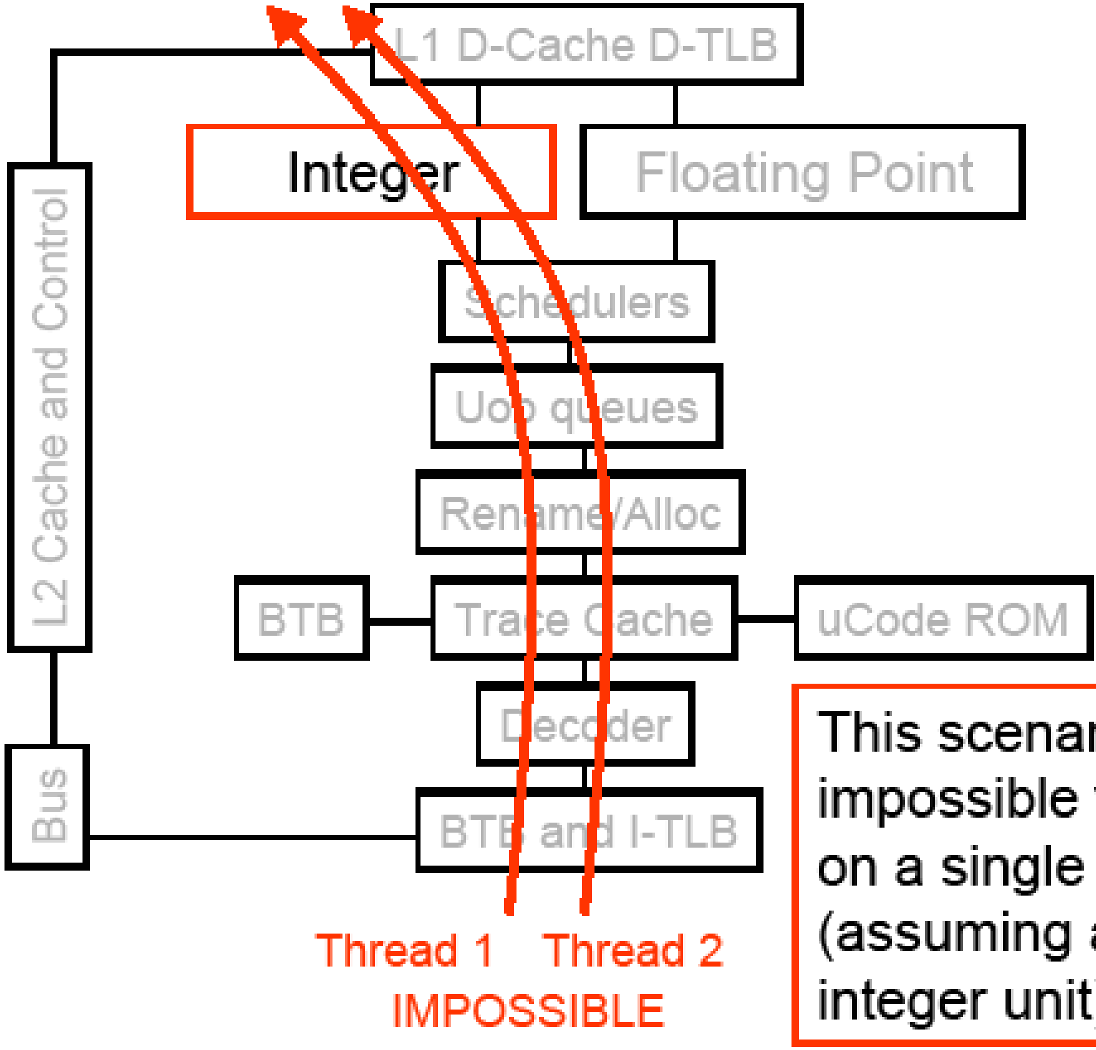
Without SMT, only a single thread can run at any given time



SMT processor: both threads can run concurrently

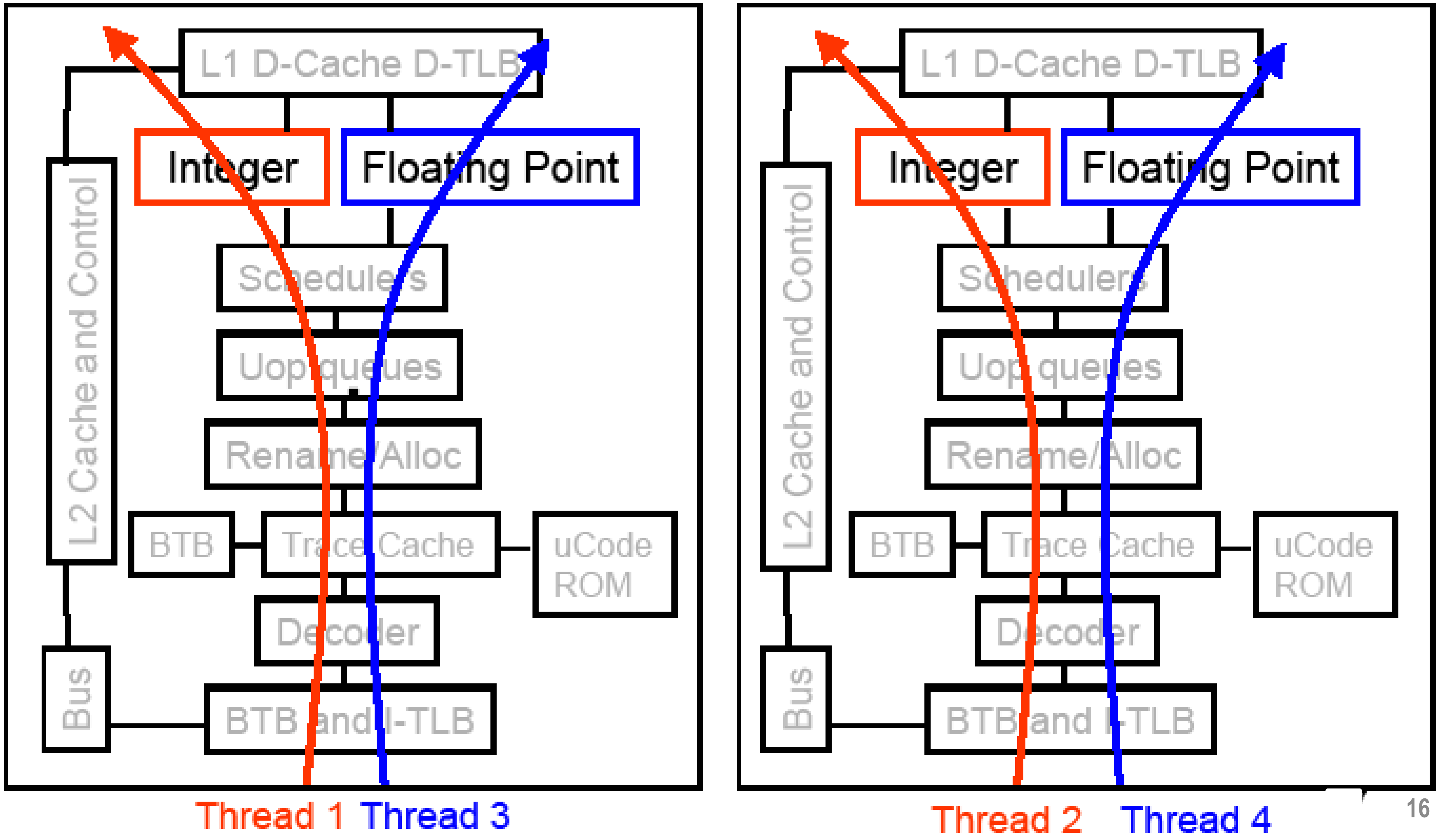


But: Can't simultaneously use the same functional unit



This scenario is impossible with SMT on a single core (assuming a single integer unit)

SMT Dual-core: all four threads can run concurrently



Comparison: multi-core vs SMT

- Multi-core:
  - Since there are several cores, each is smaller and not as powerful (but also easier to design and manufacture)
  - However, great with thread-level parallelism
- SMT
  - Can have one large and fast superscalar core
  - Great performance on a single thread
  - Mostly still only exploits instruction-level parallelism

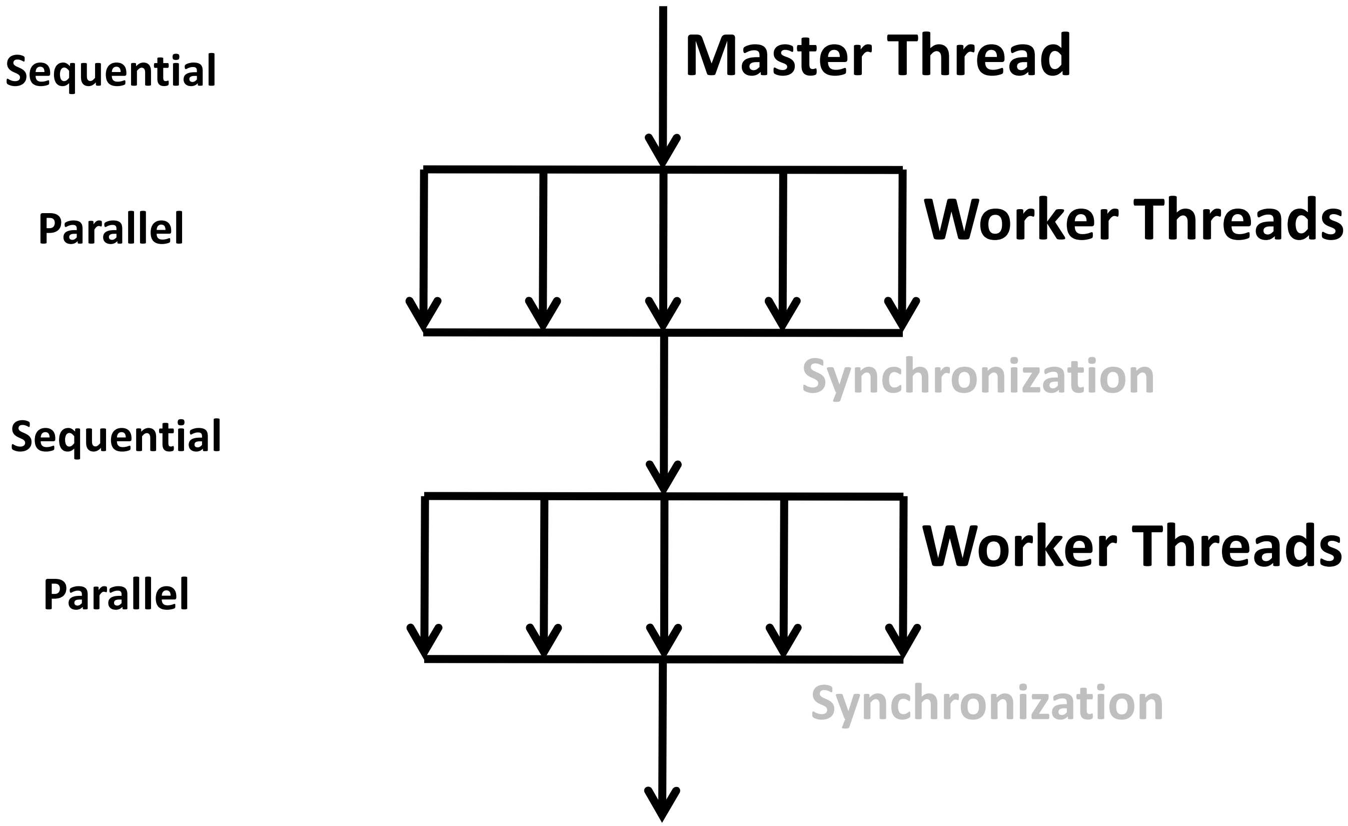
Today

- Multi-core
- Thread Level Parallelism (TLP)
- Simultaneous Multi -Threading (SMT)
- **Parallel Programming Paradigms**
  - OpenMP
  - Functional Programming
  - Thread pools, message passing, pipeline processing
  - SIMD, Vectors, and CUDA

OpenMP

- Goal: ease development of multi-threaded applications
- Implicit (sort-of) creation, synchronization, deletion of threads
- Platform independent
- Uses compiler pragmas, directives, function calls, environment variables

The OpenMP Execution Model



- **Fork and Join Model**

OpenMP Example

```
#pragma omp parallel for
for (i=0; i < numPixels; i++)
{
   pGrayScaleBitmap[i] = (unsigned BYTE)
            (pRGBBitmap[i].red * 0.299 +
             pRGBBitmap[i].green * 0.587 +
             pRGBBitmap[i].blue * 0.114);
}
```

- OpenMP divides work among threads
- Each thread performs a subset of the thread iterations
- Restrictions apply: no break or goto, update to i, etc.

OpenMP Data Dependencies

```
#pragma omp parallel for
for (i=2; i < 10; i++)      {
    factorial[i] = i * factorial[i-1];
}
```

- Each iteration depends upon data from the previous one
- Example of a *race condition*
- OpenMP does not detect data dependencies and race conditions

OpenMP Reduction

```
sum = 0;
#pragma omp parallel for reduction(+:sum)
for (i=0; i < 100; i++) {
    sum += array[i];
}
```

- OpenMP creates a sum variable for each thread
- Initialized sum variables to zero
- No race conditions or synchronization to update sums
- When threads exit, OpenMP adds the local sums to form the correct value of the global variable sum

OpenMP Critical Sections

```
#pragma critical (name)
{ Critical Section }
```

- All threads execute the critical section, but only one at a time
- Used to access/update shared variables

OpenMP Limitations

- **Assumes shared memory**
  - Does not scale up to large numbers of processors
- **C, C++, and Fortran**
  - But not more modern languages

Functional Programming

- **Fundamental idea: compute by composing functions**
- **"Function" is mapping from input to output**
  - Always get the same output when given the same input
  - No state, so no race conditions

Imperative Programming

```
for (i = 0; i < n; i++) {
    a[i] = 5 * a[i];
}
```

Functional Programming

```
b = array(1, n)
    [(i, a!i) | i <- [1..n]]
```

i is modified, a is modified

each variable gets one value

- **f(g(x), h(y))**
  - Note that g(x) and h(y) can run in parallel

Strict and Non-strict

- **producer = f() : producer()**
  - *Here,* a:b *means construct a list with* a *at the head and* b *at the tail*
- **consumer(x : rest) = g(x); consumer(rest)**
- **consumer(producer())**

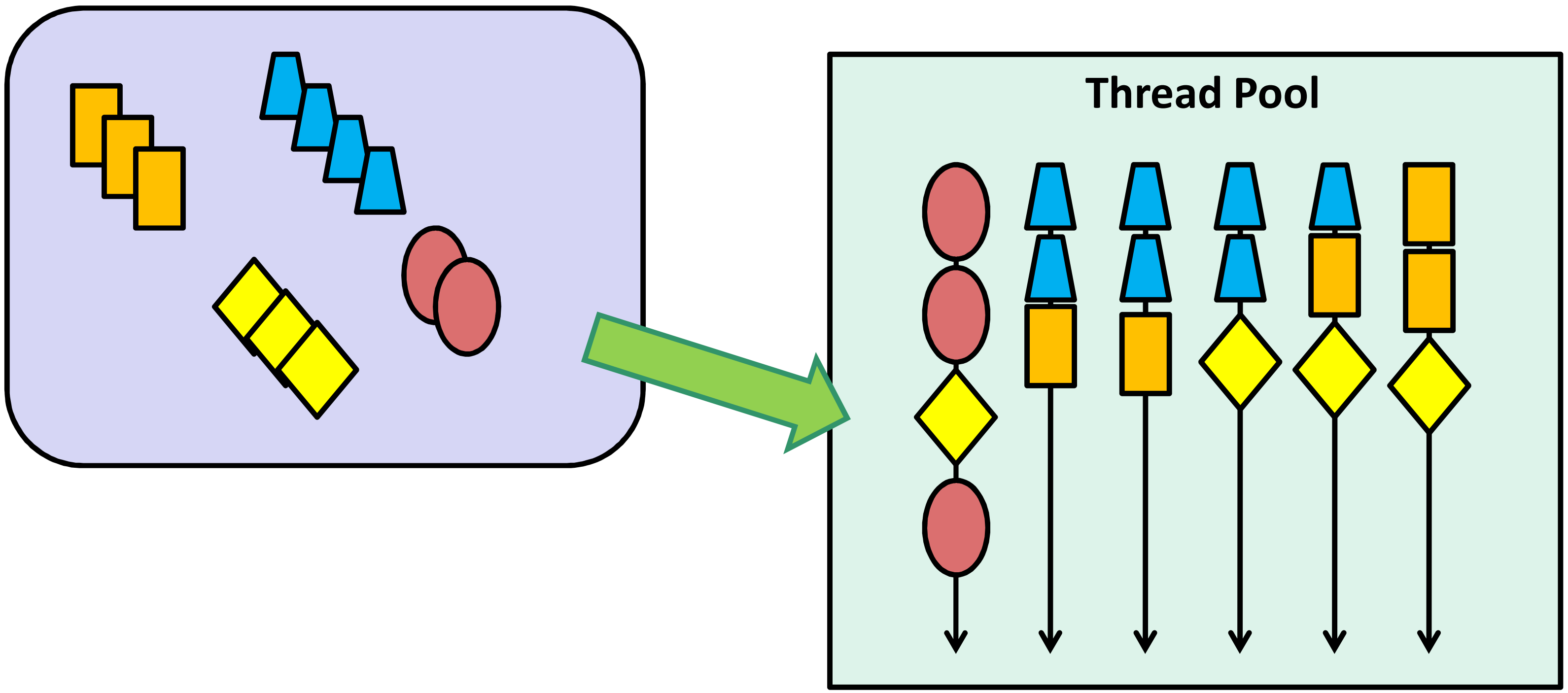
- **Imperative programming: run producer first (does it complete?) and pass result to consumer**
  - Called "strict" data structure
- **Functional programming with non-strict data:**
  - Consumer can start on head of list while producer is computing the rest of the list
  - Possible because structure is immutable
  - Implicit synchronization through data dependencies

Functional Programming Summary

- **Positive Features**
  - Stateless programming eliminates race conditions
  - Values can be used as soon as they are computed, implicit synchronization based on availability of values
  - Non-strict list structures offer an clean model related to producer-consumer, stream processing, and message passing
- **Negative Features**
  - Possibly too much parallelism: every function call potentially forks to compute each parameter
  - Non-strict lists can use infinite memory if eagerly evaluated
  - Not always a natural fit to “real” problems, e.g. how does a functional program produce output without side effects?
- **Overall, very promising, essential and useful concepts, but might not be the whole story**

Thread Pools, Message Passing, Pipelined Processing

- **Basic Concept:**
  - Express work as large set of independent tasks
  - Allocate enough threads to utilize all available cores
  - Assign a task to a thread whenever one becomes free



Small But Not Too Small

- **Tasks should be small**
  - Breaking up tasks into many small units allows more parallelism
  - Long-running task could lead to sequential execution when there are no other tasks to run
- **Tasks should not be too small**
  - Overhead to run a task includes:
    - Creation of the task description
    - Queuing task description for execution
    - Dequeue task by another thread
    - Possibly communicate results back to initial thread
  - Task size should be worth the overhead
    - If equal to overhead, no gain
    - Should be order of magnitude larger

How Many Threads in the Pool?

- **As many threads as tasks?**
  - Too many threads use extra memory
  - Contention for cache memory
  - Context switching
- **One thread per hardware core?**
  - Better, but not all threads are always ready to run
- **Enough threads to have a running thread per core**
  - Production thread pools can introduce new threads when tasks block and cores go idle
  - Reduce (or block) threads when there are more runable threads than cores

Futures

- **What if a task computes a result?**
- **Task is specified by a function that returns type V:**
  - V myTask()
- **Future<V> is a template type returned when you queue a task for execution by a thread pool**
- **V Future<V>::get()**
  - To use the return value, call its get() method
  - If value is computed (function returned), get() returns the value
  - If function has not returned, get() blocks

Examples

- OS X: "Grand Central Dispatch"
- Java: newFixedThreadPool(int nThreads)
- .NET: TPL = Task Parallel Library

- Almost all practical libraries and languages use thread pools at some level of implementation to efficiently assign tasks to cores and execute them

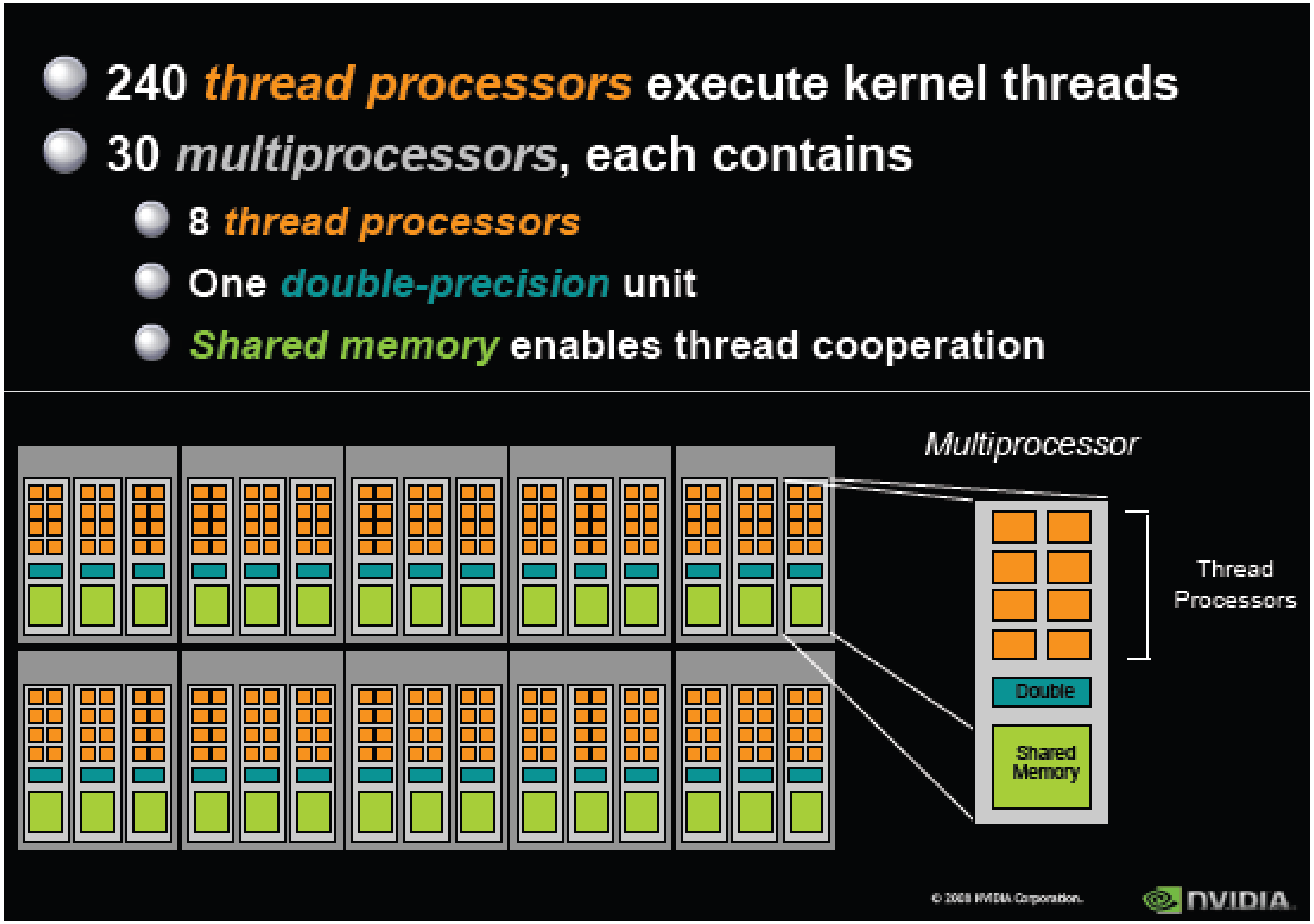
SIMD, Vectors, and CUDA

- SIMD: single instruction, multiple data
- We saw examples in the IA32 and x86-64 SSE instructions
- Instead of
  - for (i = 0; i < n; i++) a[i] = b[i] + c[i]
- Write
  - a = b + c
- Especially useful for numeric programming
- Problems crop up when there are conditionals, or data is not completely homogenious
- Example: pixel processing in computer graphics
  - A small program runs for each pixel
  - Problem: execute a small bit of code for each element of a vector

CUDA

- **nVidia's parallel programming model**
- **Maps to TESLA architecture and multi-core computers**
  - Considers 100's of cores, 1000's of threads
- **Model:**
  - An array of threads executes a single *kernel*
  - Each thread has an *id* to form thread-specific memory addresses
  - Threads are organized into groups called *blocks*
    - *Blocks* share memory and can synchronize but
    - No sharing/synchronizing between blocks
- **Scalability is central**
  - Basic unit is 8-core *multiprocessor* with shared memory
  - Each chip has many multiprocessors, e.g. 16 to 30
  - Run *N* kernel instances at a time until entire thread array executed

nVidia TESLA Architecture



Execution Model

- **Host initializes global memory**
- **Host “launches” a kernel (a block of code)**
- **Each thread runs the kernel**
  - If there are 240 thread processors, run threads 0-239, then run threads 240-479, then threads 480-719, etc.
  - Thread launching is assisted by hardware – very fast
  - Kernel can access global memory
- **Host retrieves results from global memory, and/or moves new data to global memory**
- **Host can then launch another kernel**

CUDA Memory

- **Per-Thread Memory using thread processor registers (fast) and off-chip memory (large, not cached)**
- **Per-Block Memory using multiprocessor shared memory (small but fast)**
- **Per-Device Memory**
  - Large
  - Not cached
  - Persists across kernel launches

- **Kernel is basically a C function with some extra annotations to specify type of memory for each variable**

CUDA Summary

- **Interesting combination of hardware and software**
  - Hardware has limited memory to maximize threads per chip
  - Fast kernel launching in hardware eliminates software thread pool
  - Software limited to map very directly to hardware
- **10x to 100x speedup of many scientific applications**
- **Lower power, smaller space**
- **Requires careful programming taking architecture into account**

Summary

- **Instruction-level parallelism**
  - Hitting limits due to transistor density and heat dissipation
- **Thread-level parallelism**
  - Alternative path to faster computation – if we can write software
- **Simultaneous Multithreading (SMT)**
- **OpenMP: fork/join style parallelism**
- **Functional Programming**
  - Key ideas: variables do not change value, no races, implicit synchronization via data dependencies
- **Thread Pools**
  - Key ideas: avoid thread creation overhead; break program into small tasks, thread pool maps tasks to threads; futures
- **SIMD, CUDA**
  - Key ideas: same code operates on every array element, CUDA is "Single Kernel, Multiple Instances (SKMI)"

References

- Ruud van der Pas, Sun Microsystems. http://openmp.org/mp-documents/ntu-vanderpas.pdf
- http://software.intel.com/en-us/articles/getting-started-with-openmp/
- Hinsen, K. "The Promises of Functional Programming," *Computing in Science & Engineering*, 11(4), July-Aug. 2009
- Rishiyur S. Nikhil and Arvind. "Implicit Parallel Programming: Declarative Programming Languages" http://www.embedded.com/design/multicore/201804960
- http://www.ddj.com/go-parallel/article/showArticle.jhtml?articleID=216500409
- http://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf