Statistics and Numerical Methods, Tsinghua University

Introduction to scientific computing

Xuening Bai (白雪宁)

Institute for Advanced Study (IASTU) & Department of Astronomy (DoA)

清華大學

Tsinghua University

Sep. 10, 2024

Hardware and software

Before we discuss algorithms of scientific computing, we must briefly discuss elements of hardware (CPU, memory, etc.) and software (e.g., programming language), which are central to the execution of an algorithm.

We cannot cover these aspects in full extent (you need to take specialized courses). Here, we focus on the basic elements from the perspective of scientific computing.

Course survey

How many of you are familiar with the following programming language:

C/C++: 3.5

Fortran: 0

CUDA/HIP: 1

Python: all

Matlab: 8

Julia: 1

Experience with computing environment:

Linux: all but 4

Parallel computing: 5

MPI: 3

OpenMP: 1

Programming languages

Primitive language:

Assembler, machine code

- Require explicit instructions about how to do everything
- Extremely powerful if in the right hands
- Very tedious to code and to use

Programming languages

Compiled (normal) language:

C/C++, Fortran, Pascal, BASIC, …

- Require specialized program (compiler) to translate human-readable code to machine-readable instructions (executable), a process that can be slow.
- This allows for complex and advanced code optimizations, leading to fast execution.
- Much easier than low-level language, but less convenient than interpreted language, usually non-interactive.

Examples

```
swap(size_t v[], size_t k)
{
    size_t temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

High-level language (C)

Compile:

```
swap:
        slli x6, x11, 3
        add  x6, x10, x6
        ld   x5, 0(x6)
        ld   x7, 8(x6)
        sd   x7, 0(x6)
        sd   x5, 8(x6)
        jalr x0, 0(x1)
```

Assembly language

Assemble:

machine code

```
00000000001101011001001100010011
00000000011001010000011001100111
00000000000000011001100101000011
00000000100000110011001110000011
00000000111001100110000000100011
00000000101001100110100001000111
00000000000000001000000001100111
```

Programming languages

Interpreted (scripting) language:

Python, IDL, Matlab, Mathematica, PHP, R, Ruby, Unix shell…

- Use a runtime interpreter that directly executes the program without prior compilation.
- Very easy to learn, usually interactive, short edit-run cycle, extensive libraries.
- Slow: there can be substantial interpretive overhead, 10x slower is not uncommon.

Julia is a borderline case, using a just-in-time compiler that combines the merits of both.

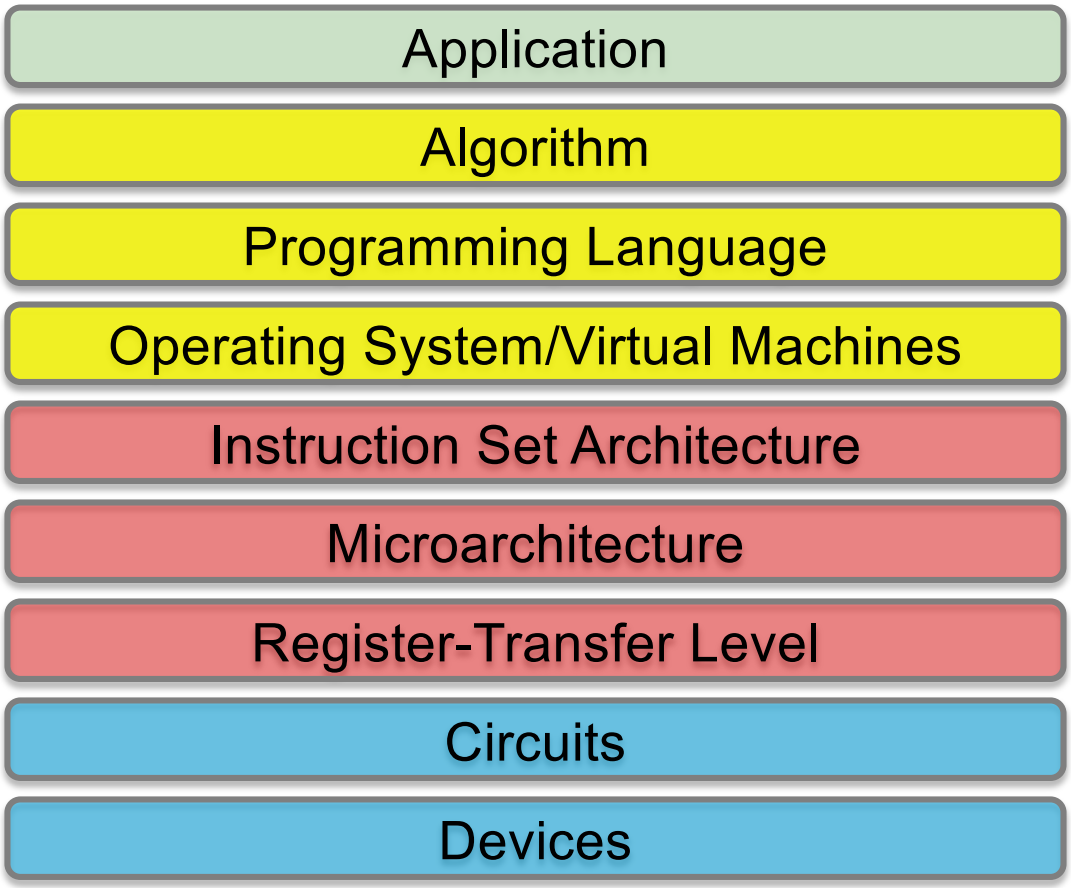
Can combine scripting+compiled languages, e.g., use C/C++ for computationally-intensive part, with python wrapper and user interface.

Outline

- Introduction to hardware
  - The processor
  - Memory
  - Parallelism
- Computer arithmetic
  - Integer arithmetic
  - Floating point arithmetic
  - Rounding error

Excellent reference: *Computer organization and design: the hardware/software interface*, by D.A. Patterson & J. L. Hennessy, 2018, 5th edition

Computer system: layers of abstraction



Instruction set

Processor (CPU): the “brains of the computer” that performs operations on data according to *instructions* controlled by the programmer.

Each processor has a unique instruction set, which is a portion that makes up an architecture.

Complex instruction set computer (CISC; e.g., x86 for most desktops/laptops)

Reduced instruction set computer (RISC; e.g., ARM for most phones/tablets)

Generally, a programmer writes algorithm in high-level language, to be translated to low-level assembly language and machine-readable instructions (compiling).

Basic example

All data processed by CPU enters and leaves via registers:

c = a + b =>

```
Load r1, a
Load r2, b
Add  r3, r1, r2
Store r3, c
```

At machine level, a classic RISC instruction (each line above) involves several stages:

- Fetch instruction from memory/instruction cache
- Decode instruction
- Execute
- Memory access (if needed)
- Write back (if needed)

These are 5 stages, each taking one clock cycle => would take 5 cycles?

Pipelining

Often, the same sequence of instructions are repeated many times (loops).

Optimize (in the processor) by operating on different steps in sequence; like a pipeline.

Clock cycle:	1	2	3	4	5	6	7	8	9
instruction *i*	IF	ID	EX	MA	WB				
instruction *i*+1		IF	ID	EX	MA	WB			
instruction *i*+2			IF	ID	EX	MA	WB		
instruction *i*+3				IF	ID	EX	MA	WB	
instruction *i*+4					IF	ID	EX	MA	WB

IF=instruction fetch; ID=instruction decode; EX=execute; MA=memory access; WB=write back

Pipeline in the above example takes 9 clock cycles to complete 5 instructions.
Un-pipelined processor would take 5*5 = 25 cycles.

Pipelining is an example of instruction level parallelism (ILP)

Hazards to pipelining

There is no guarantee that the next instruction can be properly executed in a pipeline, due to various hazards:

- Structural hazards: instructions in a pipelines competing for resources.
- Data hazards: data for instruction i+1 depends on data produced by instruction i.
- Control hazards: pipeline contains conditional branch.

These hazards typically leads to a pipeline stall, also called a "bubble", resulting in deficiencies.

Modern compilers can do substantial optimizations under the hood, including pipelining, multi-issue (process >1 instructions per cycle), and ways to resolve hazards, including out-of-order execution, operand forwarding, branch prediction, etc.

Lessons to programmers

Isolate recursion formulae from other work, since it will likely interrupt pipelining of other instructions.

Avoid conditional branches in pipelined code, at least in the innermost loops (that are executed most frequently).

BAD

```
for i=1:10
  for j=1:10
    if (i>j) sign=1.0;
    else if (i<j) sign=-1.0;
    else sign=0.0;
    C(j) += sign*A(i,j);
  end
end
```

*Modern compilers are smart enough that you probably won't see any difference between the two codes, but it's good habit to avoid branching in the innermost loop.

Good

```
for i=1:10
  for j=i+1:10
    C(j) += A(i,j);
  end
end
for i=1:10
  for j=1:i-1
    C(j) -= A(i,j);
  end
end
```

Memory technologies

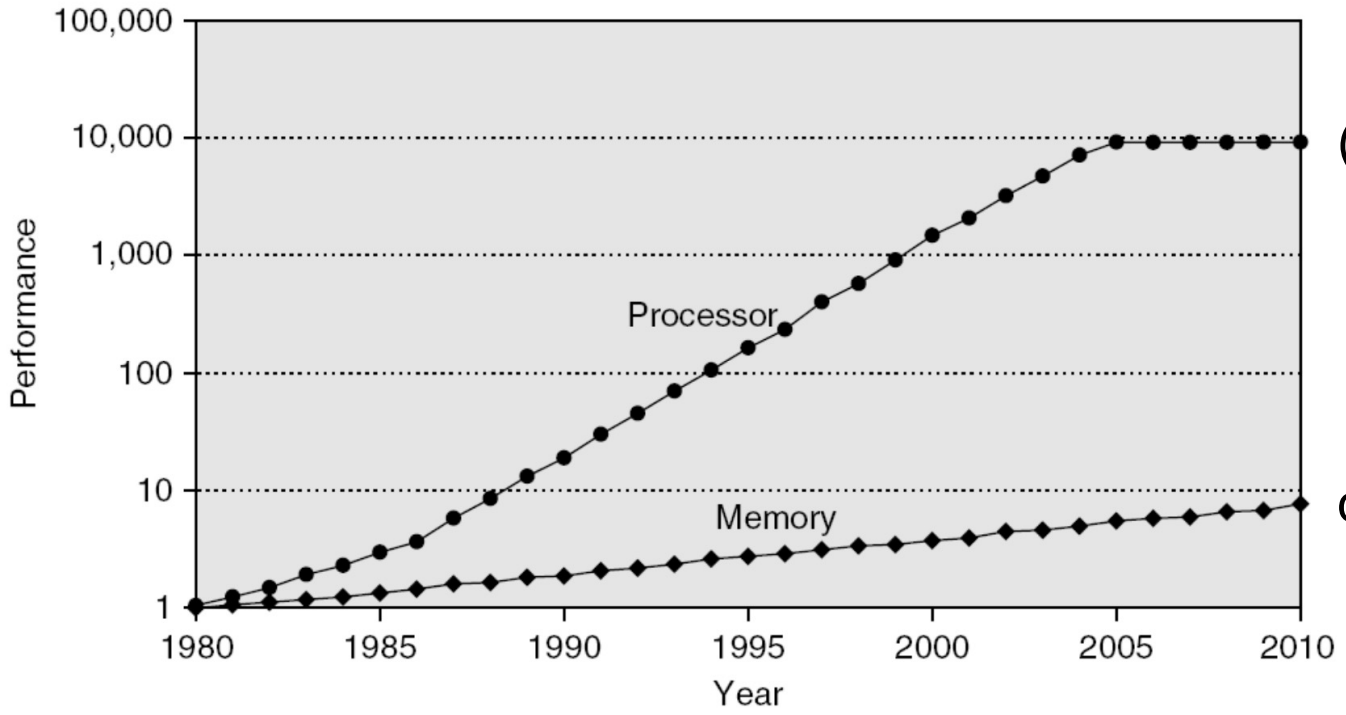
Programmers want unlimited amount of memory with low latency, but fast memory is more expensive per bit than slower memory.

Technology	Typical access time	$ per GB
SRAM (cache)	0.5-2.5 ns	$500
DRAM (main memory)	50-70 ns	$6
Flash	5-50 μs	$0.75-$1
Magnetic disk	5-20 ms	$0.05-$0.1

Static Random Access Memory (SRAM): data can be kept indefinitely as long as power is on, access time close to cycle time, very expensive.

Main memory uses dynamical RAMs (DRAM): data must be constantly refreshed. Data transferred on each I/O clock cycle, but with a latency of 10s of clock cycles.

Processor-memory performance gap



(per core)

data bandwidth

Up to 2000s, processors have been following the Moore's law.

Gap between CPU and memory increases by 50% per year.

Principle of locality

Temporal locality:

If an item is referenced, it will tend to be referenced again soon.

Example: instructions (as well as some data) in a loop.

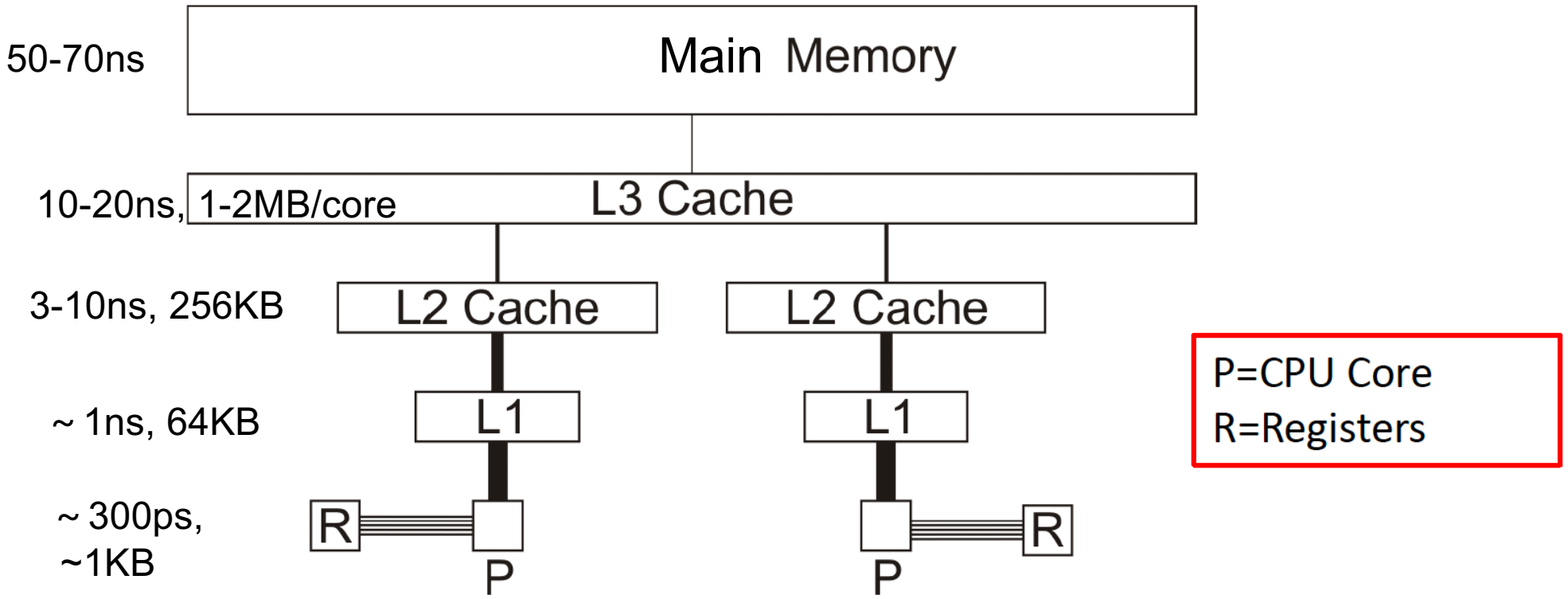
Spatial locality:

If an item is referenced, items whose addresses are close by will tend to be referenced soon.

Example: most programs access data sequentially, e.g., arrays in a loop.

Memory hierarchy

Taking advantage of the locality, modern computers organize the memory system into a hierarchy.



How cache works

Data is transferred between levels in blocks of fixed size, known as a cache line, whose size is a power of 2, usually 16-128 bytes.

Suppose processor needs item at address A, but it is not in the register.

It will first look for the address in the L1 cache to fetch data.

If not found, it will search for higher-order caches, until reaching the main memory.

Once found, the system will copy a cache line of data (A+1, etc.) to the cache.

Then, if processor needs A+1 in the next cycle, it is already in the cache.

If memory location needed by processor is in cache = cache hit

If memory location needed by processor is not in cache = cache miss

How cache works

Penalty of cache miss: the processor has to wait (stall) until it gets data.

(modern processors can mitigate this)

Fraction of requests which are cache hits is hit rate (H).

Effective access time:

$$t_{\text{eff}} = Ht_{\text{cache}} + (1 - H)t_{\text{main}}$$

~10ns ~70ns

For H~0.95, obtain t_{eff}~13ns: almost as good as cache!

Goal of computer architecture: design cache to maximize hit rate & reduce miss penalty.

Lessons to programmers

Write codes that maximizes cache hits!

In particular, always access data contiguously. Order loops so that the inner loop is over neighboring data elements. Avoid stride not equal to one.

```
for (i=0; i<=100; i++) {
   for (j=0; j<=100; j++) {
    a[j][i] = b[j][i]*c[j][i];
}}
```

BAD

```
for (i=0; i<=100; i++) {
  for (j=0; j<=100; j++) {
    a[i][j] = b[i][j]*c[i][j];
}}
```

Good

Note: the above is for C-style code. Ordering is the OPPOSITE in FORTRAN.

Computer parallelism

Besides instruction-level parallelism, modern computers proceed with a variety of mechanisms to enhance speeds.

Modern computer systems employ multiprocessors, which contain at least 2 processors => individual processors are now called cores.

High performance is achieved through task-level (or process-level) parallelism: individual processors/cores execute independent tasks.

Within each processor/core, techniques such as SIMD and multi-threading are developed to enable substantial speed up.

Amdahl's law

Define *speedup* that can be gained by some performance enhancement:

$$\text{Speedup} = \frac{\text{execution time without enhancement}}{\text{execution time with enhancement}}$$

Let a = fraction of code that can use enhancement

S = speedup of entire code

S_a = speedup of enhanced portion of the code

$$S = \frac{1}{(1-a) + a/S_a}$$

Suppose your code is 99% parallelized (only 1% of work is serial).

What is the speedup on 10 processors? S = 1/([1-0.99]+0.99/10) = 9.2

100 processors? S = 1/([1-0.99]+0.99/100) = 50

1000 processors? S = 1/([1-0.99]+0.99/1000) = 91

Code must be nearly 100% parallelized to exploit 1000s of processors!

SIMD parallelism

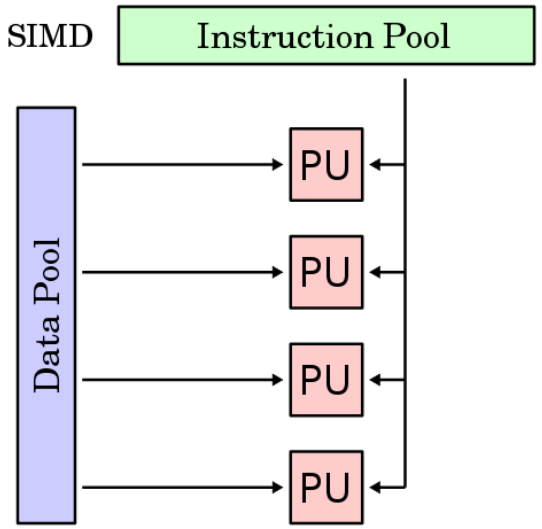
A single instruction synchronously operates on multiple different data streams (data-level parallelism).

Scalars: c = a + b

```
Load r1, a
Load r2, b
Add  r3, r1, r2
Store r3, c
```

Vectors: C = A + B

```
Load R1, A
Load R2, B
Add  R3, R1, R2
Store R3, C
```



Works best with arrays in for loops, with nearly identical structured data.

Modern processors (AVX-512) can hold a vector length of 8 double-precision numbers => speed up by a factor of 8!

Modern CPUs and supercomputers

In contemporary Intel/AMD CPUs:

- A CPU consists of multiple largely independent cores.
- The cores share the memory (i.e., shared memory processor, SMP).
- Each core can run an independent instruction on independent data, and is essentially a SIMD processor.

Supercomputers today generally have a mixed architecture:

- Shared memory within each node, which contains dozens of cores.
- Distributed memory between nodes, which communicate over a network (now reaches 100GB/s).
- Communication introduces additional overhead and latencies, which can be major barrier to good scaling.

Lessons to programmers

Write vectorized codes! Make simple, vectorizable innermost for loops.

- Avoid complex branching statements (if/else, switch/case).
- Avoid dependencies within vector length.
- Operate on data contiguous in memory (stride of one).
- Avoid loop lengths slightly larger than vector length (e.g., 9^3 data arrays are not optimal if vector length is 8).

```
#pragma omp simd
for (i = 0; i < N; i++)
{
    a[i] += b[i]*c[i];
}
```

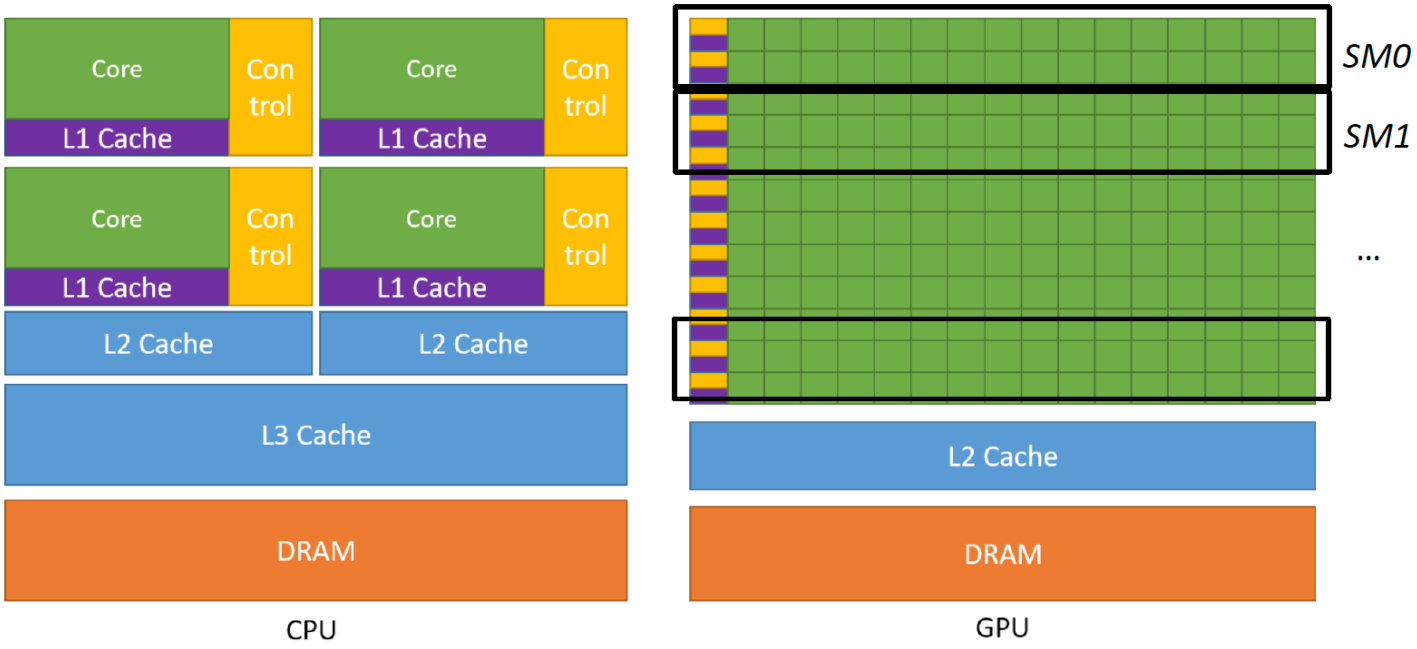
This is good.

```
#pragma omp simd
for (i = 1; i < N; i++)
{
    a[i] = a[i-1] + 1;
}
```

The code will execute but you may get wrong results.

GPU computing

Graphics processing unit (GPU): initially for digital image processing to accelerate computer graphics, now increasingly in data processing, AI and scientific computing.

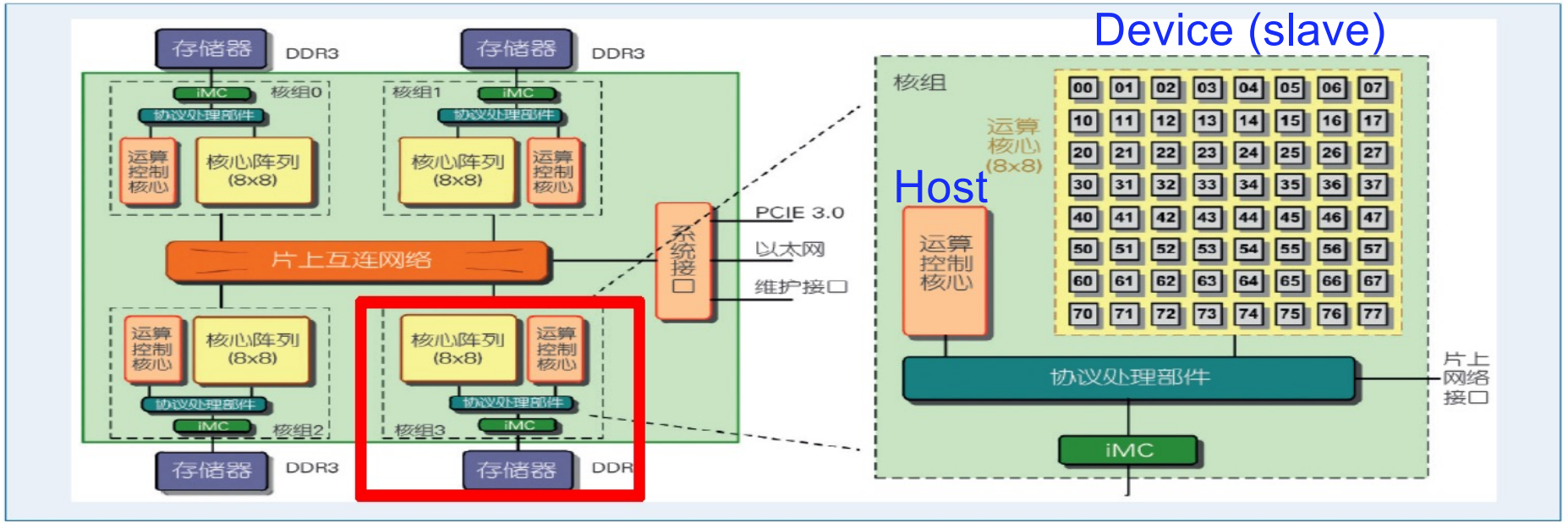


Up to 1000x more slow-processing cores to handle massively parallel but simple computational tasks.

Main challenge: small cache, and communication bandwidth with CPU.

Heterogeneous computing

Paradigm shift in programming models towards CPU + accelerator (GPU or others).



SW26010 (申威) on Sunway Taihulight.

Other domestic processors: 海光CPU/DCU (曙光), Matrix-2000 (天河), 鲲鹏/昇腾 (华为).

Outline

- Introduction to hardware
  - The processor
  - Memory
  - Parallelism
- Computer arithmetic
  - Integer arithmetic
  - Floating point arithmetic
  - Rounding error

Computer arithmetic

All numbers in a computer are represented in binary with specified number of bits.

Computer arithmetic handles the addition, subtraction, multiplication and division of these numbers (*at hardware level*).

They are mainly executed in the arithmetic logic unit (ALU) in the processor, supplemented by hardware for floating point operations.

Three data types:

- Integer
- Real (floating point)
- Complex (represented by 2 reals, usually handled by software)

Representation of numbers: integers

Integers are typically represented using 8-, 16-, 32- bit sequences. In C/C++:

Type	bits	Range (unsigned)	Range (signed)
short	8	[0,255]	[-128,127]
int	16	[0,65535]	[-32768,32767]
long	32	[0, 4294967295]	[-2147463648, 2147463647]

For unsigned integer $S=(S_N \ldots S_1)_{two}$, we have: $I = \sum_{i=1}^{N} 2^{i-1} S_i$

Example: $(00100100)_{two}=2^2+2^5=36$

For signed integer, the leading digit is negated: $I = \sum_{i=1}^{N-1} 2^{i-1} S_i - 2^{N-1} S_N$

Example: $(10100100)_{two}=2^2+2^5-2^7=36-128=-92$

Shortcut: invert all bits and add one

$(10100100)_{two}=>(01011011)_{two}=>(01011100)_{two}=2^6+2^4+2^3+2^2=92$

Arithmetic for integers

For integers, there are specific hardware implementation to compute addition/subtraction, multiplication and division. The arithmetic is exact.

Typically, it takes one clock cycle to compute addition and subtraction, and a few clock cycles (~log(# of bits) for multiplication and division (can be faster through pipelining).

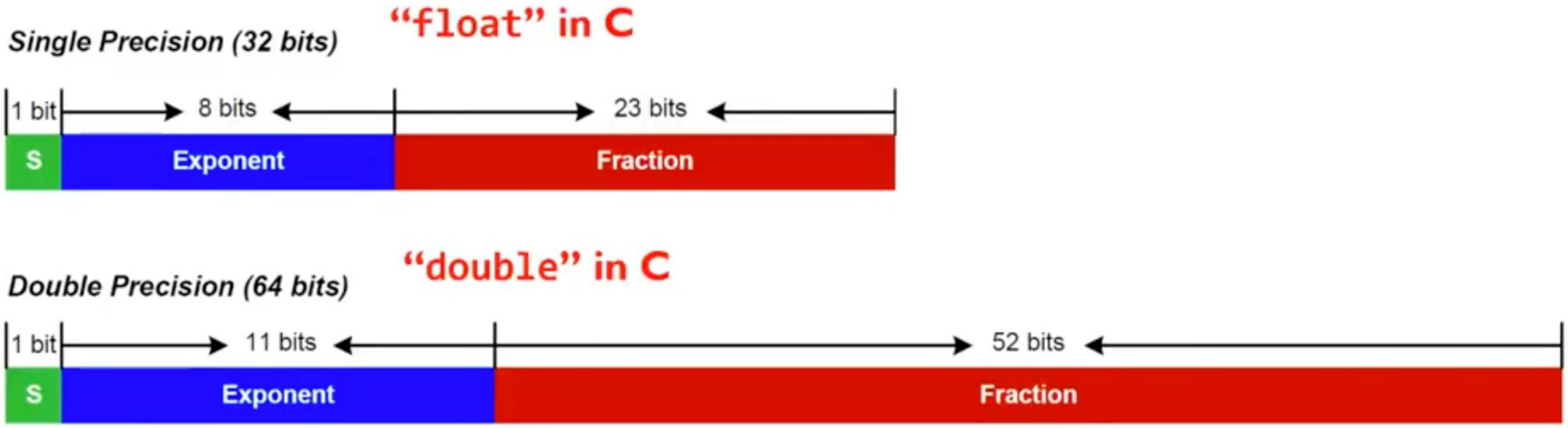
multiplicand		1000
multiplier	x	1001
		1000
		0000
		0000
		1000
product		1001000

Avoid overflows!

Typical behavior of overflow is 32767+1=-32768, but this is not guaranteed!

Representation of numbers: reals

Real numbers are typically stored in single (32 bit) or double (64 bit) precision, following the IEEE 754 standard.



Specifically, they are represented by $R = (-1)^S \times (1 + F) \times 2^{E-B}$

For single or double precision, *E* is in [1,254] or [1, 2046], the *bias* B is 127 or 1023.

The fraction F=(f_1…f_N) represents: $F = \sum_{i=1}^{N} 2^{-i} f_i$

Representation of numbers: reals

Specifically, they are represented by $R = (-1)^S \times (1+F) \times 2^{E-B}$

For single or double precision, *E* is in [1,254] or [1, 2046], the *bias* B is 127 or 1023.

The fraction *F*=($f_1 \ldots f_N$) represents: $F = \sum_{i=1}^{N} 2^{-i} f_i$

For single precision, the maximum number is $f_{max}=(2-2^{-23})x2^{127}\sim 2^{128}\sim 3.4x10^{38}$.

the smallest normalized number is $f_{min}= 2^{-126}\sim 1.2x10^{-38}$.

Similarly, for double precision, we have $f_{max}=(2-2^{-52})x2^{1023}\sim 2^{1024}\sim 1.8x10^{308}$.

$f_{min}= 2^{-1022}\sim 2.2x10^{-308}$.

Representation of numbers: reals

Specifically, they are represented by $R = (-1)^S \times (1 + F) \times 2^{E-B}$

For single or double precision, *E* is in [1,254] or [1, 2046], the *bias* B is 127 or 1023.

The fraction *F*=($f_1 \ldots f_N$) represents: $F = \sum_{i=1}^{N} 2^{-i} f_i$

Exceptions:

When E=0, (1+F) => F, so that value is 0 for F=0.

For nonzero F, the value is denormalized (gradual underflow with less precision).

When E=255 (or 2047), the value is infinity for F=0.

For nonzero F, the value is NAN (not a number).

Occurs when you /0 or taking sqrt of a negative number.

Rounding and machine precision

The smallest increment in F defines the machine precision ϵ.

For single precision: $\varepsilon=2^{-23}=1.2\times10^{-7}$.

For double precision: $\varepsilon=2^{-52}=2.2\times10^{-16}$.

In other words, the precision of a non-zero value x is about εx.

Floating point operations inevitably produces numbers that demands higher precision, which has to be truncated. This is known as rounding.

Example (decimal): a=1234.567, b=45.67834

a+b=1280.24534 which rounds to 1280.245

At hardware level, 2-3 additional bits are present to ensure correct rounding.

Error associated with rounding is called rounding error, or round-off error, which are at the level of machine precision.

Properties

Can 0.1 be represented?

No. The closest is 0.100000001490116119384765625 in single precision.

Similarly, many familiar numbers can not be represented. Also, π can not be represented, hence sin(π) is nonzero, tan(π/2) is not infinity.

Are additions and multiplications commutable?

Yes. We have a+b=b+a, axb=bxa.

Does associativity law hold [a+(b+c) = (a+b) +c]?

It does not necessarily hold because of rounding.

Does distributive law hold [ax(b+c) = axc + bxc]?

It does not necessarily hold because of rounding.

Lessons to programmers

Use double precision in general unless you don't have to.

Round-off error grows as $N_O^{1/2}$, where N_O is # of operations. For single precision, error becomes order unity for $N_O \sim 10^{14}$.

Many simple calculations require many operations (e.g., division, powers) that accumulate round-off errors faster.

Modern computers do billions of floating point operations per second, and errors easily build up.

At 1 GHz, this will occur well within 10^5s on one processor.

But on 1000 processors, it occurs well within 100 seconds!

Lessons to programmers

Avoid division as much as possible.

Division is typically implemented as computing the reciprocal then multiplying.

It is a factor of several slower than multiplication.

```
for (i = 0; i < N; i++)
{
   a[i] = a[i]/scale;
}
```

This can be inefficient.

```
inv=1./scale;
for (i = 0; i < N; i++)
{
   a[i] = a[i]*inv;
}
```

This is good.

Avoid subtracting two similar numbers.

Suppose you want to compute $y=(1-x)^6$ for $x=0.999$.

Doing it directly: $1-x=0.001$ so that $y=1.0\times10^{-18}$.

What if you expand to $y=1-6x+15x^2-20x^3+15x^4-6x^5+x^6$=?

Summary

- Modern computers employ a variety of parallelization mechanisms

  Instruction level parallelism: pipelining, multi-issue

  Data-level parallelism: SIMD

  Task-level parallelism: multiprocessors and clusters

  Avoid data hazards and control hazards, write vectorized codes.

- Memory performance is often the bottleneck of computing

  Memory design: hierarchy with multi-level caches.

  Access data contiguously to maximize cache performance.

- Computer arithmetic

  Floating point representation, machine precision and round-off error

  Use double precision, avoid division and subtracting two large numbers.