Programmazione e Calcolo Scientifico

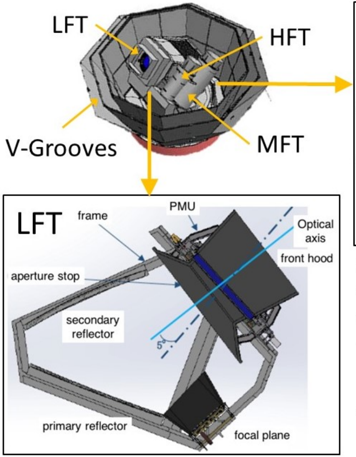
Matteo Cicuttin

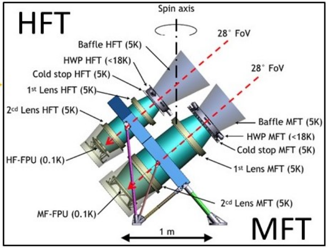
Politecnico di Torino

March 9, 2024

What do we do with scientific computing?

LiteBIRD Payload Module





Main features:

- Three telescopes with TES detector arrays
- Frequency range 40 – 450 GHz
- Polarization modulation in each telescope with rotating half-wave plate (HWP) for 1/f noise & systematics reduction
- Cryogenic systems for 0.1 K detectors' temperature

What do we do with scientific computing?

Click to play

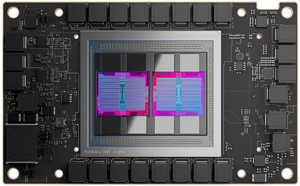
LiteBIRD is a satellite that will be launched in 2029. It will be used to observe the CMB.

- Simulation of one of the LiteBIRD telescopes
- A couple of hours on 4 AMD MI250 GPUs

LUMI supercomputer

ww.top500.org/lists/top500/2023/06/

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,194.00	1,679.82	22,703
2	**Supercomputer Fugaku** - Supercomputer Fugaku, [illegible] 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational [illegible] Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	309.10	428.70	6,016



Course outline

In the era of Artificial Intelligence, we must be very well aware that computers are actually very stupid machines with **definite limits and capabilities**. There's no magic. Santa Claus does not exist.

We want to make you aware of those limits and give you some tools to maximally exploit computer capabilities.

Some topic we will cover:

- Anatomy of a computer (processor, memory, bus, cache)
- The limitations of a computer
- C++ programming
- Some simple algorithms and data structures

The AXPY operation

BLAS stands for Basic Linear Algebra Subprograms. It is a set of standard functions implementing the most common operations occurring in scientific computing:

- Level 1: vector-vector operations, $O(n)$ complexity
- Level 2: matrix-vector operations, $O(n^2)$ complexity
- Level 3: matrix-matrix operations, $O(n^3)$ complexity

The AXPY operation

BLAS stands for Basic Linear Algebra Subprograms. It is a set of standard functions implementing the most common operations occurring in scientific computing:

- Level 1: vector-vector operations, $O(n)$ complexity
- Level 2: matrix-vector operations, $O(n^2)$ complexity
- Level 3: matrix-matrix operations, $O(n^3)$ complexity

One of the operations included in BLAS is the AXPY: $\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$ where $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^N, \alpha \in \mathbb{R}$.

```
for (size_t i = 0; i < N; i++)
    y[i] = alpha*x[i] + y[i];
```

Anatomy of AXPY

Assume to run AXPY on a vector of size N:

```
for (size_t i = 0; i < N; i++)
    y[i] = alpha*x[i] + y[i];
```

Observations:

- N cycles
- For each cycle: 1 multiplication and 1 addition

Total cost: $2N$ FLOPS (FLOP = FLoating point OPeration).

Let's measure how much time an AXPY takes and determine speed in FLOPS/s.

An hypotetical XNPY operation

We define XNPY as follows (XNPY is **NOT** part of BLAS). Let $\boldsymbol{x}, \boldsymbol{y}$ be two vectors of equal length:

$$y_i \leftarrow x_i^n + y_i \qquad i \in 1 \dots \text{length}(\boldsymbol{y})$$

Translated to code:

```
for (size_t i = 0; i < N; i++) {
    double xpow = 1.0;
    for (size_t p = 0; p < pow; p++)
        xpow *= x[i];
    y[i] = xpow + y[i];
}
```

For each **outer** cycle we do `pow+1` FLOPS.

Let's measure the speed of XNPY.

Any idea to improve AXPY?

Why AXPY is so slow compared to XNPY? Any idea to improve it?

```
for (size_t i = 0; i < N; i++)
    y[i] = alpha*x[i] + y[i];
```

Matrix multiplication

Let $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{R}^{N \times N}$. The matrix multiplication $\boldsymbol{C} = \boldsymbol{AB}$ is computed as $c_{ij} = a_{ik} b_{kj}$. This is probably the most important operation in scientific computing.

The code for the matrix multiplication looks like this:

```
for (size_t i = 0; i < N; i++)
    for (size_t j = 0; j < N; j++)
        for (size_t k = 0; k < N; k++)
            c[I(i,j)] += a[I(i,k)] * b[I(k,j)];
```

Total num of FLOPs is $2N^3$. Let's do some measurements!

Another matrix multiplication

We reorder the operations in our matrix multiplication:

```
for (size_t i = 0; i < N; i++)
    for (size_t k = 0; k < N; k++)      // swapped this
        for (size_t j = 0; j < N; j++)  // and this
            c[I(i,j)] += a[I(i,k)] * b[I(k,j)];
```

Total num of FLOPs is **again** $2N^3$. Let's do some measurements!

What happens to performance?

- Why the performance of XNPY is better than AXPY despite being more complex?
- Why parallel XNPY improves but parallel AXPY does not?
- Why swapping two cycles in matrix multiplication changes the execution time so much?
- More generally, how do you choose a computer to solve a specific problem?

What happens to performance?

- Why the performance of XNPY is better than AXPY despite being more complex?
- Why parallel XNPY improves but parallel AXPY does not?
- Why swapping two cycles in matrix multiplication changes the execution time so much?
- More generally, how do you choose a computer to solve a specific problem?

The answer to these questions is rooted deeply

- in the way computers work
- in the way compilers work

Some tales about floating point

Before being fast, code must be correct.

Your code can go infinitely fast if you don't care about correctness, but people won't be interested in offering you a good job.

I - The Chaotic Bank Society

Your bank proposes to you to open a new account with the following scheme:

- When you open the account, you deposit $e - 1$ euros
- Each year the bank multiplies your savings by the number of years, but it takes a fee of 1 euro.

$$\begin{cases} b_0 = e - 1 \\ b_i = i \cdot b_{i-1} - 1 \end{cases}$$

You make a little program to figure out what happens after 25 years:

```
account = 1.718281828459045
year = 1
while year <= 25:
    account = year*account - 1
    year = year + 1
```

Would you open the account?

I - The Chaotic Bank Society: the lesson

Let's unroll the series:

$$b_0 = e - 1 \qquad\qquad b_2 = 2b_1 - 1 = 2(1b_0 - 1) - 1$$

$$b_1 = 1b_0 - 1 \qquad\qquad b_3 = 3b_2 - 1 = 3(2(1b_0 - 1) - 1) - 1$$

You soon figure out that

$$b_n = n! \times \left(b_0 - 1 - \frac{1}{2!} - \frac{1}{3!} - \ldots - \frac{1}{n!}\right) = n! \times \left(b_0 - \sum_{i=1}^{n} \frac{1}{i!}\right)$$

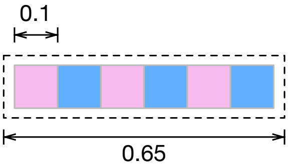
And you recall that (Maclaurin expansion)

$$e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

If $b_0 = e - 1$, then b_n should converge to zero! You can't represent $e - 1$ exactly, error is accumulating like crazy!

II - The missing block

How many blocks of width w_b fit in a space of width w_s? Of course $\lfloor w_s/w_b \rfloor$. For example, $\lfloor 0.65/0.1 \rfloor = 6$.




```
double total_width = 0.65;
double block_width = 0.1;
int blocks = (int) floor(total_width/block_width);
```

What happens with $w_s = 0.6$?

II - The missing block: the lesson

Computers have only **finite precision**: in base 2, the numbers $1/10$ and $6/10$ do not have a finite decimal expansion. Notice that in base 10 you have exactly the same problem with for example $1/3$.

- $\mathbb{F}(0.6) = 0.59999999999999978.$
- $\mathbb{F}(0.1) = 0.10000000000000006.$
- In general, $\mathbb{F}(x)$ can be **slightly smaller or slightly greater** than x and this can have really nasty effects.

III - The odometer that stopped

Design an *odometer* that keeps track of the distance with a precision of 0.01 meters.

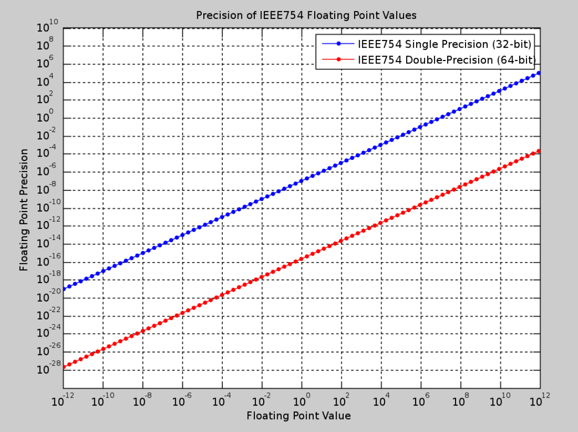
```
float meters;

void interrupt isr() {
  if (odometer_interrupt)
    meters += 0.01f;
}

int main(void) {
  meters = 0;
  while(1)
    ;
}
```

III - The odometer that stopped: the lesson

Remember that in floating point **precision is not constant**: it depends on the magnitude of the number you are representing.



Conclusions

Knowing how a computer works allows you to

- recognize simple performance issues and not end up with super slow code
- recognize some potentially dangerous operations that will give you incorrect results

Modern engineering needs strong interdisciplinary skills, and numerical computing is a central tool. Even if you only write small Matlab prototype programs, computer architecture somewhat affects you.

Also if you are a theoretical person, some computer architecture is needed: you could develop the best and most beautiful algorithm in the world, but if from a practical point of view is not good no one will use it.