
# Optimization and Vectorization

2024-05-21

By: Bart Oldeman
slides also by: Pier-Luc St-Onge

# Nos membres et partenaires

# Members and partners

Partenaire de l'
Alliance de recherche numérique du Canada

Partenaires connectivité

Partenaires financiers

# Présentation du formateur

# Instructor presentation

- Advanced Research Computing Analyst
  - McGill team, Calcul Quebec since July 2012
  - Taught at the spring school since 2015
- Speciality
  - Ph.D. in Engineering Maths, University of Bristol, UK, 2001
  - Various postdocs followed in the UK, USA, New Zealand and Canada, on the intersection between Dynamical Systems and Numerical Analysis.

# Cet après-midi

# This afternoon

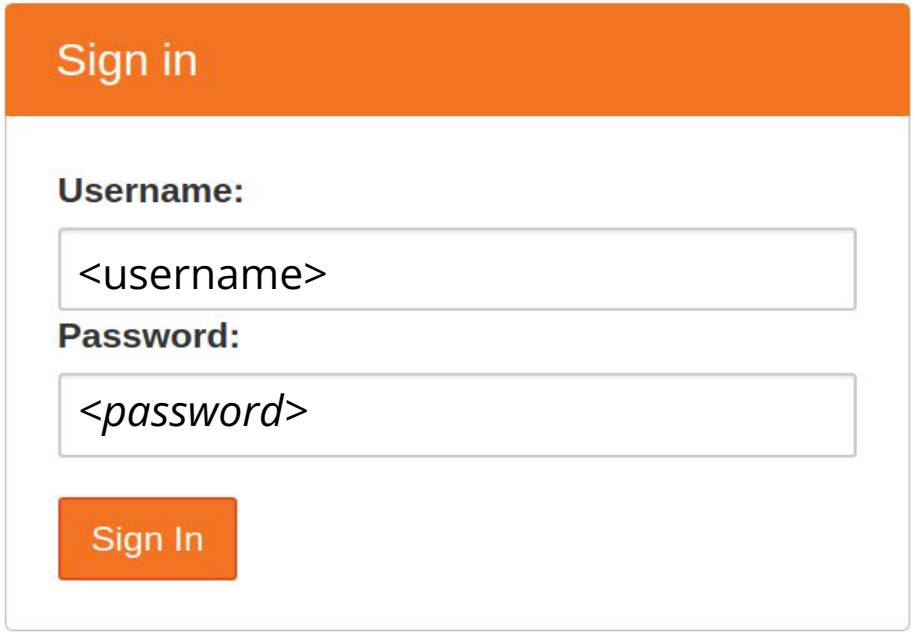
- Exercic/ses;
- Questions;

De 13h30 à 16h, 1:30pm to 4pm.

# Accéder à la plateforme

# Accessing the platform

Navigateur web browser:
**p-ecole.calculquebec.cloud**



# Exercise 0: log in to Cloud VM

- Connect to the Magic Castle platform via https:/p-ecole.calculquebec.cloud
  - Create account via link on page
  - Start a “Terminal” in the launcher
- Change directories to workshop material
  - `git clone https://github.com/calculquebec/cq-formation-convolution.git`
  - `module load StdEnv/2023 scipy-stack`
  - `cd cq-formation-convolution/noyaux; make`
  - `cd ~/cq-formation-convolution/solutions/optimisation`

# Introduction to Optimization

# What you should do!

- Make good (i.e. well structured) code first
  - Do not reinvent the wheel: try using existing optimized libraries
  - Use functions when long code gets too complicated
    - Move if/for/while inner code to a function
  - Use classes where object-oriented programming feels right. See Software design patterns
- Profile your code
  - Identify which functions and which loops are taking most of the compute time
- Then, you may try to optimize your code

# Where most time is spent

- CPU
  - We want to maximize CPU time
  - Efficiency of instruction sets (i386, mmx, sse, avx, etc.)
  - Optimization at compilation level (-O2, -O3, etc.)
  - Compiler & version (gcc/12, intel/2023)
- Memory access
  - Registers, caches
  - RAM, local hard drive, network caches
- Network
  - Latency, bandwidth
- Which parts of your code?

# Best practices

“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.” – Donald Knuth

- The best way to optimize is to choose a good algorithm, but …
- … do not try to optimize from the beginning of your project.
- More than 80% of execution time is in less than 20% of code.
- Validate your code after each step of optimization

# Serial and parallel tasks

- Serial tasks (even in a parallel job)
  - Not divisible (mandatory cost in time)
  - Initiating the parallel environment
  - Loading a file, writing to a file (often)
  - Collective communications (parts)
  - Anything else done by one thread only
  - Anything else done by one thread at a time
- Parallel tasks
  - The work is divisible in multiple tasks
  - Single instruction, multiple data (SIMD)
  - Any distinctive tasks that could be done by different threads and/or processes

# Amdahl’s law

- General speedup ratio (with `time` command):

$$\text{speedup} = \frac{\text{elapsed time for serial code}}{\text{elapsed time for parallel code}}$$

- Amdahl’s law for parallel code:

$$\text{speedup} = \frac{1}{S + P/n}$$

where n is the number of processes or threads, P is the parallel fraction of the code ($0 \leq P \leq 1$), and S is the serial fraction of the code ($S = 1 - P$).

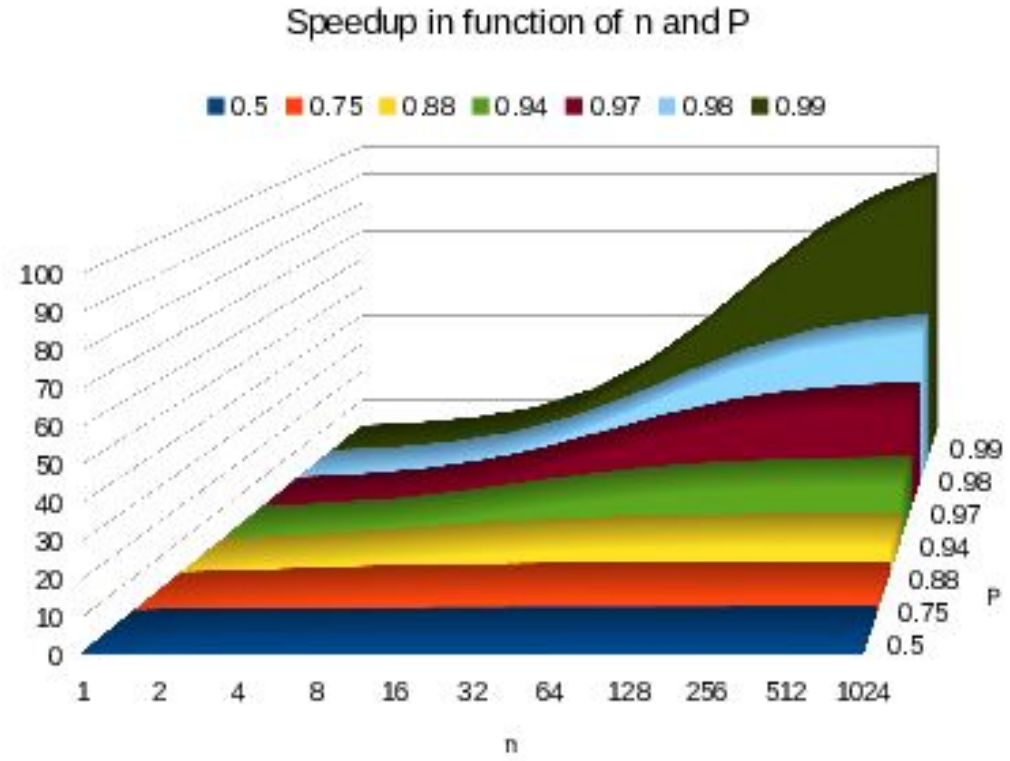
# Amdahl's law to the limit

- What if we have unlimited computing resources ($n \to \infty$) on a perfect network (high bandwidth, low latency):

$$\text{speedup} = \frac{1}{S + P/n} \longrightarrow \frac{1}{S} = \frac{1}{1-P}$$

- Example with S = 0.0035 (or P = 0.9965):
  - For $n \to \infty$, the maximum speedup is 285
  - But for n = 1024, the speedup is only 223

# Amdahl’s law to the limit



# Karp-Flatt metric

- Approximation of P (the parallel fraction of the code)

$$\text{speedup} = \frac{1}{S+P/n} \Rightarrow P = \frac{n(1-1/\text{speedup})}{n-1}$$

- We can also determine the experimentally determined serial fraction e given measured speedup.

$$e = S = 1-P = 1 - \frac{n(1-1/\text{speedup})}{n-1}$$

- Example: n = 2, speedup = 1.95, e = 0.026.
- Example: n = 1024, speedup = 200, e = 0.0040.

# Karp-Flatt metric (2)

- An acceptable speedup is at least 80% ∗ n = 0.8n

$$\text{speedup} = \frac{1}{S+P/n} \geq 0.8n \Rightarrow n \leq \frac{1/0.8 - P}{S}$$

- Given an experimentally determined serial fraction e = S = 0.01 (P = 1 − 0.01 = 0.99), what would be the highest acceptable n?

$$N \leq \frac{1.25 - 0.99}{0.01} = 26$$

# Algorithms

- A problem can be solved in many different ways
- Relative complexity:
  - Number of operations
  - Amount of memory
  - Complexity for n elements: O(1), O(log n), O(n), O(n log n), O($n^2$), etc.
  - If n is small, choose lightest algorithm
- Example: sorting algorithms
  - https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms

# Profiling your Code

# Profilers

https://docs.alliancecan.ca/wiki/Debugging_and_profiling

- gprofng (**`gprofng`**) (gprof is mostly redundant)
  - https://sourceware.org/binutils/wiki/gprofng
- Linux perf (**`perf`**)
  - https://perf.wiki.kernel.org/index.php/Main_Page
- Nvidia command-line profiler (**`nvprof`**)
  - https://docs.alliancecan.ca/wiki/Nvprof
- Python Profilers (**`cProfile`**, **`profile`**)
  - https://docs.python.org/3/library/profile.html
- R profiler (**`Rprof`**)
  - https://www.rdocumentation.org/packages/utils/versions/3.5.2/topics/Rprof

# More Profilers

https://docs.alliancecan.ca/wiki/Debugging_and_profiling

- ARM MAP (comes with DDT)
  - https://docs.alliancecan.ca/wiki/ARM_software
- Intel VTune
  - https://software.intel.com/en-us/vtune
- Intel Advisor
  - https://software.intel.com/en-us/advisor
- Tau
  - http://www.cs.uoregon.edu/research/tau/home.php
- HPC Toolkit
  - http://hpctoolkit.org/
- Valgrind
  - https://docs.computecanada.ca/wiki/Valgrind

# Example - How to use `gprofng`

- Compile your code with debugging information enabled

  **`gcc -g -o executable code.c`**
- Execute your code using gprofng collect app

  **`gprofng collect app executable args`**
- Then, get readable profiling information which parses files in a directory `test.<n>.er` written by `gprofng collect app`:

  **`gprofng display text -lines test.1.er`**

# Example - How to use `perf`

- Compile your code with debugging information enabled

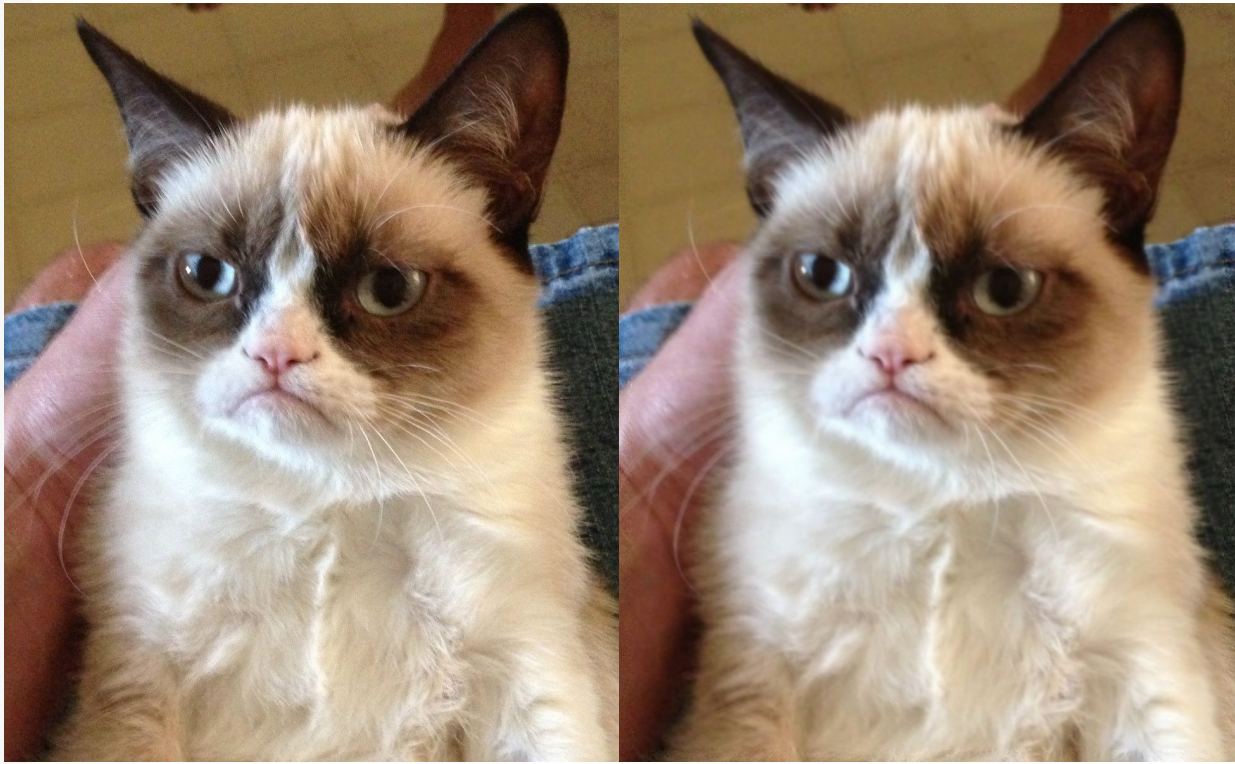
  **`gcc -g -o executable code.c`**
- Execute your code using perf record / stat

  **`perf stat -d executable arg1 arg2`**

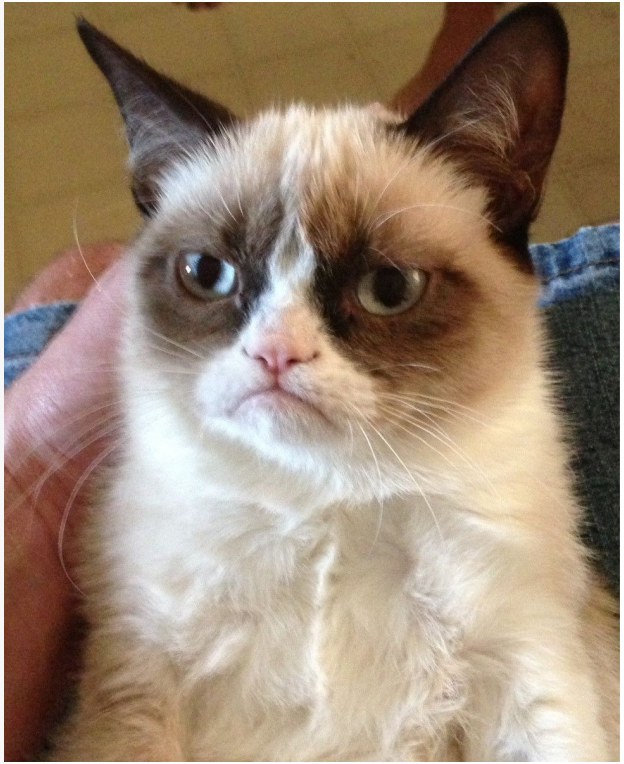
  **`perf record executable arg1 arg2`**
- Then, get readable profiling information which parses the file `perf.data` written by `perf record`:

  **`perf report`**
- See also https://jvns.ca/perf-zine.pdf

# Example 1: convolution

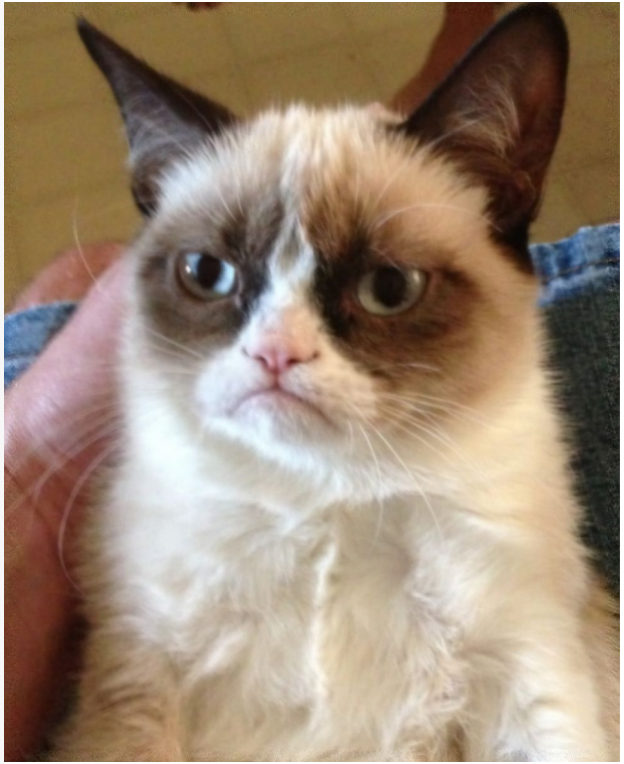


# Example 1: grumpy cat



973 x 1200 PNG image
Every pixel undergoes a filter (noyau_flou_45): weighted average of the red/green/blue values of a square of 45x45 pixels with the current pixel at the centre.
The weights are in the file noyau_flou_45.

# Example 1: blurry fuzzy cat



973 x 1200 PNG image
Every pixel undergoes a filter (noyau_flou_45): weighted average of the red/green/blue values of a square of 45x45 pixels with the current pixel at the centre.
The weights are in the file noyau_flou_45.

# Exercise 1 : using `gprofng`

1.

1. Compile exercise 1 code with debug information, noting the -g flag:
   ```
   $ make -C ../..
   $ make convolution
   ```
2. Run:
   ```
   $ gprofng collect app -O test.1.er
   ./convolution exemple.png
   noyau_flou_45
   ```
3. Check correctness
   ```
   $ md5sum resultat.png
   685abc74c4139719155edc84d2c10dde
   resultat.png
   ```
4. Look for results:
   ```
   $ gprofng display text -lines
   test.1.er
   ```

# Exercise 2 : using `perf`

1.

1. Run:
   ```
   $ perf stat -d ./convolution exemple.png noyau_flou_45
   $ perf record ./convolution exemple.png noyau_flou_45
   ```
2. Look for results:
   ```
   $ perf report
   ```

# Exercise 3 : python version

1.

1. Run:
   ```
   $ time python
   ../../defi-mpi/python-conv.py
   exemple.png noyau_flou_45
   ```
2. For this afternoon: Python version can be optimized using np.convolve and scipy.signal.convolve

# Optimizations

# Two ways of optimizing a code

1)

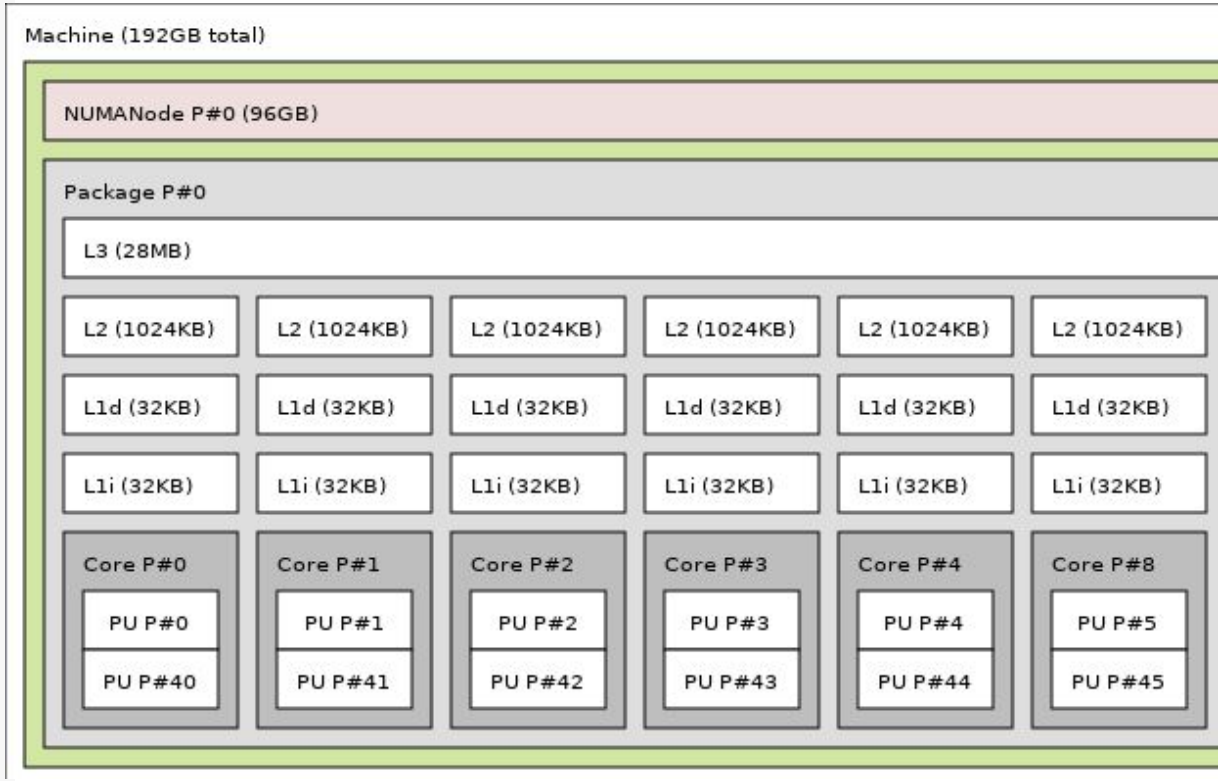
1) Review chosen algorithms and data structures vs the size of the problem
   a) Use arrays (vectors, matrices), lists, sets and dictionaries where appropriate
   b) Sometimes, a compute task is so small that even a naive algorithm may outperform a clever one
2) Take into account the computer architecture
   a) Memory architecture
   b) Available CPU instructions

# Computer Architecture

# Non-uniform memory architecture (NUMA) with Intel

- `lstopo --no-io --of png node.png`

# Non-uniform memory architecture (NUMA) with Intel



# Data Proximity

From:
https://www.7-cpu.com/cpu/Skylake_X.html

- Cache line: 64 Bytes or 512 bits
- L1 cache = 2* 32 KB, latency of 4-5 cycles
  - Average L1 write latency: 0.5 cycles per access
- L2 cache = 1024 KB, latency of 14 cycles
  - Average L2 write latency: 3.2 cycles per cache line
- L3 cache = 27.5 MB, latency of 45 cycles
  - Average L3 write latency: 10 cycles per cache line
- RAM latency: >45 cycles + 50ns
  - Probably more between CPU sockets

# Data Proximity

Whenever possible

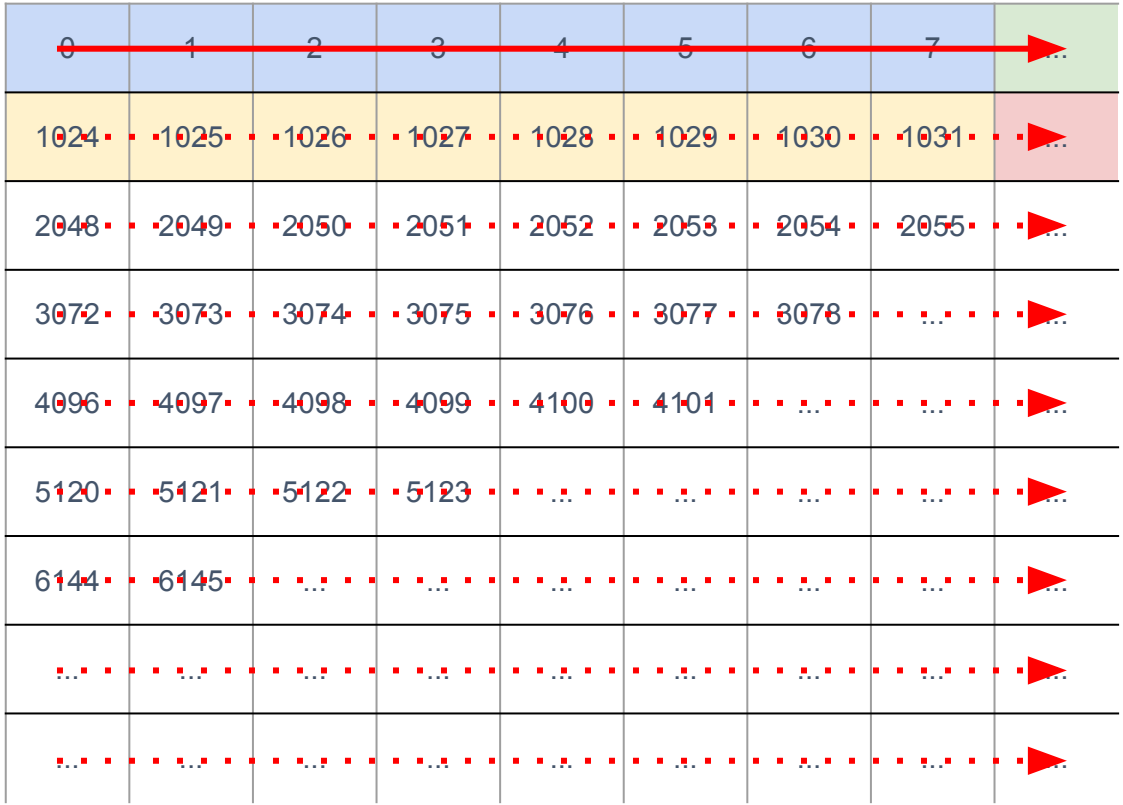
- Reuse the same data in L1 (up to 32 KB per core), in L2 (from 256 KB to 1 MB per core) and in L3 (around 27.5 MB per socket)
  - Maximize the amount of operations on this data
  - Efficient parallel codes will run entirely in caches, which avoids access to the big RAM
- Do not waste access to memory
  - Remember: 64 B/cache line = 8 int64 or pointers
  - Jumps in memory space should be done carefully

# 2D Arrays and Contiguous vs Random Access

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... |
| 1024 | 1025 | 1026 | 1027 | 1028 | 1029 | 1030 | 1031 | ... |
| 2048 | 2049 | 2050 | 2051 | 2052 | 2053 | 2054 | 2055 | ... |
| 3072 | 3073 | 3074 | 3075 | 3076 | 3077 | 3078 | ... | ... |
| 4096 | 4097 | 4098 | 4099 | 4100 | 4101 | ... | ... | ... |
| 5120 | 5121 | 5122 | 5123 | ... | ... | ... | ... | ... |
| 6144 | 6145 | ... | ... | ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |

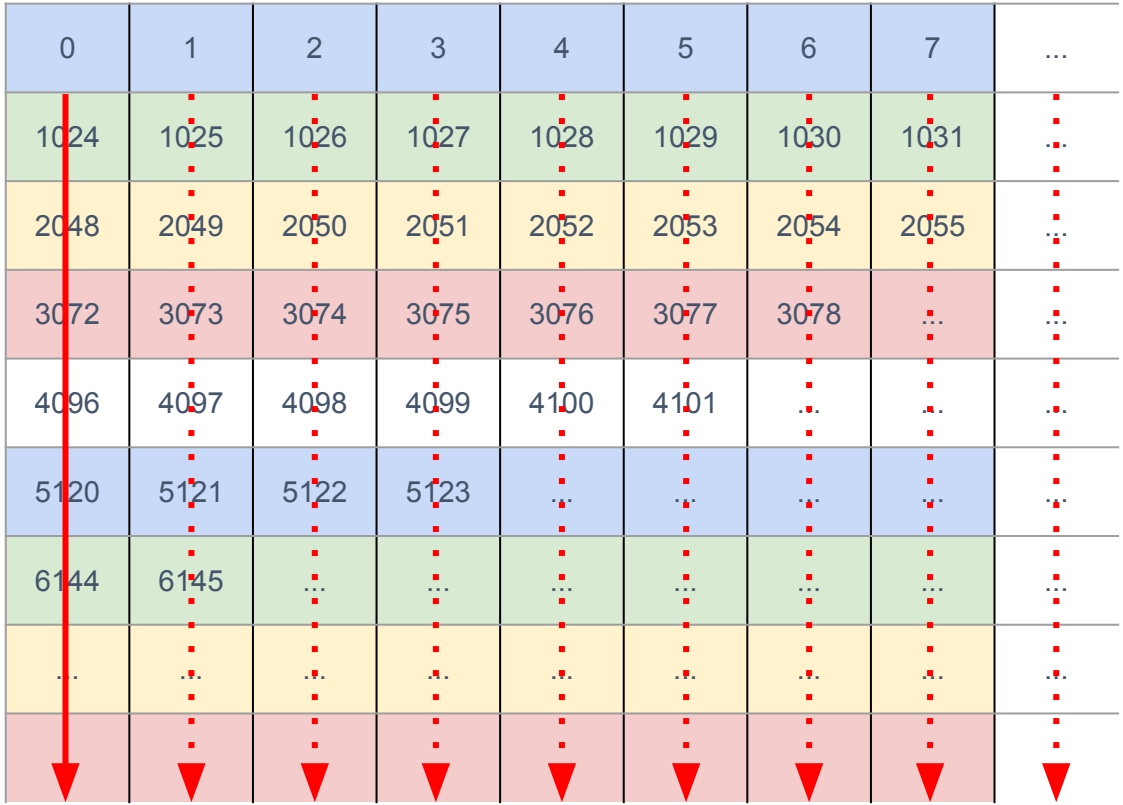
- N*1024 2D array
- Values are indexes

# 2D Arrays and Contiguous vs Random Access




- Horizontal access = contiguous access
- Each value is used once, but at least no cache line is wasted

# 2D Arrays and Contiguous vs Random Access




- Vertical access = jumping in memory
- If N is large enough, going from row 0 through N-1 will continuously waste lots of cached data

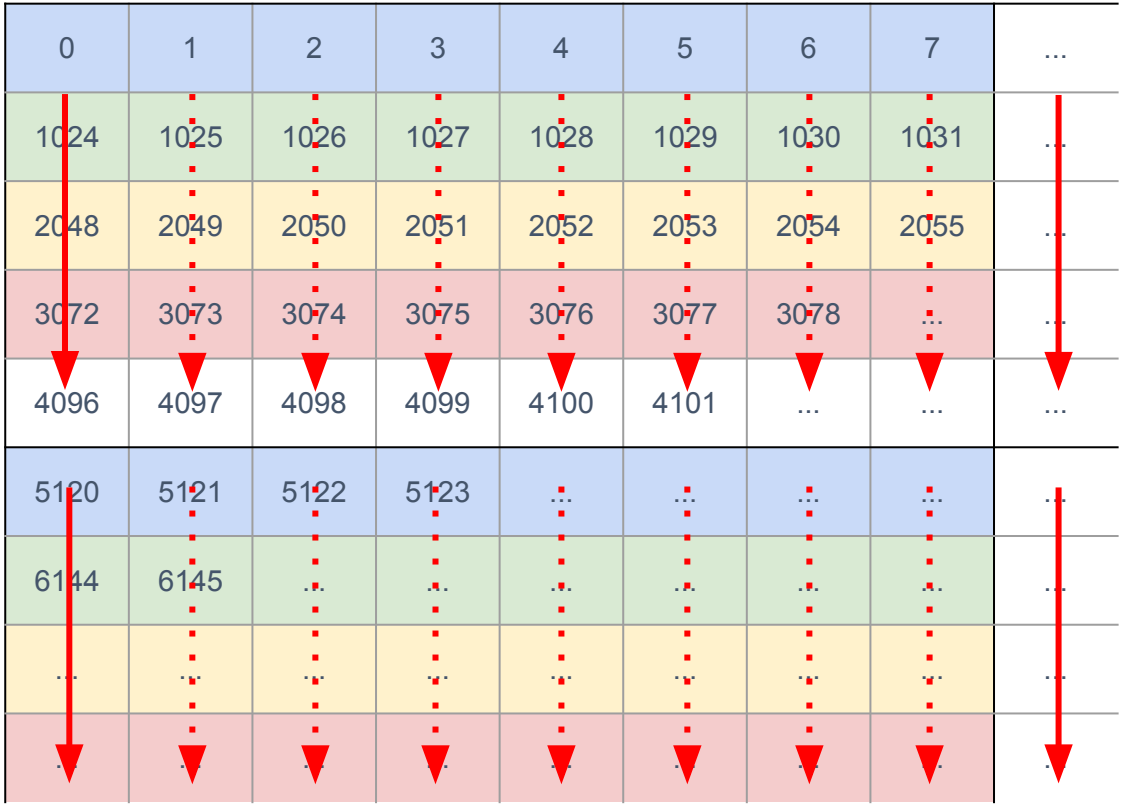
# Example Matrix-vector multiplication

run:

```
cd ~
module load flexiblas
git clone
https://github.com/calculquebec/cq-fo
rmation-matrice-vecteur
cd cq-formation-matrice-vecteur
make
export OMP_NUM_THREADS=1
./matvec
```

This compares a naive inefficient matrix-vector multiplication with the optimized version from MKL via FlexiBLAS. We'll try to optimize this.

# 2D Arrays and Access by Blocks of Data




- Vertical access = jumping in memory
- Vertical access is OK if cached data is used as soon as possible OR as much as possible
- Blocks should fit in L1, L2, L3

# About the Matrix Multiplication

```
for i = 1 .. N
  for j = 1 .. N       # For each C[i,j] value
    C[i,j] = 0         # Initialize with 0
    for k = 1 .. N     # Compute dot-product
      C[i,j] += A[i,k] * B[k,j]   # Jumps in B

# Solution 1: transpose content of B (A[i,k] *
                                      B[j,k])
# Solution 2: exchange both inner for loops
for i = 1 .. N
  for j = 1 .. N       # Whole row C[i,:] in cache
    C[i,j] = 0         # Initialized with 0s
  for k = 1 .. N       # A[i,k] is fixed for each j
    for j = 1 .. N     # Whole row B[k,:] is used
      C[i,j] += A[i,k] * B[k,j]   # Contiguously
```

# To Keep in Mind...

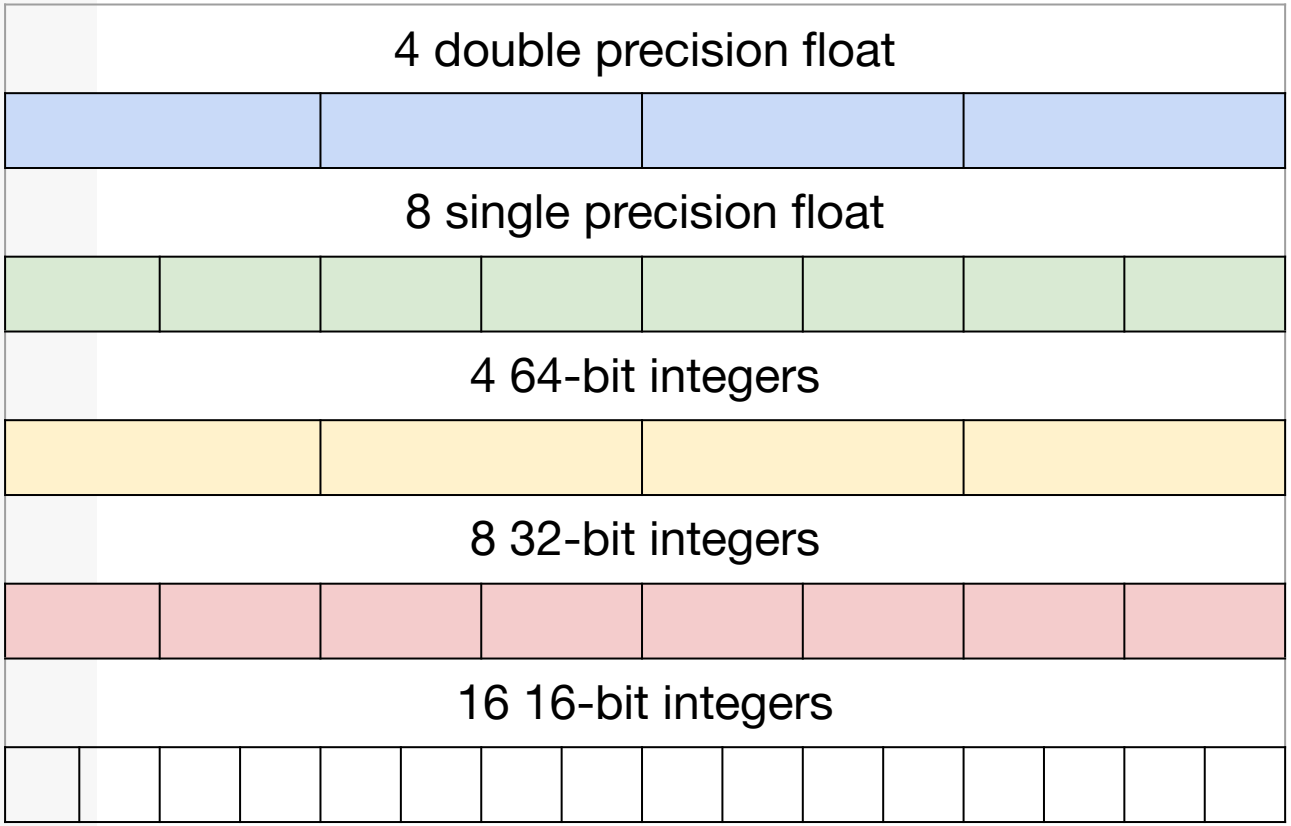
- Do not reinvent the wheel!
  - There are already many optimized libraries that can do operations on matrices
  - But for smaller amounts of data, like 2x2 matrices, a different (custom-made) algorithm may be better
- Do not completely fill L1, L2 and L3 caches
- Modern compilers may figure out what you are doing in your original algorithm and optimize memory access automatically
  - Therefore, make sure the code is clean!

# Vectorization

# About Vectorization

- Modern processors and accelerators (like GPUs) can apply the same basic operation on multiple data at once
  - SIMD: Single Instruction on Multiple Data
- Have you ever heard about x86, MMX, SSE2-SSE4.2, AVX, AVX2 and AVX512?
  - These are CPU instruction sets
- All national systems support at least AVX2
  - More SSE and AVX instructions in 256 bits
  - While AVX512 has instructions in 512 bits

# What Fits in 256-bit registers?

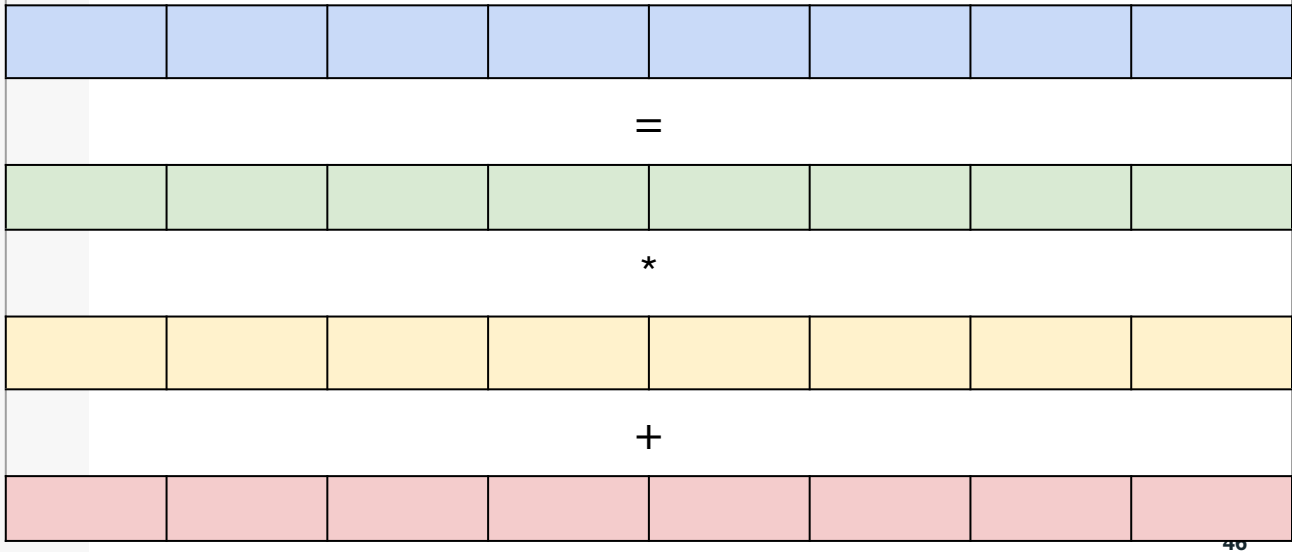


# Example - Fused Multiply-Accumulate (FMA)

- With old x86:

```
for i = 0 .. 7
  A[i] = B[i] * C[i] + D[i]
```

- But with SIMD, this can be done with four wide registers and a single CPU instruction:



# For Good Vectorization

- Align similar data on cache lines (512 bits)
  - You may consider over-allocation for each row in a 2D array
- Recent C/C++ versions come with tools to allocate memory-aligned buffers
  - https://en.cppreference.com/w/c/memory/aligned_alloc
  - https://en.cppreference.com/w/cpp/memory/align
- Use structures of vectors instead of vectors of structures. See next slide ->

# Using a Structure of Vectors

- Instead of:

```
class Point3D {x,y,z}
Point3D array[N] #
x1,y1,z1,x2,y2,z2,...
```

- Consider using:

```
class Ptr3D {*x,*y,*z,resize()}
Ptr3D array    # x1,x2... y1,y2...
z1,z2...
array.resize(N)  # x, y and z with N
values
```

# Loop vectorization

- Compilers are now able to identify loops doing independent and identical operations:
  - No dependency between iterations (indexes i and i − 1, for example)
  - The execution path must be the same: be careful with if, switch, break, while and for statements
  - Function calls are allowed if they follow the above rules
  - It works very well with vectors or arrays
- This will be vectorized in chunks of 2, 4, 8 doubles (CPU dependent):

```
for (i = 0; i < N; i++) c[i] = a[i] * b[i];
```

Can be forced with OpenMP 4+: `#pragma omp simd`

# Flops: floating point operations per second

Modern CPUs:
Peak double precision flops =
CPU GHz * #cores * FMA units * (multiplication + addition) * vector length.
E.g. for Narval CPUs (AMD 7532)
2.4 GHz * 32 * 2 * 2 * 4 = 1228.8 Gflops

BUT can only be reached for compute intensive applications (high Flop/byte), e.g. matrix multiplication.

Matrix-vector multiplication “streams” memory, only reading the matrix once…

# Memory bandwidth: Gigabyte/s

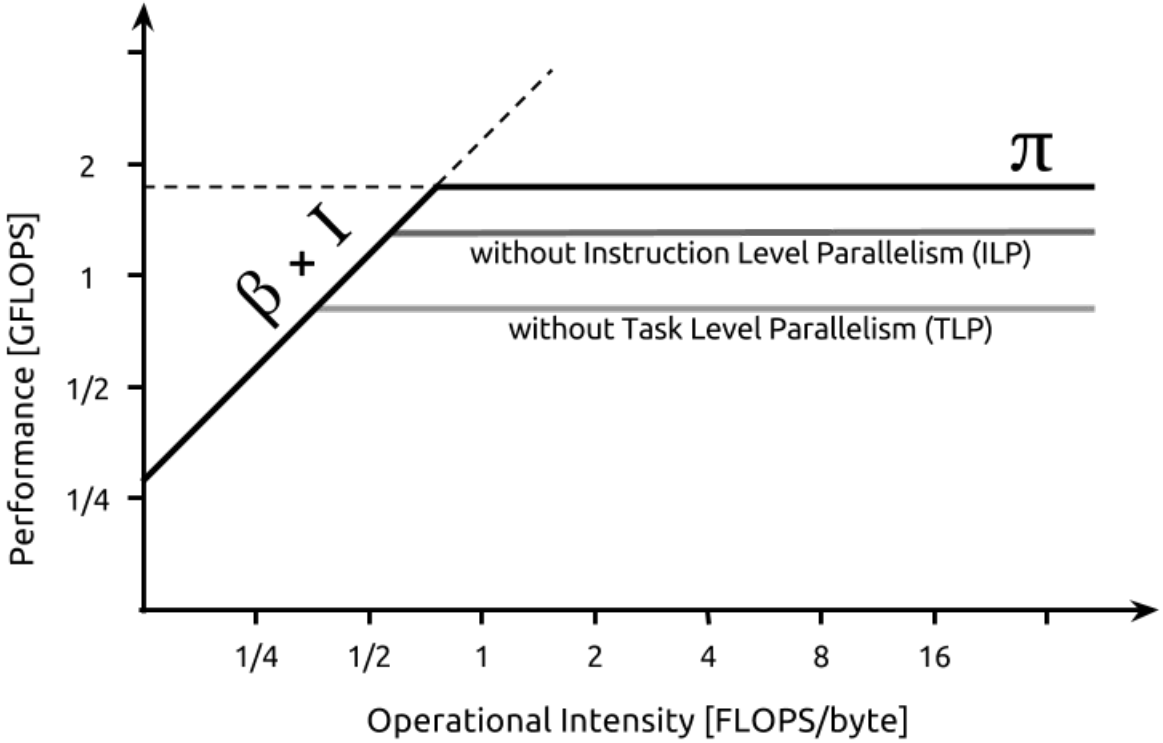
Streaming memory applications (e.g. matrix-vector multiplication), limited by memory speed.

- E.g. (AMD Rome):
- One core: 32 GB/s
- Peak: 210 GB/s: with 7 cores out of 32 cores used on a socket you've already reached peak bandwidth!

Latency-bound applications (random memory accesses), e.g. databases even worse:

- CPU spends most of its time waiting for memory reads/writes
- May benefit from hyperthreading
- Not extremely common in HPC

# Roofline model




https://upload.wikimedia.org/wikipedia/commons/c/c0/Roofline_model_in-core_ceilings.png

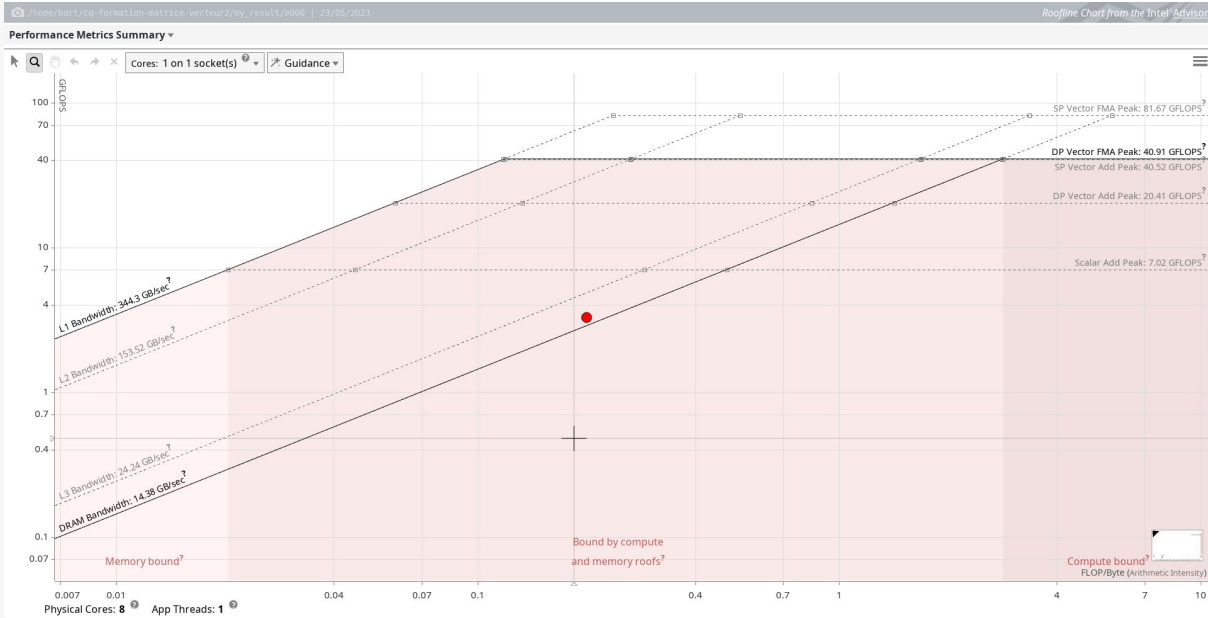
# Roofline using Intel Advisor

- Create roofline graph in Advisor:

```
advixe-cl -collect survey
-project-dir my_result --
./matvec
advixe-cl -collect tripcounts
-flop -project-dir my_result --
./matvec
advixe-gui my_result
```

- In navigation panel, click black icon under “Run Roofline”
- Click “Survey and Roofline”, then click vertical bar “ROOFLINE”

# Roofline using Intel Advisor for optimized matrix-vector multiplication



# Example: optimizing convolution

1. `1_convolution_double`: Use an image array of doubles to avoid conversions from char to double in the inner loops.
2. `2_convolution_soa`: Use structure of arrays for the image.
3. `3_convolution_omp_simd`: Use #pragma omp simd for the inner loop so the compiler can optimize the summing

# Example: optimizing nbody

Runge-Kutta integration of n particles with random initial positions, velocity, and mass, interacting with each other using gravity.

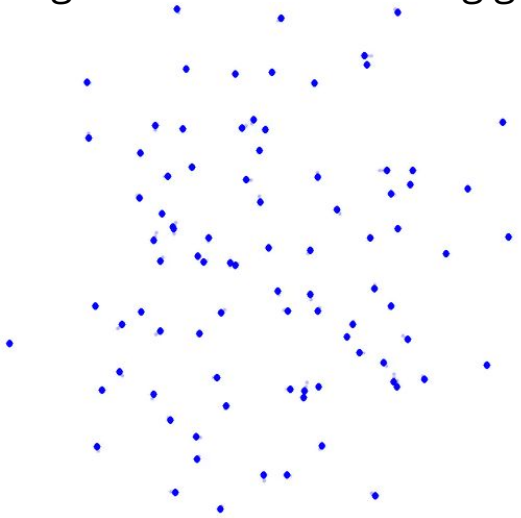


Image from: https://medium.com/swlh/create-your-own-n-body-simulation-with-python-f417234885e9

# Example: optimizing nbody

1.

This example can be obtained via

```
cd
git clone \
https://github.com/calculquebec/cq-formation-nbody
cd cq-formation-nbody
cd solutions/optimisation
```

1. `1_nbody_transpose`: Transpose arrays to allow better data locality with longer inner loops.
2. `2_nbody_permute_loops`: Provide better data locality and vectorization possibilities in inner loops.
3. `3_nbody_sqrtf`: Use the “float” one-over-square-root to gain performance BUT with reduced precision.

# Results: optimizing convolution and n-body (in secs)

| Program | Run time (seconds) |
|---|---|
| convolution | 4.2 |
| 1_convolution_double | 4.1 |
| 2_convolution_soa | 4.2 |
| 3_convolution_omp_simd | 2.6 |
| python_conv | 15.0 |
| nbody | 13.4 |
| 1_nbody_transpose | 14.4 |
| 2_nbody_permute_loops | 11.6 |
| 3_nbody_sqrtf | 7.9 |
| python-nbody | 178 |

# Conclusion

Get things correct before getting it optimized. Reminder:

1) Create well structured code with functions, classes and calls to optimized libraries
2) Profile your code
3) Then you can try to optimize your code
    a) Study memory access
    b) Vectorize the code to allow SIMD