ESCAPE 824064

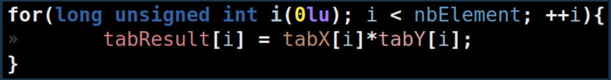
Laboratoire d'Annecy de Physique des Particules

Optimisation : The Hadamard Product

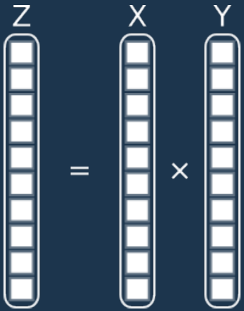
Pierre Aubert

The Hadamard product

$$z_i = x_i \times y_i, \quad \forall i \in 1, N$$



```
for(long unsigned int i(0lu); i < nbElement; ++i){
»       tabResult[i] = tabX[i]*tabY[i];
}
```



Compilation options

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Compilation options

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

- **-O0**
  - Try to reduce compilation time, but **-Og** is better for debugging.

Compilation options

`https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

- **-O0**
  - Try to reduce compilation time, but **-Og** is better for debugging.
- **-O1**
  - Constant forewarding, remove dead code (never called code)...

Compilation options

`https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

- **-O0**
  - Try to reduce compilation time, but **-Og** is better for debugging.
- **-O1**
  - Constant forewarding, remove dead code (never called code)...
- **-O2**
  - Partial function inlining, Assume strict aliasing...

Compilation options

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

- -O0
  - Try to reduce compilation time, but **-Og** is better for debugging.
- -O1
  - Constant forewarding, remove dead code (never called code)...
- -O2
  - Partial function inlining, Assume strict aliasing...
- -O3
  - More function inlining, loop unrolling, partial vectorization...

Compilation options

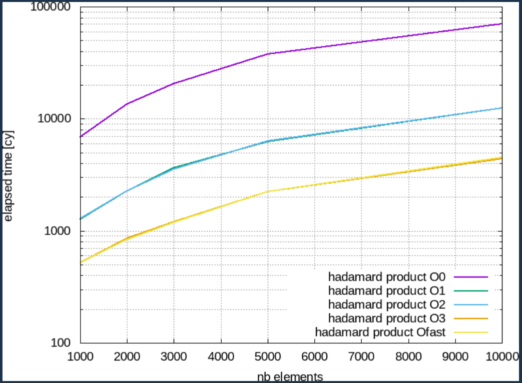
`https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

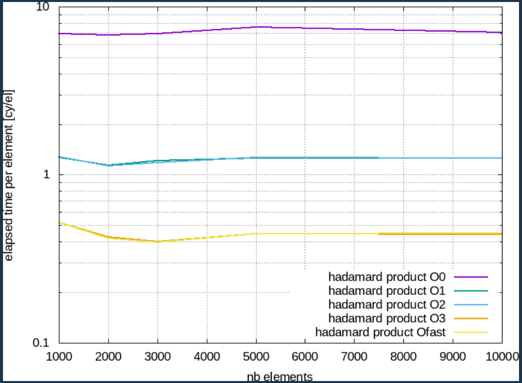
- **-O0**
  - Try to reduce compilation time, but **-Og** is better for debugging.
- **-O1**
  - Constant forewarding, remove dead code (never called code)...
- **-O2**
  - Partial function inlining, Assume strict aliasing...
- **-O3**
  - More function inlining, loop unrolling, partial vectorization...
- **-Ofast**
  - Disregard strict standards compliance. Enable **-ffast-math**, stack size is hardcoded to 32 768 bytes (borrowed from **gfortran**). **Possibily degrades the computation accuracy.**

The Hadamard product : Performance

Total Elapsed Time (cy)

Elapsed Time per element (cy/el)



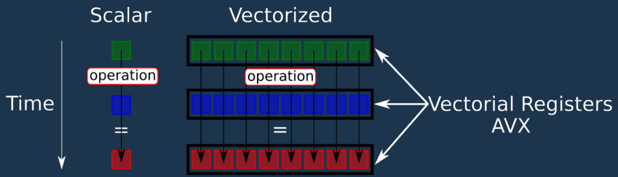


Speed up of **14** between -**O0** and -**O3** or -**Ofast**

What is vectorization ?

The idea is to compute several elements at the same time.

Architecture	Instruction Set	CPU	Nb **float** Computed at the same time
SSE4	2006	2007	4
AVX	2008	2011	8
AVX 512	2013	2016	16

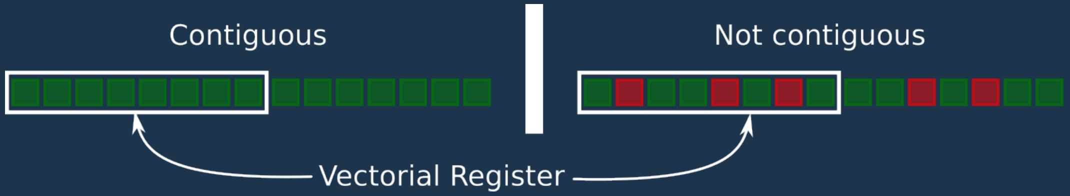


LINUX : **cat /proc/cpuinfo | grep avx** MAC : **sysctl -a | grep machdep.cpu | grep AVX**

What is vectorization ?

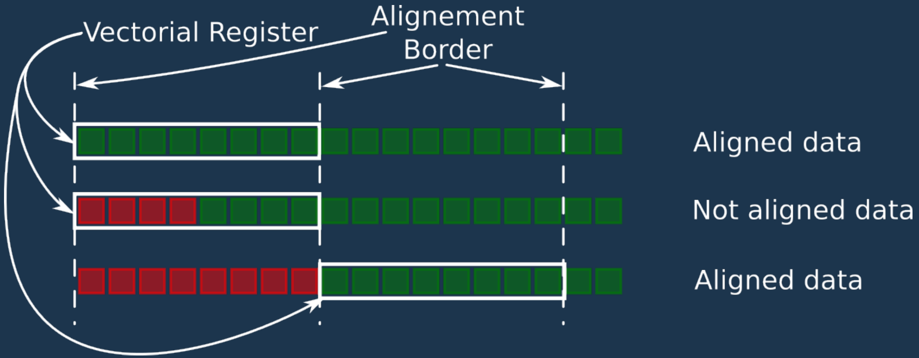
The CPU has to read several elements at the same time.

- Data contiguousness :
  - All the data to be used have to be adjacent with the others.
  - Always the case with pointers but be careful with your applications.



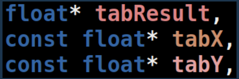
What is vectorization ?

- Data alignement :
  - All the data to be aligned on vectorial registers size.
  - Change **new** or **malloc** to **memalign** or **posix_memalign**



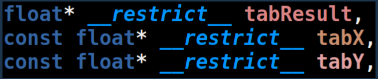
What do we have to do with the code ?

- The **__restrict__** keyword :
  - Specify to the compiler there is no overhead between pointers

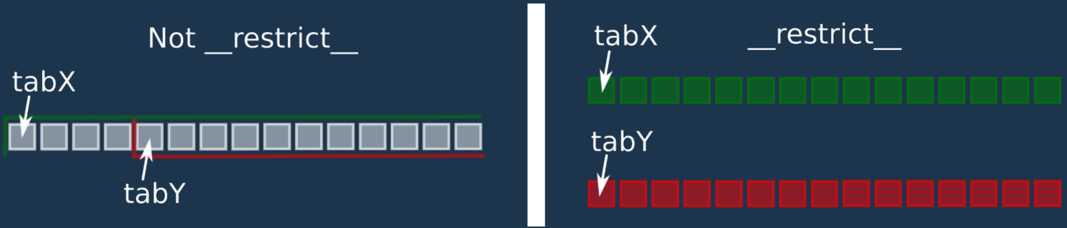


```
float* tabResult,
const float* tabX,
const float* tabY,
```

⟹

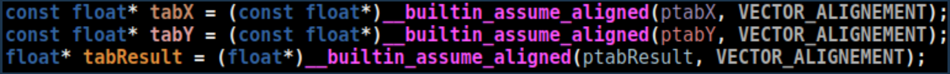


```
float* __restrict__ tabResult,
const float* __restrict__ tabX,
const float* __restrict__ tabY,
```



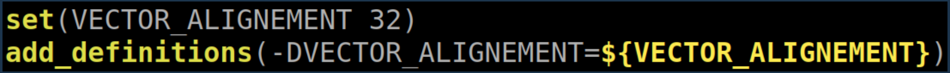
What do we have to do with the code ?

- The **__builtin_assume_aligned** function :
  - Specify to the compiler pointers are aligned
    - If this is not true, you will get a **Segmentation Fault**.
  - Here **VECTOR_ALIGNEMENT** = 32 (for **float** in **AVX** or **AVX2** extensions).



```
const float* tabX = (const float*)__builtin_assume_aligned(ptabX, VECTOR_ALIGNEMENT);
const float* tabY = (const float*)__builtin_assume_aligned(ptabY, VECTOR_ALIGNEMENT);
float* tabResult = (float*)__builtin_assume_aligned(ptabResult, VECTOR_ALIGNEMENT);
```

Definition in the file **ExampleMinimal/CMakeLists.txt** :



```
set(VECTOR_ALIGNEMENT 32)
add_definitions(-DVECTOR_ALIGNEMENT=${VECTOR_ALIGNEMENT})
```

Compilation Options

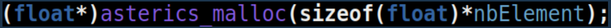
- The Compilation Options become :
  - **-O3 -ftree-vectorize -march=native -mtune=native -mavx2**

- **-ftree-vectorize**
  - Activate the vectorization
- **-march=native**
  - Target only the host CPU architecture for binary
- **-mtune=native**
  - Target only the host CPU architecture for optimization
- **-mavx2**
  - Vectorize with AVX2 extention

Modifications Summary

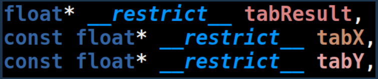
- Data alignement :
  - All the data to be aligned on vectorial registers size.
  - Change **new** or **malloc** to **memalign** or **posix_memalign**

  You can use **asterics_malloc** to have LINUX/MAC compatibility (in **evaluateHadamardProduct**):



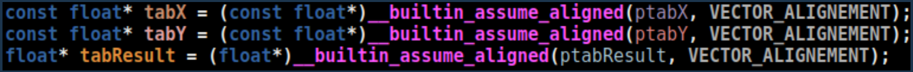
```
(float*)asterics_malloc(sizeof(float)*nbElement);
```

  The __**restrict**__ keyword (arguments of **hadamard_product** function):



```
float* __restrict__ tabResult,
const float* __restrict__ tabX,
const float* __restrict__ tabY,
```

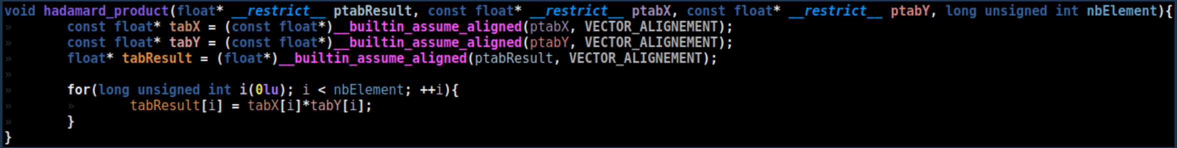
  The __**builtin_assume_aligned** function call (in **hadamard_product** function):



```
const float* tabX = (const float*)__builtin_assume_aligned(ptabX, VECTOR_ALIGNEMENT);
const float* tabY = (const float*)__builtin_assume_aligned(ptabY, VECTOR_ALIGNEMENT);
float* tabResult = (float*)__builtin_assume_aligned(ptabResult, VECTOR_ALIGNEMENT);
```

  - The Compilation Options become :
    - **-O3 -ftree-vectorize -march=native -mtune=native -mavx2**

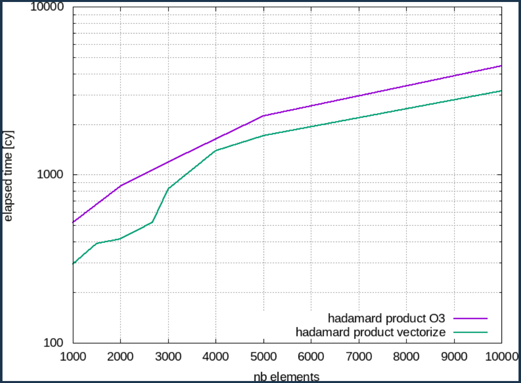
Code Correction



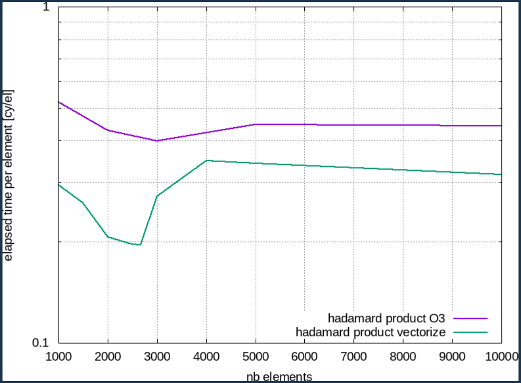
```
void hadamard_product(float* __restrict__ ptabResult, const float* __restrict__ ptabX, const float* __restrict__ ptabY, long unsigned int nbElement){
        const float* tabX = (const float*)__builtin_assume_aligned(ptabX, VECTOR_ALIGNEMENT);
        const float* tabY = (const float*)__builtin_assume_aligned(ptabY, VECTOR_ALIGNEMENT);
        float* tabResult = (float*)__builtin_assume_aligned(ptabResult, VECTOR_ALIGNEMENT);

        for(long unsigned int i(0lu); i < nbElement; ++i){
                tabResult[i] = tabX[i]*tabY[i];
        }
}
```

The Hadamard product : Vectorization

Total Elapsed Time (cy)



Elapsed Time per element (cy/el)

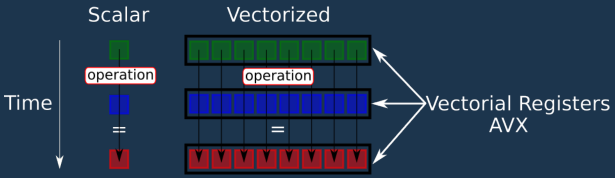


Vectorization by hand : Intrinsic functions

The idea is to force the compiler to do what you want and how you want it.

The Intel intrinsics documentation : `https://software.intel.com/en-us/node/523351`.

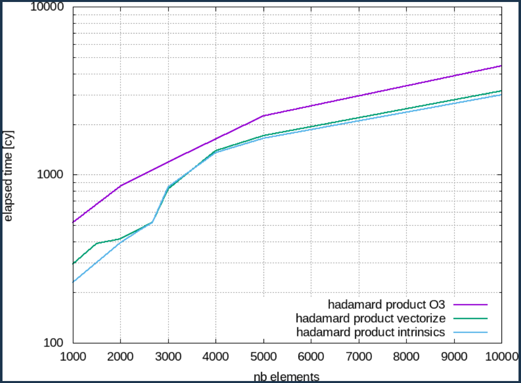
- Some changes (for AVX2):
  - Include : **immintrin.h**
  - **float** $\Longrightarrow$ **__m256** (= 8 **float**)
  - Data loading : **_mm256_load_ps**
  - Data Storage : **_mm256_store_ps**
  - Multiply : **_mm256_mul_ps**



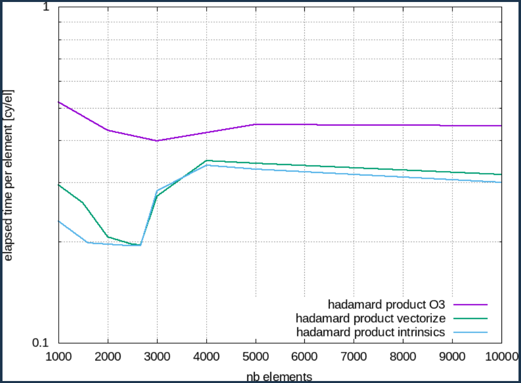
Only on aligned data of course.

The Hadamard product : Intrinsics

Total Elapsed Time (cy)

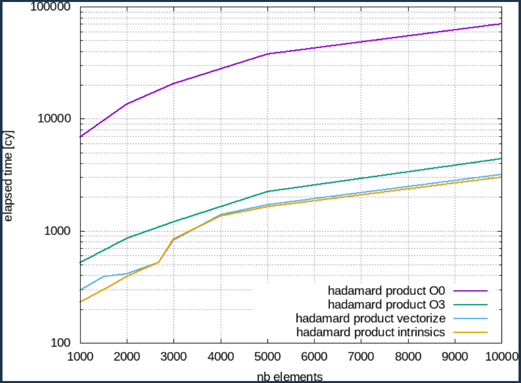


Elapsed Time per element (cy/el)

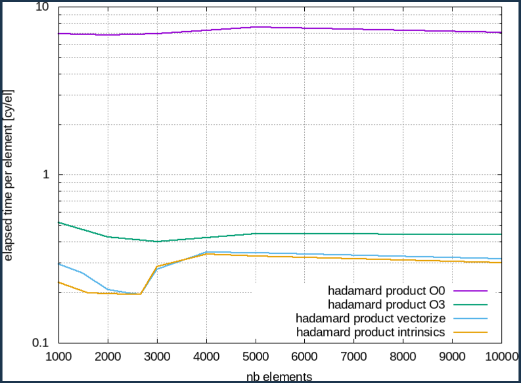


The Hadamard product : Summary

Total Elapsed Time (cy)



Elapsed Time per element (cy/el)



For 1000 elements : intrinsics version is 43.75 times faster than O0

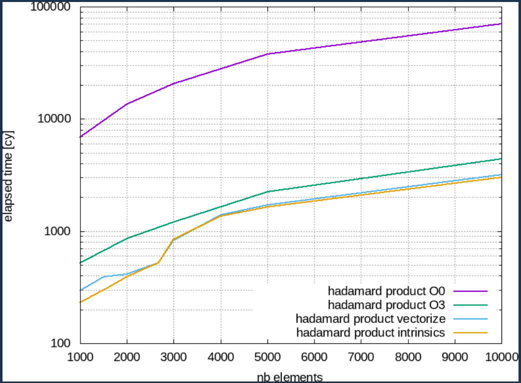
For 1000 elements : intrinsics version is 3.125 times faster than O3

Intrinsics version is a bit faster than vectorized version.

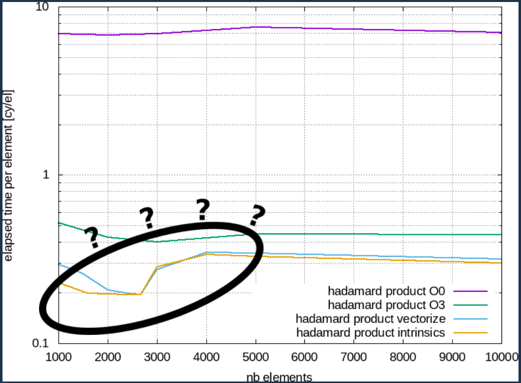
Compiler is very efficient

By the way... what is this step ?

Total Elapsed Time (cy)



Elapsed Time per element (cy/el)



For 1000 elements : intrinsics version is 43.75 times faster than O0

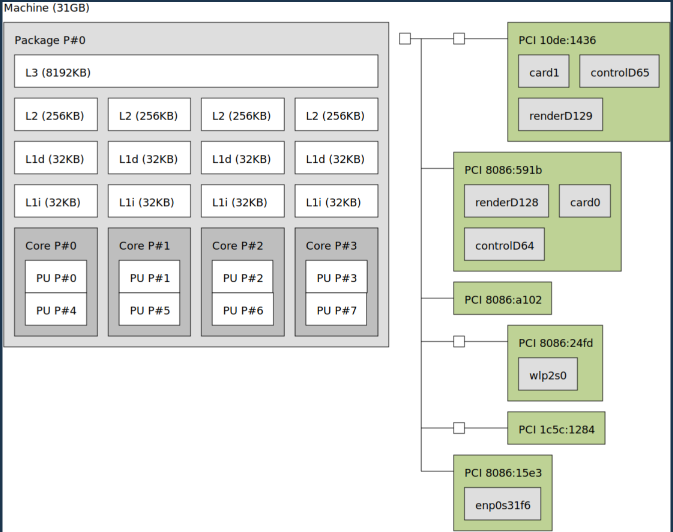
For 1000 elements : intrinsics version is 3.125 times faster than O3

Intrinsics version is a bit faster than vectorized version.

Compiler is very efficient

It is due to the Caches !

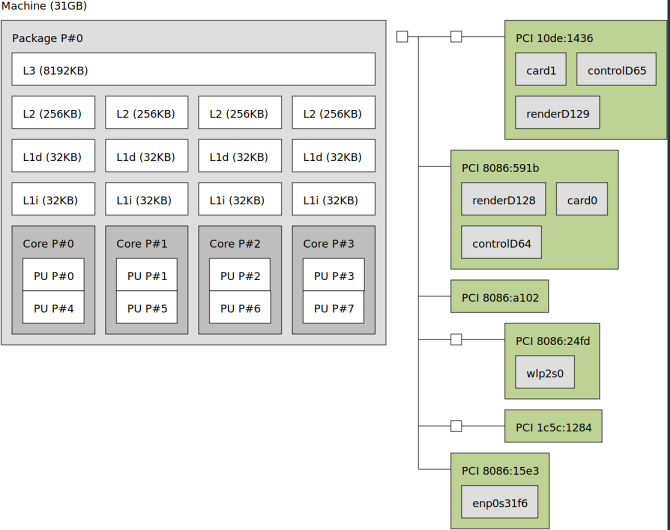
Let's call **hwloc-ls**



It is due to the Caches !

Let's call **hwloc-ls**

- Time to get a data :
  - **Cache-L1** : 1 cycle
  - **Cache-L2** : 6 cycles
  - **Cache-L3** : 10 cycles
  - **RAM** : 25 cycles

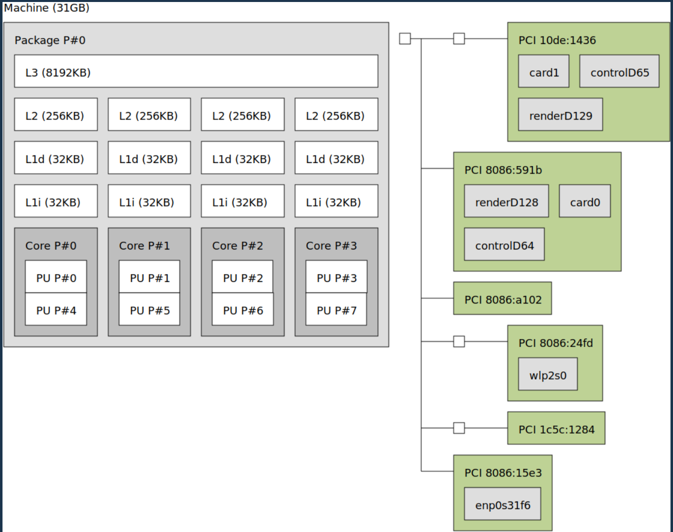


It is due to the Caches !

Let's call **hwloc-ls**

- Time to get a data :
  - **Cache-L1** : 1 cycle
  - **Cache-L2** : 6 cycles
  - **Cache-L3** : 10 cycles
  - **RAM** : 25 cycles

With no cache, 25 cycles to get a data implies a 2.0 *GHz* CPU computes at 80 *MHz* speed.



Machine (31GB)

Package P#0

L3 (8192KB)

L2 (256KB) | L2 (256KB) | L2 (256KB) | L2 (256KB)

L1d (32KB) | L1d (32KB) | L1d (32KB) | L1d (32KB)

L1i (32KB) | L1i (32KB) | L1i (32KB) | L1i (32KB)

Core P#0: PU P#0, PU P#4

Core P#1: PU P#1, PU P#5

Core P#2: PU P#2, PU P#6

Core P#3: PU P#3, PU P#7

PCI 10de:1436: card1, controlD65, renderD129

PCI 8086:591b: renderD128, card0, controlD64

PCI 8086:a102

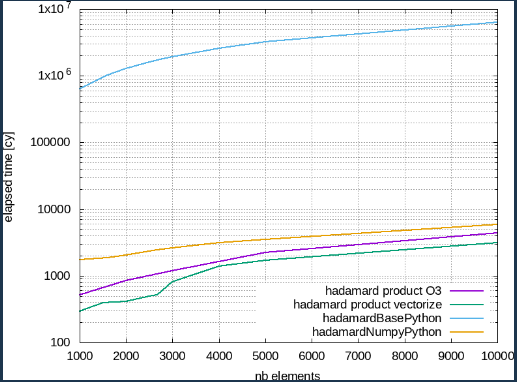
PCI 8086:24fd: wlp2s0

PCI 1c5c:1284

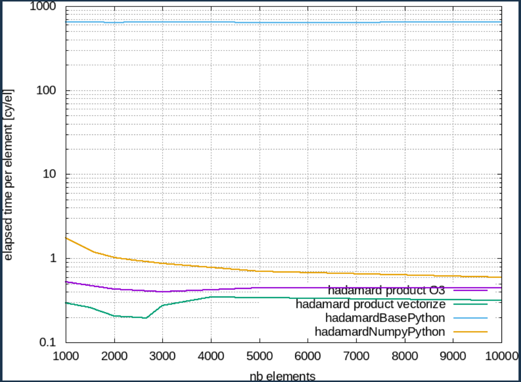
PCI 8086:15e3: enp0s31f6

The Hadamard product : Python

Total Elapsed Time (cy)



Elapsed Time per element (cy/el)



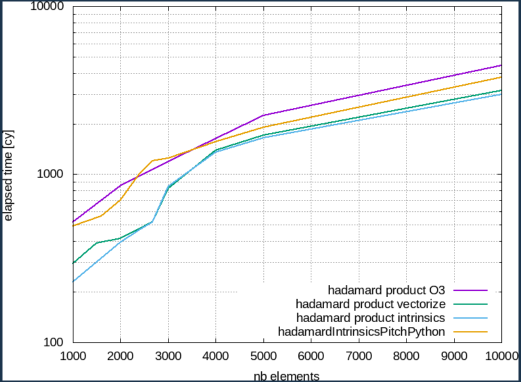
For 1000 elements : vectorized version is 3400 times faster than pure Python !!! (on numpy tables)

For 1000 elements : vectorized version is 8 times faster than numpy version

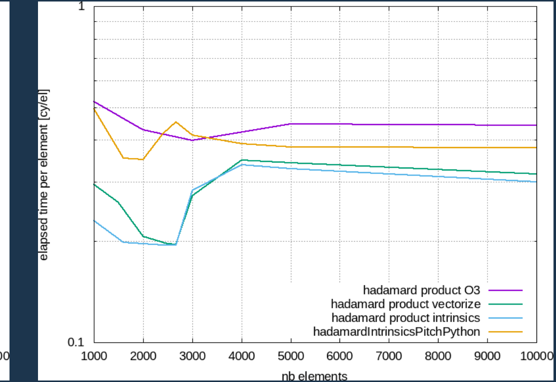
So, use numpy instead of pure Python (numpy uses the Intel MKL library)

The Python Hadamard product : Summary

Total Elapsed Time (cy)



Elapsed Time per element (cy/el)

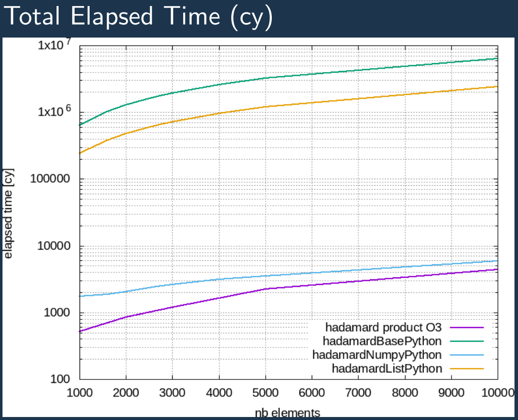


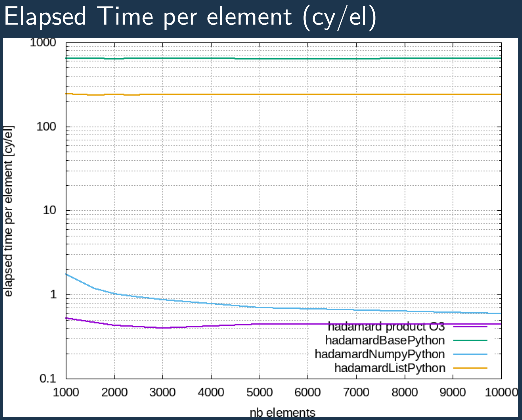
For 1000 elements : intrinsics C++ version is 4 times faster than our Python intrinsics

For 1000 elements : python intrinsics version is 1.2 times faster than O3

The Python function call cost a lot of time

The Python Hadamard product : list





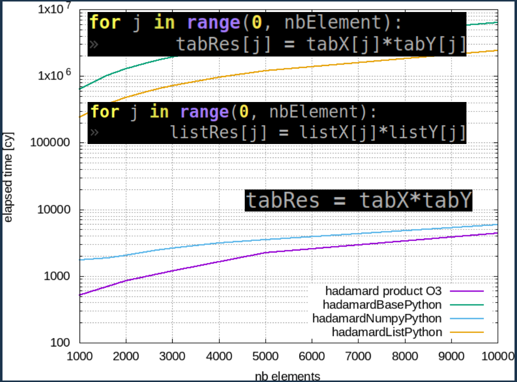
If you want to get elements one per one : lists are faster than **numpy** arrays

If you want to global computation : **numpy** arrays are faster than lists

If you want to be able to wrap you code : use **numpy** arrays

The Python Hadamard product : list

Total Elapsed Time (cy)

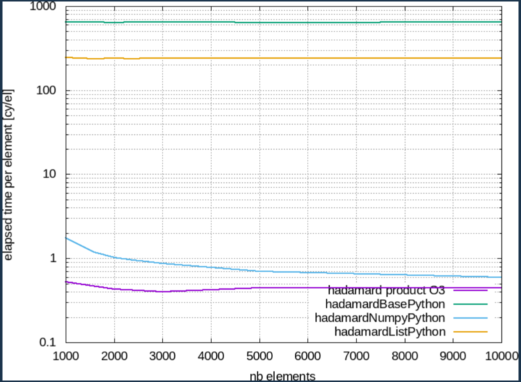


```
for j in range(0, nbElement):
»        tabRes[j] = tabX[j]*tabY[j]
```

```
for j in range(0, nbElement):
»        listRes[j] = listX[j]*listY[j]
```

```
tabRes = tabX*tabY
```

Elapsed Time per element (cy/el)



If you want to get elements one per one : lists are faster than **numpy** arrays

If you want to global computation : **numpy** arrays are faster than lists

If you want to be able to wrap you code : use **numpy** arrays