CentraleSupélec

SG6: High Performance Computing

Serial optimizations and vectorization

Stéphane Vialle

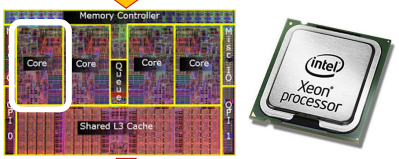
CentraleSupélec université PARIS-SACLAY — ÉCOLE DOCTORALE Sciences et technologies de l'information et de la communication (STIC) — RISEGrid — Grand Est ALSACE CHAMPAGNE-ARDENNE LORRAINE

Stephane.Vialle@centralesupelec.fr
http://www.metz.supelec.fr/~vialle

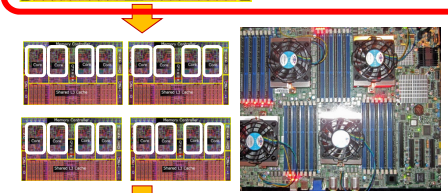
CentraleSupélec

HPC programming strategy

Numerical algorithm



Optimized code on one core:
- Optimized compilation
- Serial optimizations
- Vectorization



Parallelized code on one node
- Multithreading
- Minimal/relaxed synchronization
- Data-Computation localization
- *NUMA effect avoidance*

Distributed code on a cluster:
- Message passing accros processes
- Load balanced computations
- Minimal communications
- *Overlapped computations and comms*

2

CentraleSupélec

1 - Three important issues to get optimized code on one CPU core

1.1 – Compile with optimization options

1.2 – Implement data storage and data accesses taking advantage of cache memory hierarchy

1.3 – Implement sequences of instructions taking advantage of vector computing units

3

CentraleSupélec

Compile with optimization options

Any compiler propose options to produce executable code:

- Requiring less memory
- Running faster
- …

With Linux C/C++ compilers (ex: gcc/icc):

-O0: no optimization
-O1: 1st level of standard optimizations
-O2: 2nd level of standard optimizations
-O3: 3rd level of standard optimizations,
- usual level for HPC codes
- enables vectorization

And many specific optimization options:
Ex: *gcc –funroll-loops...* (see further)

→ ***gcc –O3 –funroll-loops pgm.c –o pgm***

With Windows compilers:

- similar and different options
- in the configuration menu of Visual Studio (for example)

4

CentraleSupélec

Compile with optimization options

Any compiler propose options to produce executable code:

- Requiring less memory
- Running faster
- …

But an optimized compilation:

- **Lasts longer**
  Ex: Intel compiler takes long time to attempt to vectorize the code but produces very helpful vectorization reports!
- **Requires a higher quality of the source codes**
  Some source codes can compile and successfully run when using -O0 or -O1, but fails when using –O3 !
  (especially when tinkering the data alignment in memory…)

CentraleSupélec

Compile with optimization options

Any compiler propose options to produce executable code:

- Requiring less memory
- Running faster
- …

HPC compilation rules:

1. Engage the standard optimization options of the compiler (always!) *(-O3 for HPC code)*
2. Improve your source code up to support the standard optimization options
3. Look at the specific optimization options of your compiler, and experiment these options

CentraleSupélec

1 - Three important issues to get optimized code on one CPU core

1.1 – Compile with optimization options

1.2 – Implement data storage and data accesses taking advantage of cache memory hierarchy

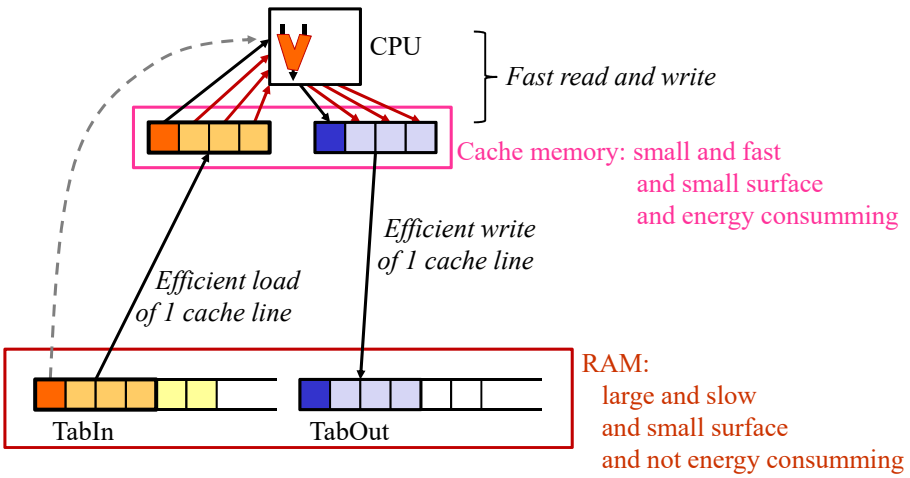
1.3 – Implement sequences of instructions taking advantage of vector computing units

7

CentraleSupélec

Take advantage of cache memory

Principles of the cache memory:



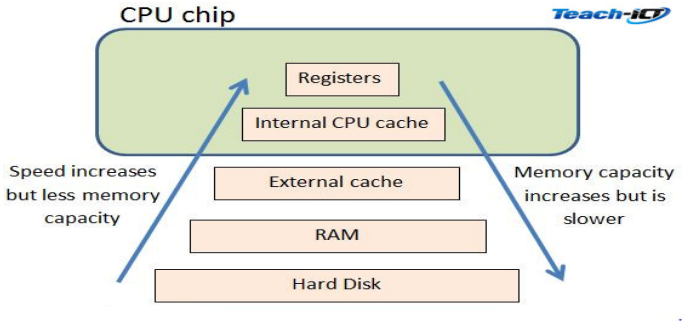
8

CentraleSupélec

3 important issues to get optimized code on 1 core

Take advantage of cache memory

Memory hierarchy:



- L1 and L2 cache memory are « internal CPU cache »
- L3 cache is external, or internal, or does not exist: depends on the CPU

9

CentraleSupélec

3 important issues to get optimized code on 1 core

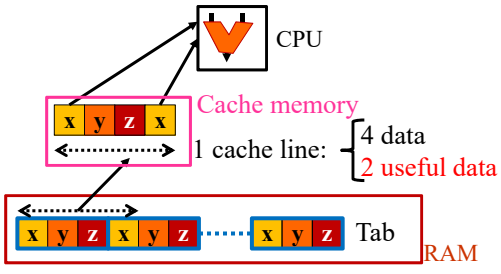
Take advantage of cache memory

Example of inefficient data storage and accesses:

```
typedef struct {
  double x, y, z;
} Point;

Point Tab[N];
double Sx = 0.0;
double MoyX;

...
for (i=0; i<N; i++) {
  Sx += Tab[i].x;
}
MoyX = Sx/N;
...
```



Array of structures (objects) →
2 'x coordinates' are separated by 2 others values, unused in this computations, but tacking space into cache memory

10

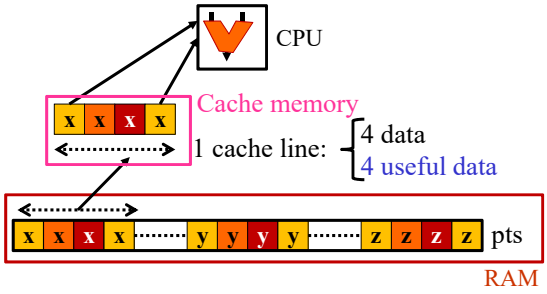
Take advantage of cache memory

Example of efficient data storage and accesses:

```
typedef struct {
  double TabX[N];
  double TabY[N];
  double TabZ[N];
} Points;

Points pts;
double Sx = 0.0;
double MoyX;

...
for (i=0; i<N; i++) {
  Sx += pts.TabX[i];
}
MoyX = Sx/N;
...
```



Structure of arrays →

'x coordinates' are contiguously stored in RAM, and contiguously copied/stored in cache: any valid value in cache should be useful

1 - Three important issues to get optimized code on one CPU core

1.1 – Compile with optimization options

1.2 – Implement data storage and data accesses taking advantage of cache memory hierarchy

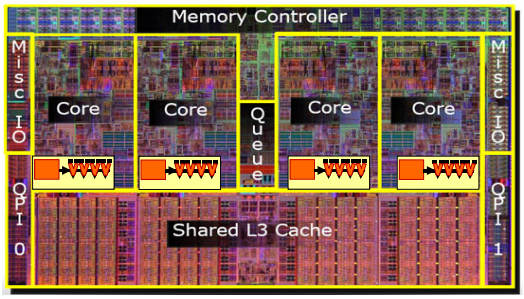
1.3 – Implement sequences of instructions taking advantage of vector computing units

Take advantage of vector computing units

Vector computing:

Each CPU core is a small vector machine:

1 instruction can be applied on vectors of input data and can produce a vector of output data

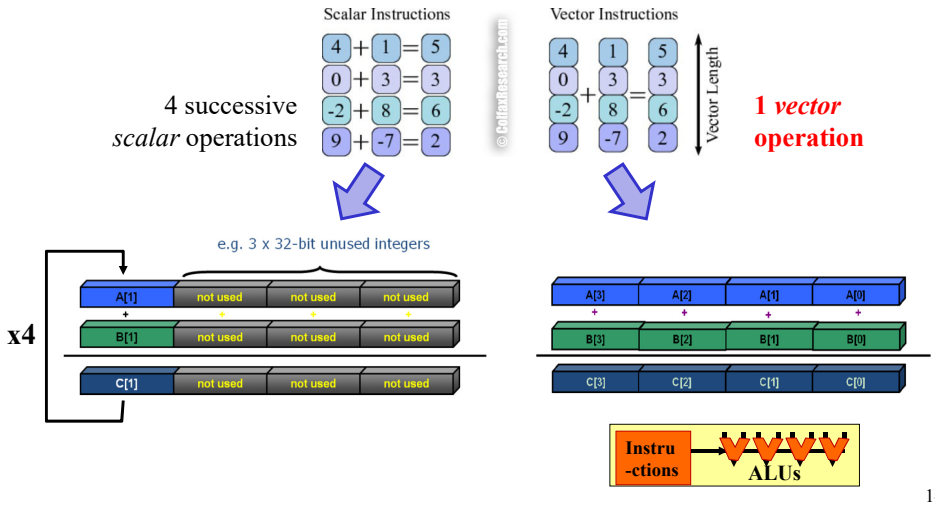


→ One instruction can be executed in parallel by different ALU, on different data

AVX units

Take advantage of vector computing units

Scalar *vs* vector operations:



Take advantage of vector computing units

Requirements to enable vector operations:

- Input and output data must be stored in arrays at contiguous indexes
  (compliant with an efficient data storage based on cache memory usage)
- Internal loop must have:
  - Identical operations
    - no *if-then-else* in the loop body
    - but *if-then* is possible, as an ALU can do the operation or nothing
  - Idependent operations
    - can be executed in any order
      res[i] = input[i] + res[i-1] : impossible to vectorize
    - write in different variables → *reductions* must be adapted
      res += input[i]*input[i] : limit/stop the vectorization
- Arrays must not overlap (« *no aliasing* ») → operations could not be parallelized

Take advantage of vector computing units

How to write vector code (instead of scalar code) ?

- Implement explicit vector code using « *intrinsic* » low level library…

- Add *directives* to *guide* the compiler to generate vector code:

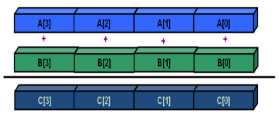
```
#pragma vector
#pragma ivdep
for (j = 0; j < n; j++)
  DynamicTab[j] = …
```

- ***auto-vectorization***:

  Write clean standard code with respect to all vectorization constraints and ask to the compiler to vectorize the code (compilation option)

```
for (j = 0; j < 1000; j++)
  StaticTab[j] = …

gcc -O3 pgm.c -o pgm
```



CentraleSupélec

2 – Optimization and vectorization of a Dense Matrix Product

2.1 – Stengths and weaknesses of the naive version

2.2 – 1st solution: evolution of the data storage

2.3 – 2nd solution: evolution of the loop order

17

CentraleSupélec

Strenghts and weaknesses of the naive version

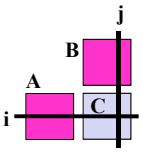
Strenghts :

The naive version implements directly the mathematical expression

Code is:

- easy to implement
- easy to maintain

$$C_{i,j} = \sum_{k=0}^{k<n} A_{i,k} \times B_{k,j}$$




```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++)
      C[i][j] += A[i][k]*B[k][j];
```

Weaknesses :

1. Internal loop does not acces B[k][j] elements in contiguous order
   → makes a not-optimal usage of the cache memory
2. Internal loop writes in the same variable C[i][j] at each iteration
   → limits/stops the vectorization *(iterations not independent)*

18

CentraleSupélec

2 – Optimization and vectorization of a Dense Matrix Product

2.1 – Stengths and weaknesses of the naive version

2.2 – 1st solution: evolution of the data storage

2.3 – 2nd solution: evolution of the loop order

19

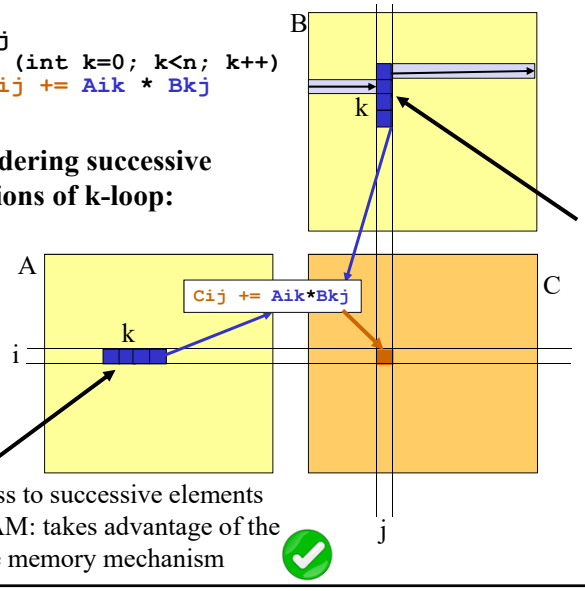
CentraleSupélec

1st solution: evolution of the data storage

Identification of « cache misses »

```
for i
 for j
  for (int k=0; k<n; k++)
    Cij += Aik * Bkj
```

Considering successive iterations of k-loop:



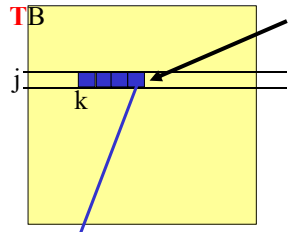
Access non contiguous elements in RAM:
- many « cache misses »
- poor usage of the cache memory

Access to successive elements in RAM: takes advantage of the cache memory mechanism

20

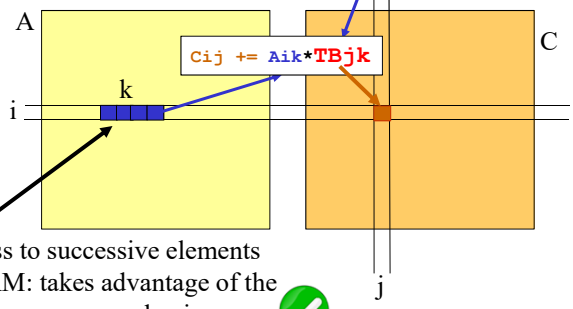
Avoiding « cache misses »

```
for i
 for j
  for (int k=0; k<n; k++)
    Cij += Aik * TBjk
```



Access to successive elements in RAM: takes advantage of the cache memory mechanism

Considering successive iterations of k-loop:



Access to successive elements in RAM: takes advantage of the cache memory mechanism

Avoiding « cache misses »

Source code with new data storage

```
// Dense matrix product: C = A×B
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++) {
      C[i][j] += A[i][k]*TB[j][k];
    }
}
```

Compilation: gcc –O3 pgm.c –o pgm

This source code version is closer of the architecture of the processor

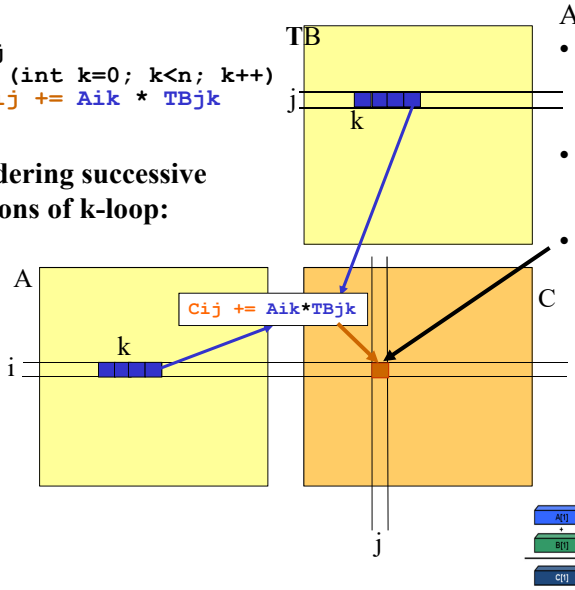
- faster
- more complex to understand and to maintain

1st solution: evolution of the data storage

Identification of a vectorization lock

```
for i
 for j
  for (int k=0; k<n; k++)
    Cij += Aik * TBjk
```

Considering successive iterations of k-loop:



At each iteration:

- Identical computation *(no if-then-else, no divergence)*
- Access to successive array indexes
- Write/Accumulation in the same variable (Cij):
  - iterations are not independent
  - vectorization is limited

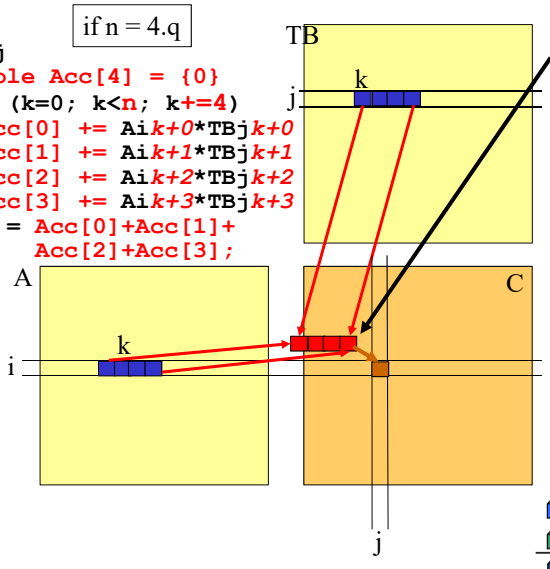
23

1st solution: evolution of the data storage

Unlocking the vectorization

```
for i
 for j
  double Acc[4] = {0}
  for (k=0; k<n; k+=4)
    Acc[0] += Aik+0*TBjk+0
    Acc[1] += Aik+1*TBjk+1
    Acc[2] += Aik+2*TBjk+2
    Acc[3] += Aik+3*TBjk+3
  Cij = Acc[0]+Acc[1]+
        Acc[2]+Acc[3];
```

if n = 4.q

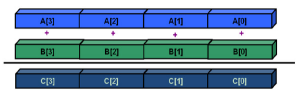


- Loop unrolling
- Accumulation in a vector of buffers

Each k-loop iteration:

- includes 4 identical & independent instructions
- reading and writing successive array indexes

→ **Compiler can vectorize each k-loop iteration**



24

CentraleSupélec

1st solution: evolution of the data storage

Unlocking the vectorization

Source code 1: with new data storage & loop unrolling

```
// Dense matrix product: C = A×B
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++) {
    double accu[8] = {0.0};
    for (int k = 0; k < (n/8)*8; k += 8) {
      accu[0] += A[i][k+0]*TB[j][k+0];
      accu[1] += A[i][k+1]*TB[j][k+1];
      .....
      accu[7] += A[i][k+7]*TB[j][k+7];
    }
    for (int k = (n/8)*8; k < n; k++)
      accu[0] += A[i][k]*TB[j][k];
    C[i][j] = accu[0] + … + accu[7];
  }
```

Loop unrolling with 8-factor *(in case of long AVX units)*

Generic solution *runs for any value of n*

(int value / 8 : integer division)

Compilation: gcc –O3 –f unroll-loops pgm.c –o pgm

Specific compilation option: to improve loop unrolling

25

CentraleSupélec

1st solution: evolution of the data storage

Unlocking the vectorization

Source code 2: with new data storage & loop unrolling

```
// Dense matrix product: C = A×B
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++) {
    for (int k = 0; k < (n/8)*8; k += 8) {
      C[i][j] += A[i][k+0]*TB[j][k+0] +
                 A[i][k+1]*TB[j][k+1] +
                 .....
                 A[i][k+7]*TB[j][k+7];
    }
    for (int k = (n/8)*8; k < n; k++)
      C[i][j] += A[i][k]*TB[j][k];
}
```

Loop unrolling with 8-factor *(in case of long AVX units)*

Generic solution *runs for any value of n*

Compilation: gcc –O3 –f unroll-loops pgm.c –o pgm

Implement loop unrolling & *a big instruction*, grouping many identical operations on successive array indexes

→ Better or worst than previous solution: depends on the compiler

26

CentraleSupélec

2 – Optimization and vectorization of a Dense Matrix Product

2.1 – Stengths and weaknesses of the naive version

2.2 – 1st solution: evolution of the data storage

2.3 – 2nd solution: evolution of the loop order

27

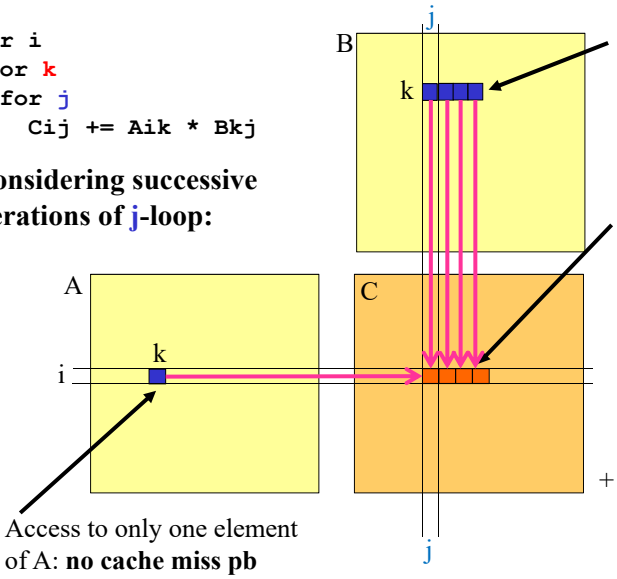
CentraleSupélec

2nd solution: evolution of the loop order

Inversion of *j* and *k* loops

```
for i
 for k
  for j
    Cij += Aik * Bkj
```

Considering successive iterations of j-loop:



Access to successive array indexes: **right use of the cache memory**

Access to successive array indexes: **right use of the cache memory**

+ Independent & identical operations
+ No write conflict

→ **auto-vectorization**

+ **no explicit loop-unrolling** (use only –funroll-loops option of the compiler)

Access to only one element of A: **no cache miss pb**

28

CentraleSupélec

2nd solution: evolution of the loop order

Inversion of *j* and *k* loops

Source code with new loop order: « ikj »

```
// Dense matrix product: C = A×B
for (int i = 0; i < n; i++)
  for (int k = 0; k < n; k++)
    for (int j = 0; j < n; j++)
      C[i][j] += A[i][k]*B[k][j];
```

Compilation: gcc –O3 –f unroll-loops pgm.c –o pgm

Usually faster when compiling with –f unroll-loops option

- Right use of the cache memory
- Suppression of the write conflict during vectorization of the inner loop

→ Decreases the number of cache misses

→ Enable the auto-vectorization

Elegant and efficient ! … but not always so simple !

29

CentraleSupélec

2nd solution: evolution of the loop order

Investigating all possible inner loop

	Cij	+=	Aik	*	Bkj	Inner loop
Right use of cache	NO! cache misses		NO! cache misses		OK access one elt	i
Vectorisation enabled	NO not-contiguous	→ NO	NO! not-contiguous	→ NO ←	OK access one elt	
Right use of cache	OK contiguous		OK access one elt		OK contiguous	j
Vectorisation enabled	OK contiguous no W conflict	→ OK	OK access one elt	→ OK ←	OK contiguous	
Right use of cache	OK access one elt		OK contiguous		NO! cache misses	k
Vectorisation enabled	NO! W conflit	→ NO	OK contiguous	→ NO ←	NO! not-contiguous	

→ **inner loop = j-loop** : the only right solution

(without changing the data storage)

30

CentraleSupélec

3 – Next step: « cache blocking / tiling »

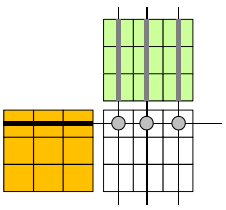
31

CentraleSupélec

« Cache blocking / Tiling »

Standard matrix product:

- Read one row of A and one column of B (or a line of B^t) and compute one element of C
- Compute the « next » element of C: next column, same line
- Optimization avoids many cache misses



NxN matrixes

But each input value is still read many times from the RAM, while computing all C elements:

- Each A row is read N times
- Each B column is read N times

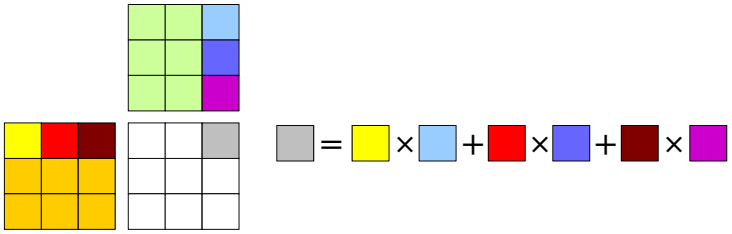
→ poor usage of the cache memory !

→ **Compute blocks / tiles of C matrix to decrease the global number of RAM accesses**

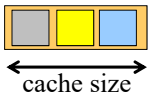
32

CentraleSupélec

« Cache blocking / Tiling »



Set Tile size to have: **3×Tile size ≤ Cache size**



At any time:
- 3 tiles in cache memory
- computation using only the 3 tiles
- maximum re-use of each tile while it is in cache (optimize the schedule of the C tile computation)

« Tiling » is very classical in HPC algorithmic and programming

33

CentraleSupélec

Experiments

34

Experiments on an Intel Xeon Haswell

Experiments (2018) :

Dense matrix product: 4096x4096, double precision

Processor: Intel Xeon *Haswell* E5-2637 v3 - 2014

(4 physical cores – 2 threads/core)

Seq. Naive -O0	Seq. Naive -O3	-O3 + Optimized code + Vectorization	BLAS monothread (OpenBLAS)
0.12 Gflops	0.35 Gflops	3.10 Gflops	46.3 Gflops
x1.0	x2.9	x25.8	x385.8
		2 threads/core	1 thread/core

→ Use optimized HPC libraries when available

→ Optimize your source code when HPC library does not exist

35

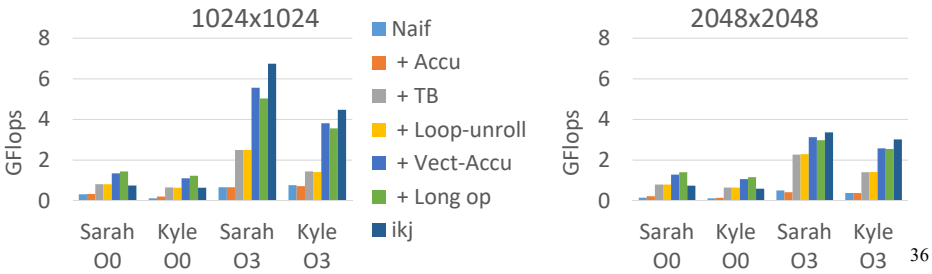
Experiments on 2 Intel Xeon architectures

Expérimentation (2019) : Kernel développé en TP (*K0*)

- Sarah : quad-cores à 3.5 GHz
  (E5-2637 v3 « haswell », 15MB cache, 2014 – gcc 5.4.0)
- Kyle : octo-cores à 2.1 GHz
  (Silver 4110 CPU « skylake », 11MB cache, 2017 – gcc 7.3.0)

Les petites matrices tiennent mieux en cache → perfs + élevées

La solution « ikj » apparait très bonne.



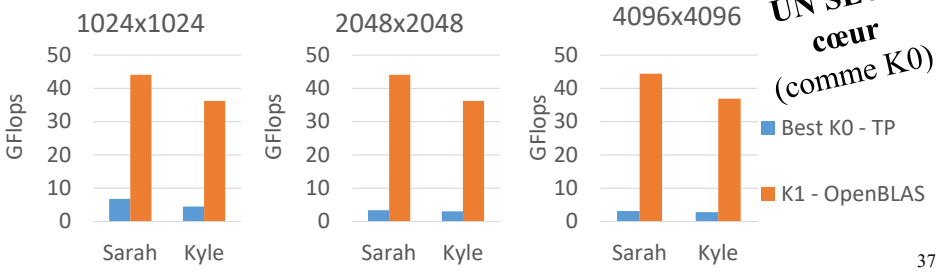
36

Experiments on 2 Intel Xeon architectures

Expérimentation (2019) : Kernel OpenBLAS (*K1*)

- Sarah : quad-cores à 3.5 GHz
  (E5-2637 v3 « haswell », 15MB cache, 2014 – gcc 5.4.0)
- Kyle : octo-cores à 2.1 GHz
  (Silver 4110 CPU « skylake », 11MB cache, 2017 – gcc 7.3.0)

Les BLAS cumulent des développement très conséquents
Toujours les utiliser pour des calculs d'algèbre linéaire



BLAS sur UN SEUL cœur (comme K0)

37

CentraleSupélec

Serial optimizations and vectorization

Questions ?

38