Guider le compilateur pour optimiser la performance de vos applications.

Lyon – 25 Feb 2014

Gunter Roeth
Applications Engineer
Applications & Performance Team
Extreme Computing Business Unit

PLAN

1. COMPILATEURS POUR LE CALCUL HAUTE PERFORMANCE
2. OPTIONS DE COMPILATION GNU/INTEL
3. AVX INSTRUCTION SET (VECTORISATION)
4. BANDE PASSANTE MEMOIRE
5. MODÈLE ROOFLINE
6. FIRST TOUCH POLICY
7. ALIGNEMENT
8. EXEMPLES: BOUCLE TYPIQUE ET DIFFÉRENCES FINIS
9. CONCLUSIONS

FORTRAN AND C/C++ COMPILERS

Freeware

- GNU Fortran and C/C++ compilers : gfortran, gcc, g++ are popular
  gnu.org
- Oracle Solaris Studio
  oracle.com/technetwork/server-storage/solarisstudio/downloads/index.html
- Open64
  - Not too good, and not very clear about license free

  open64.net

Proprietary

- Intel
  intel.com
- NVIDIA bought Portland Group
  pgroup.com
- Absoft, Cray, IBM, NAG, PathScale

USER PROGRAM DEVELOPMENT

From equations to simulations

- Choose a high level programming language (C,C++, Fortran) avoid scripting.
- Design the program flow
  - Try to keep the program modular.
  - Write validation routines, not only for toy cases :
    - Seismic may check amplitudes/frequencies …
    - Avoid absolute reference results obtained through simulation
  - Write performance (timer) measurements around the kernels.
- Use several compilers

GNU COMPILERS OPTIMIZATIONS

Very popular compiler and tools
GNU GCC 4.7 March 2012 latest version : GCC 4.8

- Needs the latest mtune/march/with-cpu options available designed for AVX and Intel's newest CPUs.
  - –mtune=corei7-avx –mavx
  - Build with --with-mfpmath=avx to use AVX floating-point arithmetic
- Most optimizations are only enabled if an -O level is set on the command line.
  - optimization with very conservative defaults
  - O and –O2 will not increase the code size, and work everywhere
  - gfortran/gcc auto-vectorization needs –O3
  - Profile feedback available

GNU COMPILERS AGGRESSIVE OPTIONS

-ffast-math

- Allows mathematical simplifications for computations.
- May be needed for vectorization.

-ftree-loop-distribution, -ftree-vectorize

- Perform loop distribution : one loop is distributed into several smaller loops.
- allow further loop optimizations, like parallelization or vectorization, to take place.

-ftree-vectorizer-verbose=2

- vectorization report

C may need the restrict qualifier for the pointers and intrinsic __builtin_assume_aligned

```
void test4(double * restrict a, double * restrict b)
{
int i;
double *x = __builtin_assume_aligned(a, 16);
double *y = __builtin_assume_aligned(b, 16);
for (i = 0; i < SIZE; i++) { x[i] += y[i]; }
}
```

GNU COMPILERS LOOP OPTIMIZATIONS

To use this code transformation, GCC has to be configured with --with-ppl and --with-cloog to enable the Graphite loop transformation infrastructure.

-floop-interchange

Perform loop interchange transformations on loops.

-floop-strip-mine (use with loop-block-tile-size parameter for striplength)

Loop blocking of a single loop. Strip mining splits a loop into two nested loops. The outer loop has strides equal to the strip size and the inner loop has strides of the original loop within a strip.

-floop-block

Perform loop blocking transformations on loops. Blocking strip mines each loop in the loop nest such that the memory accesses of the element loops fit inside caches.

```
DO II = 1, N, 51
 DO JJ = 1, M, 51
  DO I = II, min (II + 50, N)
   DO J = JJ, min (JJ + 50, M)
      A(J, I) = B(I) + C(J)
   ENDDO
  ENDDO
 ENDDO
ENDDO
```

```
DO II = 1, N, 51
    DO I = II, min (II + 50, N)
          A(I) = A(I) + C
    ENDDO
ENDDO
```

INTEL OPTIMIZATION FLAGS

Common flags for ifort, icc, icpc

- **-O3** to do
  - loop transformations first
  - attention includes FP model fast=1 with „value-unsafe“ optimizations
  - -fp-model fast=2 -no-prec-div -no-prec-sqrt
- **-ipo**
  - inlining, loop counts, alignment information
- **-xavx**
  - to use the AVX instructions on Intel CPUs (better then –mavx)

ifort –O3 –ipo –xavx

C specific flags

- **-fargument-noalias**
  - assume function arguments not aliased
- **-fansi-alias**
  - assume different data types not aliased
- **-fno-alias**
  - assume pointers not aliased (dangerous!)
- **–restrict**
  - or “restrict” keyword, –std=c99

INTEL SPECIFIC COMPILER PRAGMAS

#pragma	Description
vector/novector always (un)aligned (non)temporal (no)vecremainder	Instructs the compiler to vectorize override the cost model, and vectorize non-unit strides or very unaligned memory accesses; use of streaming stores vectorize remainding loop
ivdep	The compiler is instructed to ignore not proven dependencies. However still performs a dependency analysis, and will not vectorize if it finds a proven dependency that would affect results.
simd vectorlength(*n1*[, *n2*]...) vectorlengthfor(*data type*) (first/last)private(*var1*[, *var2*]...) reduction(*oper:var1*[,*var2*]…) linear(*var1:step1*[,*var2:step2*]...) (no)vecremainder	Compiler skips dependency analysis that might cause incorrect results after vectorization. Compilation fails if not vectorized implies the loop unroll factor from OMP parallel do syntax For every iteration var is incremented by step. Every iteration of the vector loop var is incremented by VL*step specify different strides for different variables.

REQUIREMENTS FOR VECTORIZATION

- Must be a unit strided inner loop and may contain :
  - mathematical operators (sqrt, sin, exp,…)
  - if statements
  - reduction loops
  - Fortran vectorizes on the first index : array(i,j,k)
  - C,C++ on the last (unit stride) : array[i] [j] [k]

- Avoid:
  - Function/subroutine calls (unless inlined)
  - Non-mathematical operators
  - Data-dependent loop exit conditions
  - Iteration count must be known at entry to loop
  - Loop carried data dependencies
  - Non-contiguous data (indirect addressing; non-unit stride)
  - Inefficient (compiler heuristics)
  - Align your data where possible
    - to 32 byte boundaries (for AVX instructions)
    - to 16 bytes, or at least to “natural” alignment

```
#pragma simd
for(int ray=0; ray < N; ray++) {
  float Color = 0.0f, Opacity = 0.0f;
  int len = 0;
  int upper = raylen[ray];
  while (len < upper) {
    int voxel = ray + len;
    len++;
    if(visible[voxel] == 0) continue;
    float O = opacity[voxel];
    if(O == 0.0) continue;
    float Shading = O + 1.0;
    Color += Shading * (1.0f -
Opacity);
    Opacity += O * (1.0f - Opacity);
    if(Opacity > THRESH) break;
  }
  color_out[ray] = Color;
}
```

VECTORIZATION DIRECTIVES

```
void my_combine(int * ioff, int nx, double * a, double * b, double * c)
{
        int i;
//#pragma ivdep
#pragma simd
        for(i=0; i<nx; i++)
                a[i]=b[i]+c[i+*ioff];
}
```

icc -c combine.c -vec-report=3

```
combine.c(4): (col. 2) remark: loop was not vectorized: existence of vector dependence.
combine.c(5): (col. 3) remark: vector dependence: assumed ANTI dependence between c line 5 and a line 5.
combine.c(5): (col. 3) remark: vector dependence: assumed ANTI dependence between ioff line 5 and a line 5.
combine.c(5): (col. 3) remark: vector dependence: assumed FLOW dependence between a line 5 and ioff line 5.
combine.c(5): (col. 3) remark: vector dependence: assumed FLOW dependence between a line 5 and ioff line 5.
combine.c(5): (col. 3) remark: vector dependence: assumed ANTI dependence between ioff line 5 and a line 5.
combine.c(5): (col. 3) remark: vector dependence: assumed FLOW dependence between a line 5 and b line 5.
combine.c(5): (col. 3) remark: vector dependence: assumed ANTI dependence between b line 5 and a line 5.
```

icc -c combine.c -vec-report=3 with #ivdep

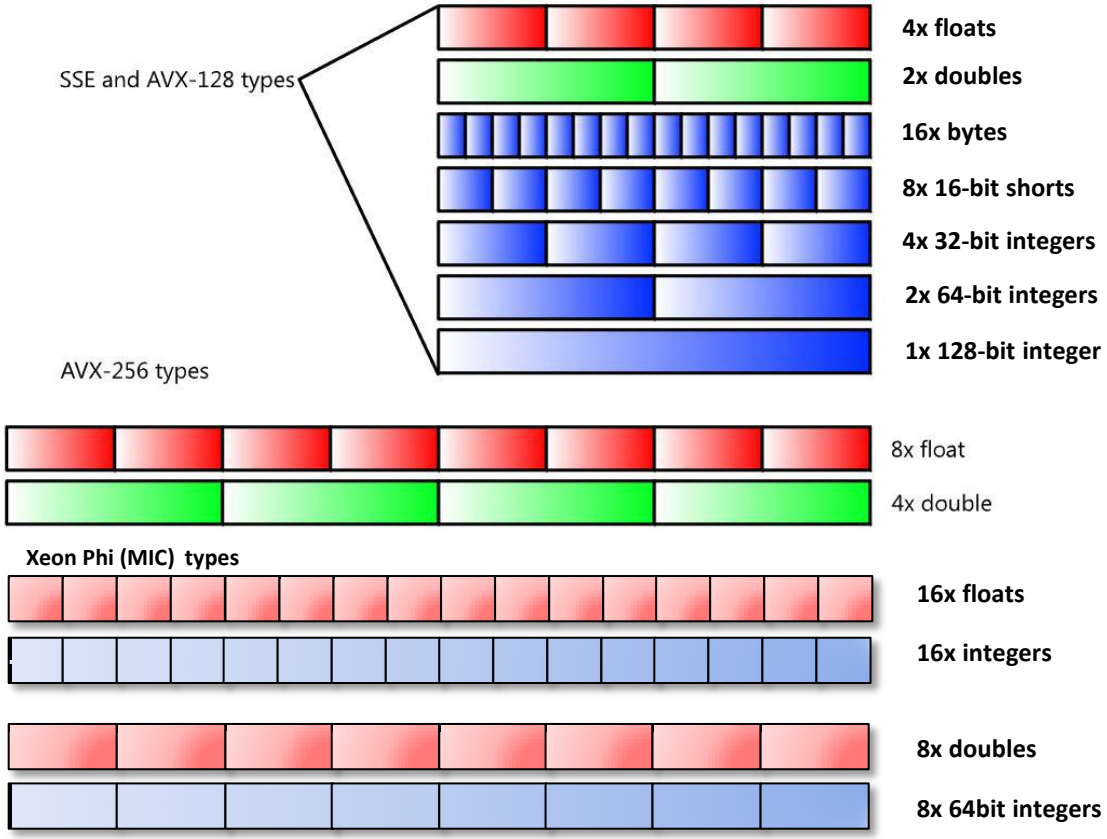
```
combine.c(5): (col. 2) remark: loop was not vectorized: vectorization possible but seems inefficient.
```

icc -c combine.c -vec-report=6

```
combine.c(5): (col. 2) remark: vectorization support: unroll factor set to 4.
combine.c(5): (col. 2) remark: SIMD LOOP WAS VECTORIZED.
```

gcc -c combine.c -O3 -ftree-vectorize -ftree-vectorizer-verbose=2 -march=corei7-avx -mtune=corei7-avx

INTEL SSE/AVX DATA TYPES



SSE

SSE-2

MMX™

Vector size: 64bit

Data types: 8, 16 and 32 bit ints

VL: 2,4,8

Intel® SSE

Vector size: 128bit

Data types: 8,16,32,64 bit ints
32 and 64bit floats

VL: 2,4,8,16

Intel® AVX

Vector size: 256bit

Data types:
32 and 64 bit floats

VL: 4, 8, 16

Intel® MIC

Vector size: 512bit

Data types: 32 and 64 bit ints
32 and 64 bit floats
(some support for 16 bits floats)

VL: 8,16

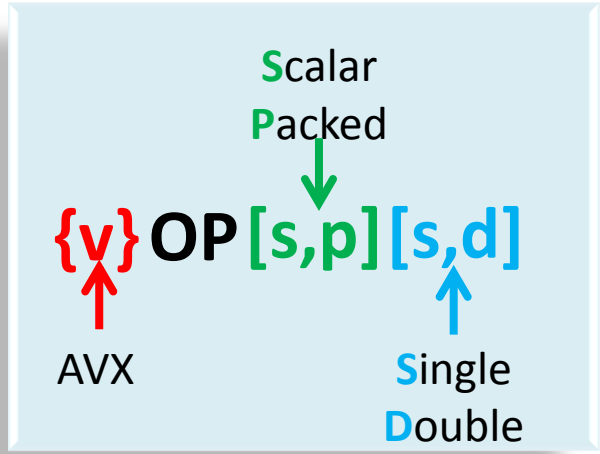
RECOGNIZE VECTOR CODE …

Compiler Report

Edit assembler

- Use : « -S –masm=intel »
- For an existing binary or object file : « objdump -M intel –D mon_binaire.x»

```
#pragma omp parallel for
#pragma unroll(4)
                for(i=0;i<N;i+=1) {
                        C[i]=alpha*A[i]+B[i];
                }
```



NO vectorisation

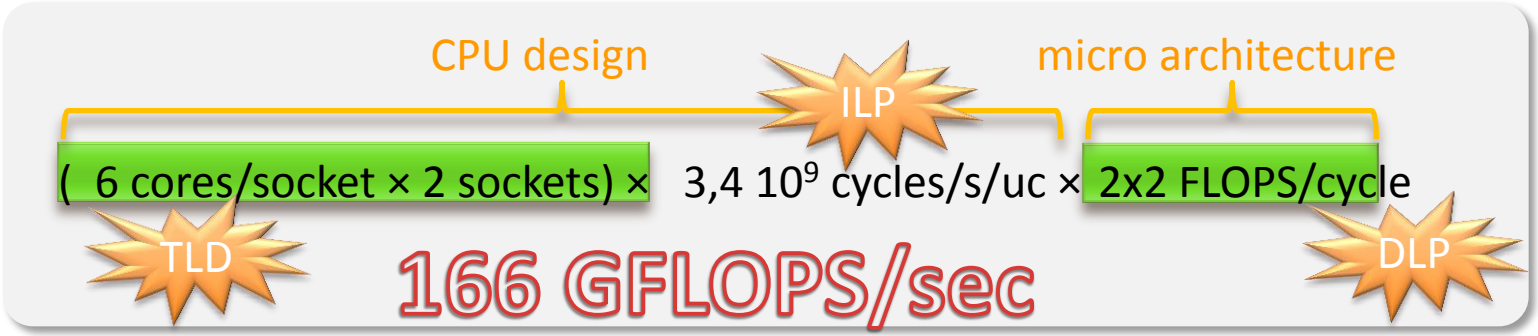
```
movsd      xmm1, QWORD PTR [8+r9+rdi]
mulsd      xmm0, xmm4
addsd      xmm0, QWORD PTR [r9+rcx]
movsd      QWORD PTR [r9+r8], xmm0
```

WITH vectorisation

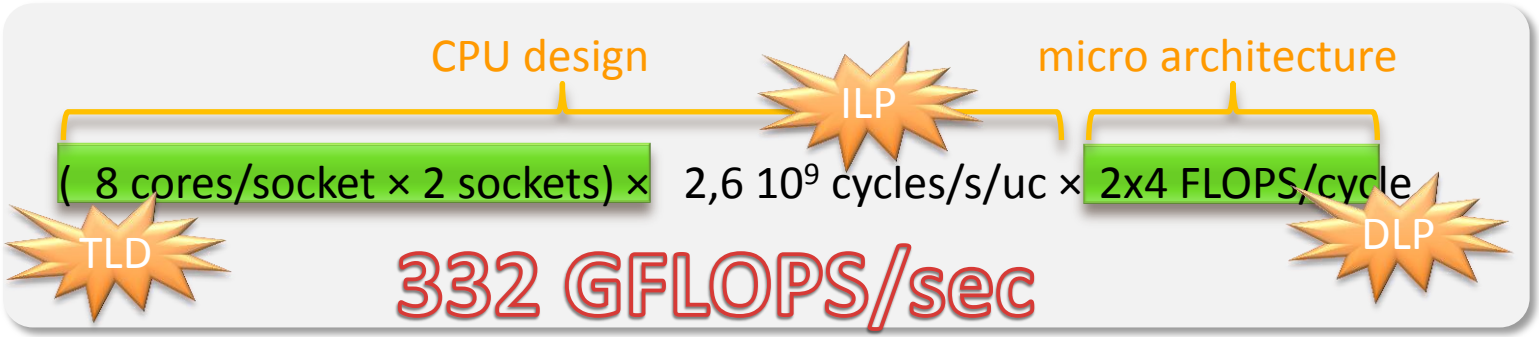
```
vmulpd     ymm1, ymm0, YMMWORD PTR [rax+r14]
vaddpd     ymm2, ymm1, YMMWORD PTR [rdx+r14]
vmovntpd   YMMWORD PTR [rdi+r14], ymm2
```

PEAK PERFORMANCE WESTMERE TO SANDY BRIDGE

Intel® Westmere X5698 (Double precision):



Intel® Sandy Bridge E5-2670 (Double precision):



FLOPS/SEC AND BANDWIDTH

- CPU Frequency is « not an issue »
- Data movement is the issue

The model : “total elapsed time = T_mem + T_cpu + … “ is fine but …

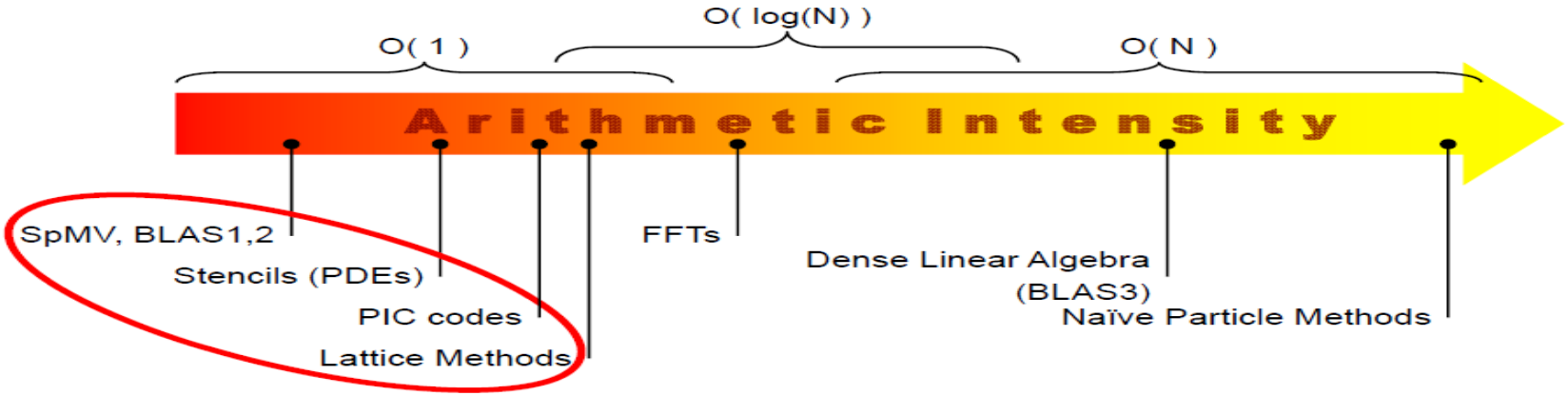
T_CPU and t_mem are strongly correlated

```
#pragma omp parallel for
#pragma unroll(4)
                for(i=0;i<N;i+=1) {

                        C[i]=alpha*A[i]      +B[i]*    D[i];
                        Store        Load      Load      Load
}
```

Achieved Flops/s won’t be high enough if load / store are not fast enough

$$t = t_M + t_C = n_m t_m + n_c t_c = n_c t_c \left(1 + \frac{t_m}{t_c}\frac{1}{q}\right) \text{ avec } q = \frac{n_c}{n_m} \text{ [FLOPS/bytes]}$$

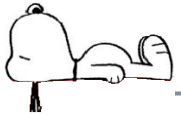
ROOFLINE MODEL



Arithmetic intensity [Williams, Patterson, 2008] :
number of float-point operations to run the program divided by the number of bytes accessed in main memory.

Dense Matrix have an arithmetic intensity that scales with problem size
Many kernels with arithmetic intensities independent of problem size.
For kernels in this former case, weak scaling can lead to different results, since it puts much less demand on the memory system

ROOFLINE MODEL

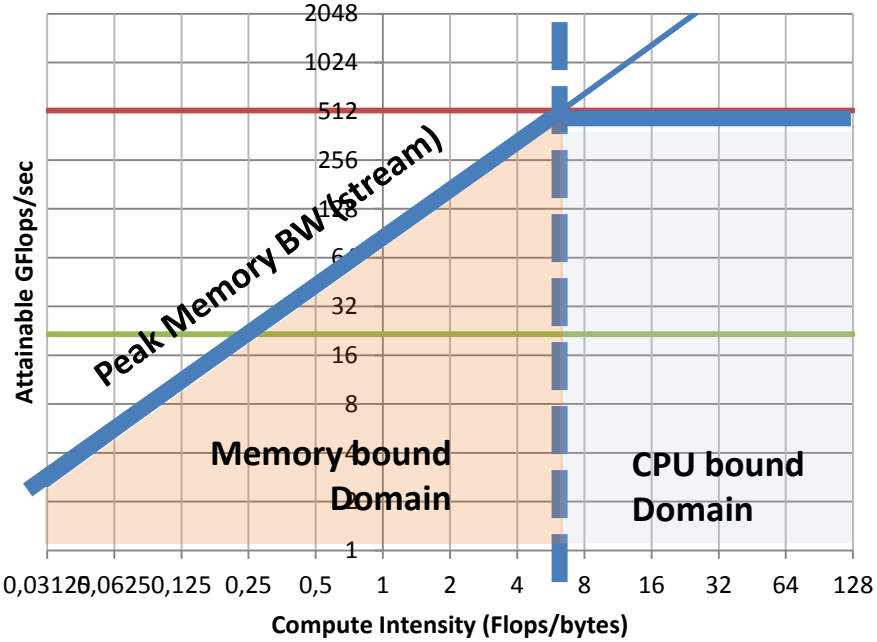


Quel est le « pic » atteignable pour mon application (disons mon kernel) ?

« ROOFLINE MODEL »

- La performance est bornée par le « pic FLOP » théorique de la machine et le produit de la bande passante avec q
  - Memory bound domain : improve prefetch,memory placement …
  - CPU bound domain : improve vectorization

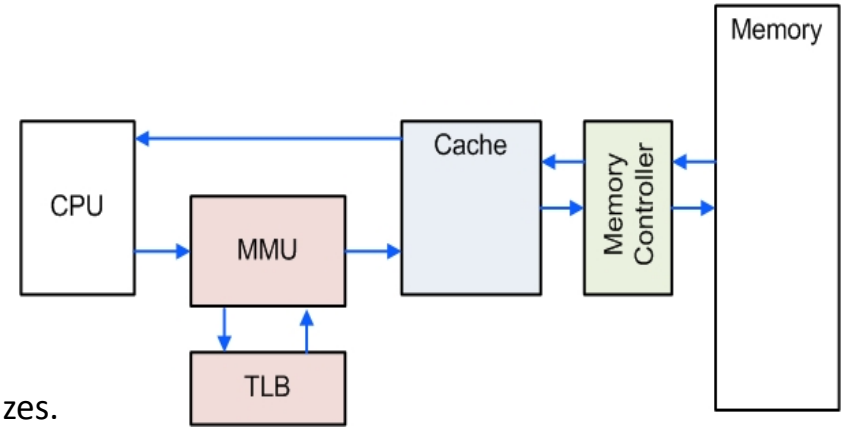
- Example 1 node E5-2697v2 :
  - Ivy Bridge 12 cores, 2.7GHz
  - 21.3 Gflops/s per core
  - 119 GB/s with 1866MHz DRAM



GETTING DATA: HARDWARE

Every memory reference has to go through the Memory Management Unit (MMU)

- virtual into physical address translation.
  - translation table set up by the OS.
  - Translation (page) table maps virtual page numbers to physical frame numbers (starting addresses).



Translation Lookaside Buffer (TLB)

- Small hardware cache
- Keeps the most recent pairs of page/frame numbers.
- Effective due to locality of reference.
- Page size can be changed by the OS from 4kB to huge page sizes.

Memory Controller (MCC)

- Receives a physical address
- Accesses the memory chips
- Bound to a certain memory technology, e.g., DDR2, DDR3, ...
- Integrated in the CPU chip or external, e.g., in the Northbridge

MISE EN OEUVRE DES HUGE PAGES

A l'aide de la libraire libhugetlbfs

- Interface via un pseudo système de fichiers
- Prérequis

```
> mkdir /libhugetlbfs
> groupadd libhp
> chgrp libhp /libhugetlbfs
> chmod 770 /libhugetlbfs
> usermod moi -G libhp
> mount -t hugetlbfs hugetlbfs /libhugetlbfs
```

- Méthode 1 : relinker votre application

```
> gcc -B $HOME/local/lib/libhugetlbfs/ -Wl,--hugetlbfs-link=BDT  mon_programme.c
> ./a.out
```

- Méthode 2 : interposition (ld_preload)

```
> export LD_PRELOAD=/usr/lib64/libhugetlbfs.so
> export HUGETLB_MORECORE=yes
> ./a.out
```

- Transparent huge pages (thp)

```
> echo "always" >/sys/kernel/mm/redhat_transparent_hugepage/enabled
```

CACHE USAGE

Localité temporelle ou spatiale des données

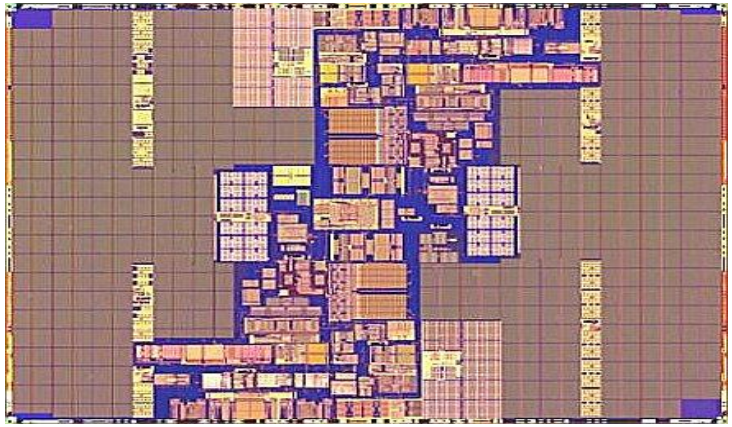
Les caches sont des éléments matériels complexes.

Cette complexité est cachée au programmeur.

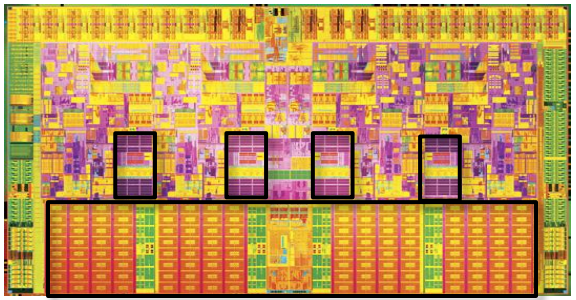
Par contre il existe des règles de bonne usage des caches

- Il faut travailler sur la réutilisation des données
- Leur localité

- Le programmeur peut lui même (à avoir) gérer la cohérence de cache
  - “Explicit cache control”
  - Exemple d’instruction ou principe :
    - prefetching
    - “non temporal stores” ou “streaming stores”



Intel IA64 Montecito (1.6 GHz)



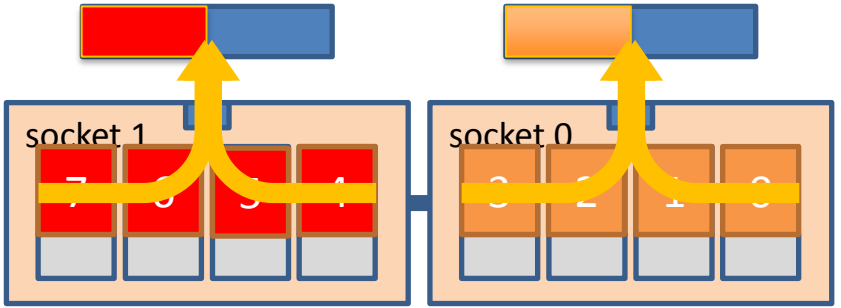
Intel Xeon Nehalem (3.2 GHz)

TASK BINDING

Nous sommes sur des sytèmes CC NUMA : accès à la mémoire depend le placement.

Le noyau dispose d'un petit nombre d'appels systèmes lui permettant de gérer le placement/l'attachement des processus et la gestion mémoire

- **#include <sched.h>**
  - **sched_setaffinity**
  - **sched_getaffinity**
  - **sched_getcpu (glibc > 2.6)**
- **#include <numaif.h>**
  - **mbind**
  - **set_mempolicy**

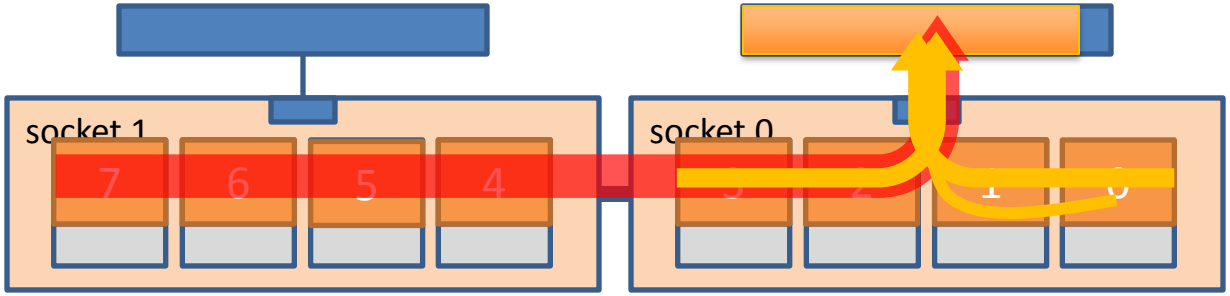


Existe des interfaces noyau de plus haut niveau **CPUSET, MPIRUN, SLURM**

- Un malloc (voire un calloc) alloue la mémoire au niveau virtuelle pas physique
- La page doit être "touchée" pour être physiquement allouée

FIRST TOUCH POLICY

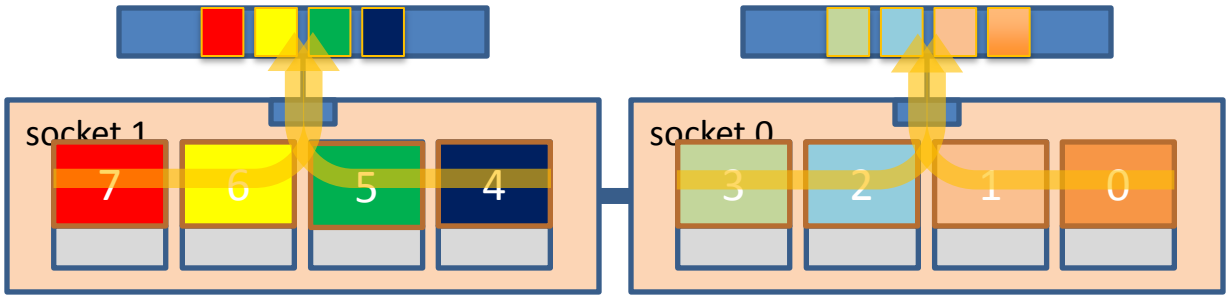
```
// initialisation des données
for(i=0; i<N; i++)
    for(j=0; j<M; j++) { … }

#pragma omp parallel for private(j)
for(i=0; i<N; i++)
    for(j=0; j<M; j++) { … }
```



La moitié des accès mémoires sont distantes !

FIRST TOUCH POLICY

```
#pragma omp parallel for private(j)
// initialisation des données
for(i=0; i<N; i++)
    for(j=0; j<M; j++) { … }

#pragma omp parallel for private(j)
for(i=0; i<N; i++)
    for(j=0; j<M; j++) { … }
```



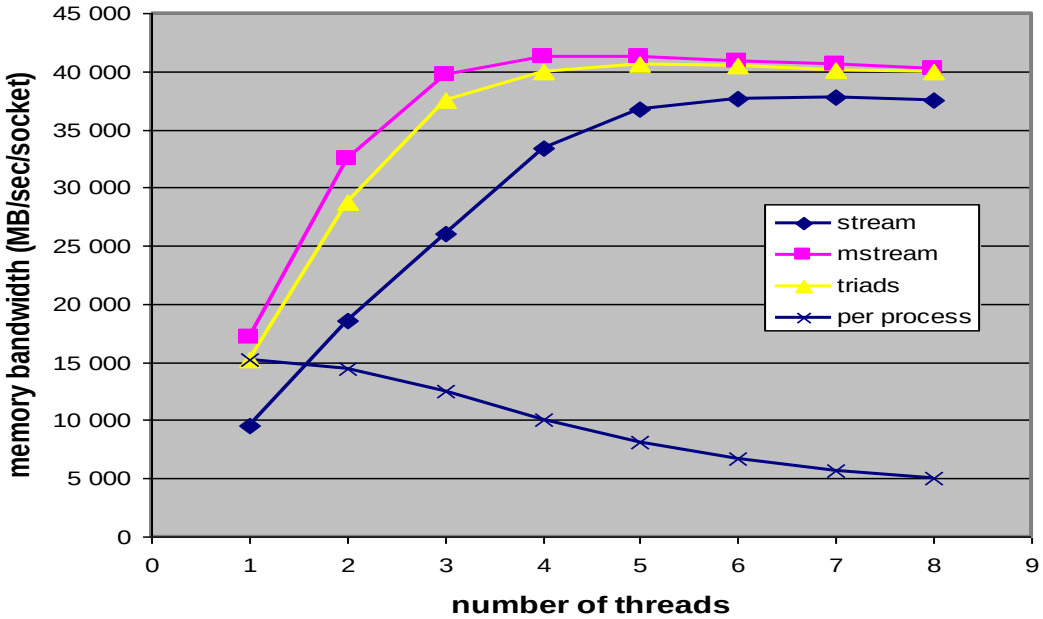
Tous les accès mémoires sont maintenant locaux !

STREAM BENCHMARK

Basical rules for theoretical memory BW [Bytes / second] :

8 [Bytes / channel] * Mem freq [Gcycles/sec] * nb of channels * nb of sockets

Memory bandwith (socket related) in MiB/sec		
threads	triads	per process
8	40 000	5 000
7	40 150	5 736
6	40 462	6 744
5	40 600	8 120
4	39 977	9 994
3	37 480	12 493
2	28 760	14 380
1	15 250	15 250



Intel ® Xeon ® “Sandy Bridge” E5-2690

FIRST TOUCH POLICY | EXEMPLE

Un exemple:

Code stream modifié

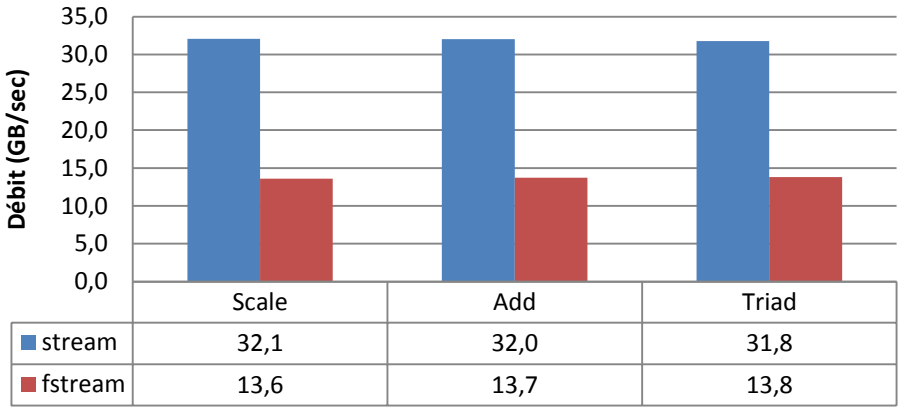
```
…
/* Get initial value for system clock. */
#pragma omp parallel for
    for (j=0; j<N; j++) {
         a[j] = 1.0;
         b[j] = 2.0;
         c[j] = 0.0;
         }
…
```

Comment résoudre le problème:

- Corriger le code si c'est possible (sources)
- Si ce n'est pas possible, avec numactl envisager l'interleaving

Politique d'interleaving:

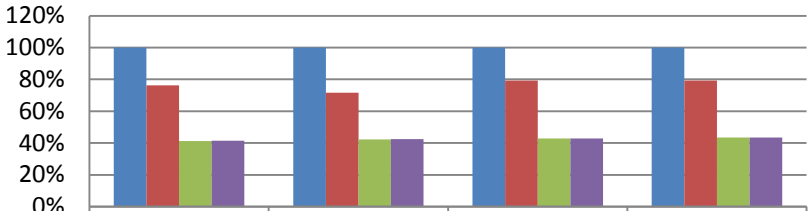
- L'allocateur distribue les pages en "round-robin" sur les différents noeuds (ceux qu'on lui spécifie).
- On augmente la probabilité de Hits



	Scale	Add	Triad
stream	32,1	32,0	31,8
fstream	13,6	13,7	13,8

FIRST TOUCH POLICY | INTERLEAVE

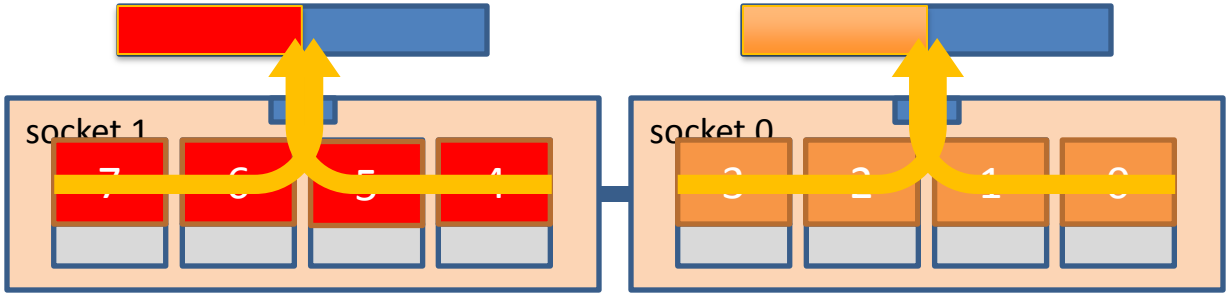
	numactl --interleave=0-1 ./fstream	numactl –membind=1 ./fstream	./fstream
	node0 node1 numa_hit **588005** **587422** numa_miss 0 0 numa_foreign 0 0 interleave_hit **587315** **587274** local_node **587898** 178 other_node 107 **587244**	node0 node1 numa_hit 749 **1174717** numa_miss 0 0 numa_foreign 0 0 interleave_hit 0 0 local_node 749 221 other_node 0 **1174496**	node0 node1 numa_hit **1175137** 215 numa_miss 0 0 numa_foreign 0 0 interleave_hit 0 0 local_node **1175137** 215 other_node 0 0
Copy	18515	10015	10057
Scale	23000	13574	13593
Add	25392	13749	13732
Triad	25166	13794	13797



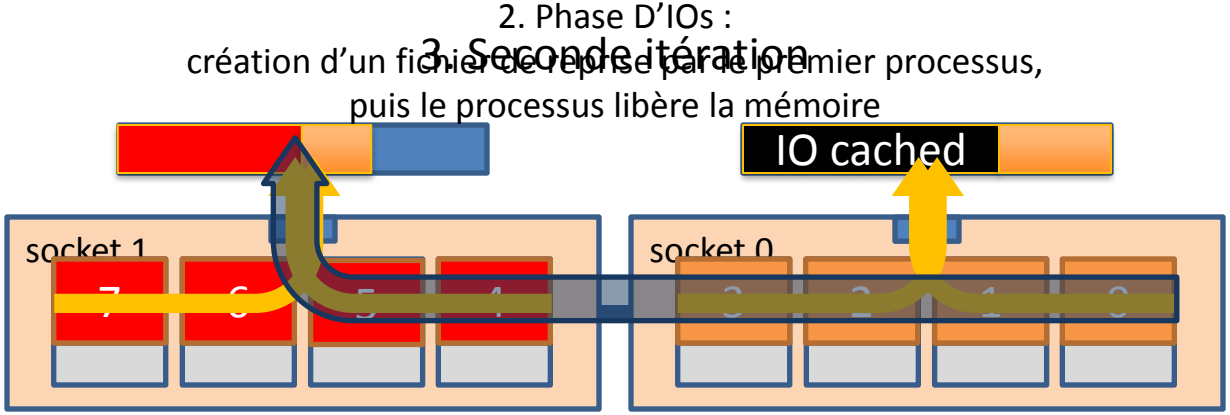
	Copy	Scale	Add	Triad
max	100%	100%	100%	100%
interleave	76%	72%	79%	79%
membind=0	41%	42%	43%	43%
default	41%	42%	43%	43%

GESTION DU CACHE IO ET EFFETS DE BORDS

1. Première Itération



GESTION DU CACHE IO ET EFFETS DE BORDS



Par défaut, linux alloue la mémoire sur le noeud local et sur un noeud distant si la mémoire locale est occupée par des entrées du cache IOs. Cette politique est ajustable via l'entrée du noyau **`/proc/sys/vm/zone_reclaim_mode`** ou **`vm.zone_reclaim_mode`**.

PHYSICS CODES

```
do k over depth/height
   do j over y grid
      do i over x grid

         access arrays with not much re-use
         no clear kernel

         CPU and memory-bandwidth bound ..

 Running a i,j,k grid          500          200          80
 Time to execute mm subroutine:      0.416193962097168   sec
                 mm v1:              0.385792016983032   sec
                 mm2 block: 100x50   0.470975875854492   sec
```

Sample Earth to run a simulation in a couple of hours.
Decomposition of the physical horizontal grid on a x-y processor grid.

Traditionally running on vector machines.
Fortran example for register pressure and blocking

INTEL COMPILER REPORTS

Very detailed information of all work done by the compiler
a special vectorization report, can be piped into other scripts
-vec-report<n>

-opt-report-phase=

- ipo_inl
  - Interprocedural Optimization Inlining Report
- ilo
  - Intermediate Language Scalar Optimization
- hpo
  - High Performance Optimization
- hlo
  - High-level Optimization
- pgo
  - Profile Guided Optimizer

-guide

- get advice on how to help the compiler to vectorize loops

EXAMPLE : HLO OPTIMIZATION REPORTS

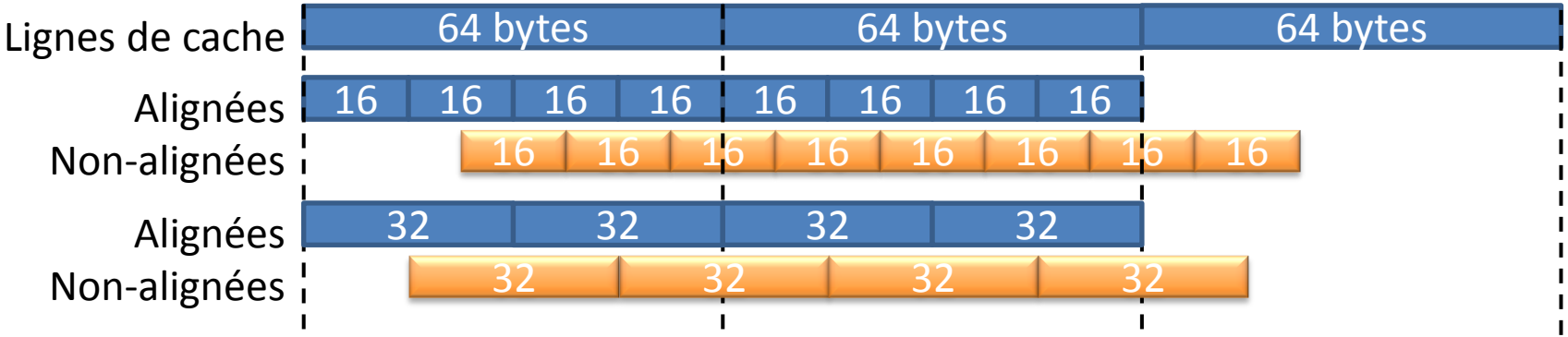
```
%ifort -O3 -opt_report_phase hlo -opt-report-phase hpo matmul.f90
HPO VECTORIZER REPORT (matmul_)
…
matmul.f90(9:1-9:1):VEC:matmul_: PERMUTED LOOP WAS VECTORIZED
…
High Level Optimizer Report (matmul_)
#of Array Refs Scalar Replaced in matmul_ at line 9=36
…
<matmul.f90;9:9;hlo_linear_trans;matmul_;0>
LOOP INTERCHANGE in loops at line: 9 8 7
Loopnest permutation ( 1 2 3 ) --> ( 2 3 1 )
…
<matmul.f90;9:9;hlo_unroll;matmul_;0>
Loop at line 9 blocked by 128
…
Loop at line 7 blocked by 128
Loop at line 8 blocked by 128
Loop at line 8 unrolled and jammed by 4
Loop at line 7 unrolled and jammed by 4
```

```
subroutine matmul(a,b,c,n)
real(8) a(n,n),b(n,n),c(n,n)
do j=1,n
   do i=1,n
      do k=1,n
         c(j,i)=c(j,i)+a(k,i)*b(j,k)
      enddo
   enddo
enddo
end
```

ALIGNEMENT DES DONNÉES

Peut être pénalisant.

Mieux vaut aligner les données sur la taille des vecteurs.



INTEL DATA ALIGNMENT

Compiler can do best optimizations if loads /stores are aligned

- 16 Bytes boundaries for SSE, 32Byte for AVX and 64Bytes for MIC
- Could be imposed for static data
- (v)movupd (vectorload) if unaligned 2x128 stores may be faster than 1x256 unaligned store.

```
__declspec(align(32)) X[1000];
void foo(float *restrict a, … )
__assume_aligned(a,32)
__assume(n1%8=0);
__assume(n2%8=0);
for(i=0;i<n;i++) X[i] +=
a[i+n1]
}
```

- Use special malloc libraries
- posix_memalign(void **memptr, size_t alignment, size_t size);
- Use in C void* _mm_malloc (int size, int n)
  - Compiler creates an n-byte boundary aligned pointer to memory.
- C __declspec(align(n, [offset])) Fortran !dir$ attributes align:n::varname
  - Compiler creates the variable aligned on an “n”-byte boundary, with an “offset” in bytes.

- C __assume_aligned(a,n) Fortran !dir$ assume_aligned varname:n
  - Instructs the compiler to assume that array a is aligned on an n
- #pragma vector aligned
  - Vectorize using aligned loads /stores for vector accesses

INTEL COMPILER DATA ALIGNMENT OPTIONS

Fortran Compiler options
-align <keyword>
[no]commons, [no]dcommons,[no]qcommons, [no]zcommons,
rec1byte, rec2byte, rec4byte,rec8byte, rec16byte, rec32byte,
array8byte, array16byte, array32byte,array64byte,
array128byte, array256byte

-falign-functions=[2|16]
    align the start of functions on a 2 (DEFAULT) or 16 byte boundary

-Zp[n] specify alignment constraint for structures

In C
-[no]align
    analyze and reorder memory layout for variables and arrays

```
subroutine foo(a,f2,st,rep,rstr)
real*8 :: a(*),a1
integer f2,st,rep,rstr,i
!DIR$ ASSUME_ALIGNED A: 64
!DIR$ ASSUME (mod(f2,8) .eq. 0)
!dir$ simd
do i=1,(rep-1)*rstr+1
  a1=a(i)+a(f2*st+i)
  a(f2*st+i)=a(i)-a(f2*st+i)
  a(i)=a1
enddo
end subroutine
```

ALIGNEMENT DES DONNÉES

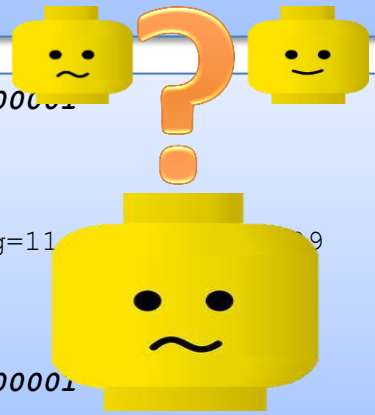
```
#define N 1000000000
double A[N] ;
double B[N] ;
double S[N] ;

#pragma vector aligned(A,B,C)
        for(i=0;i<N;i+=1)
        {
                C[i]=0.0;
                A[i]=1.0;
                B[i]=0.1;
        }
```

segfault

```
#define N 1000000000
double A[N] __attribute__((aligned(64)));
double B[N] __attribute__((aligned(64)));
double S[N] __attribute__((aligned(64)));

#pragma vector aligned(A,B,C)
        for(i=0;i<N;i+=1)
        {
                C[i]=0.0;
                A[i]=1.0;
                B[i]=0.1;
        }
```

ALIGNEMENT DES DONNÉES

```
#define N 1000000000
double A[N] __attribute__((aligned(64)));
double B[N] __attribute__((aligned(64)));
double S[N] __attribute__((aligned(64)));

#pragma omp parallel for
#pragma vector aligned(A,B,C)
        for(i=0;i<N;i+=1)
        {
                C[i]=0.0;
                A[i]=1.0;
                B[i]=0.1;
        }
```

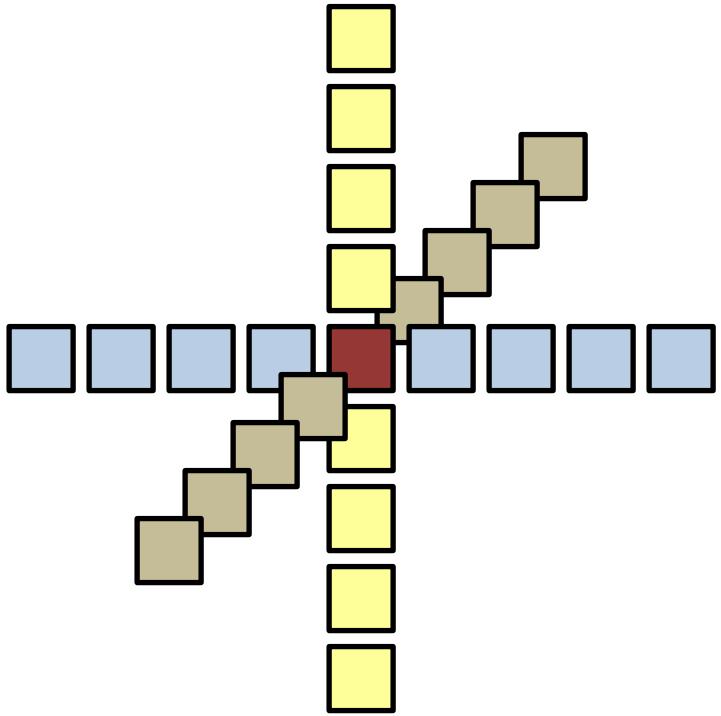


```
> OMP_NUM_THREADS=10 ./daxpy.x 1.000001
Go!
Done!
i=1000000000 j=10
T=15972239952 cycles, 5.70 sec, avg=11
42072.07051 MB/s
3506.00588 MFLOPS/s

> OMP_NUM_THREADS=12 ./daxpy.x 1.000001
Segmentation fault
```

INTEL TUNING EXAMPLE FINITE DIFFERENCE METHOD

Isotropic Wave Equation :

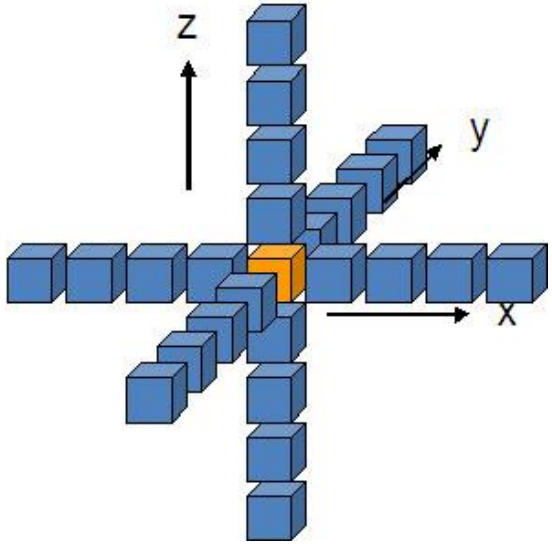
$$\frac{1}{v^2}\frac{\partial^2 P}{\partial t^2} = \frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2} + \frac{\partial^2 P}{\partial z^2}$$



- order-*k* in space stencil (here *k* is 4 or 8)
- memory bandwidth bound code
- Written in C/C++
- Starting version using aligned arrays
- Performance Measure : Mpoints/second

FINITE DIFFERENCES - STENCIL COMPUTATION

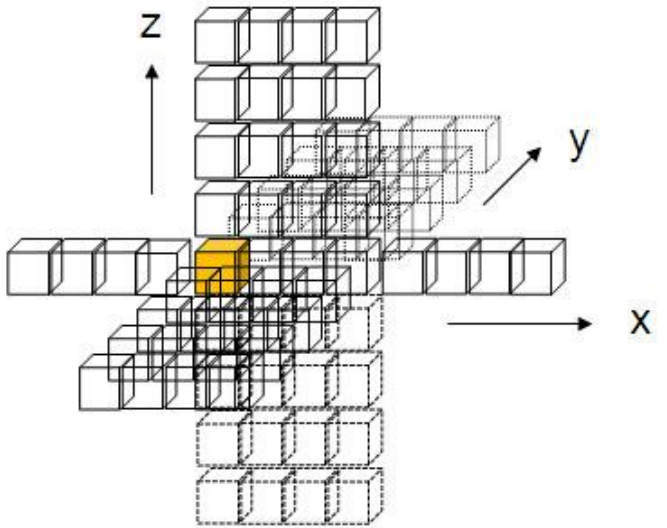
```
for (i over Z){
 for (j over Y){
   for (k over X){
   Lapl_xx= x*wavefield_t1[i][j][k]
         + (wavefield_t1[i][j][k-1]+ wavefield_t1[i][j][k+1])
         + (wavefield_t1[i][j][k-2]+ wavefield_t1[i][j][k+2))
         + (wavefield_t1[i][j][k-3]+ wavefield_t1[i][j][k+3])
         + (wavefield_t1[i][j][k-4]+ wavefield_t1[i][j][k+4]);

   Lapl_yy= y*wavefield_t1[i][j][k]
         + (wavefield_t1[i][j-1][k]+ wavefield_t1[i][j+1][k])
         + (wavefield_t1[i][j-2][k]+ wavefield_t1[i][j+2][k])
         + (wavefield_t1[i][j-3][k]+ wavefield_t1[i][j+3][k])
         + (wavefield_t1[i][j-4][k]+ wavefield_t1[i][j+4][k]);
```



No problem for vectorization, parallelization or loop changes.
3 dimensions * 8 values = 24+1 point stencil
A single stencil computation underutilizes 4*4=16 cache lines by accessing only one floating-point value from each of these cache-lines.

FINITE DIFFERENCES - STENCIL COMPUTATION

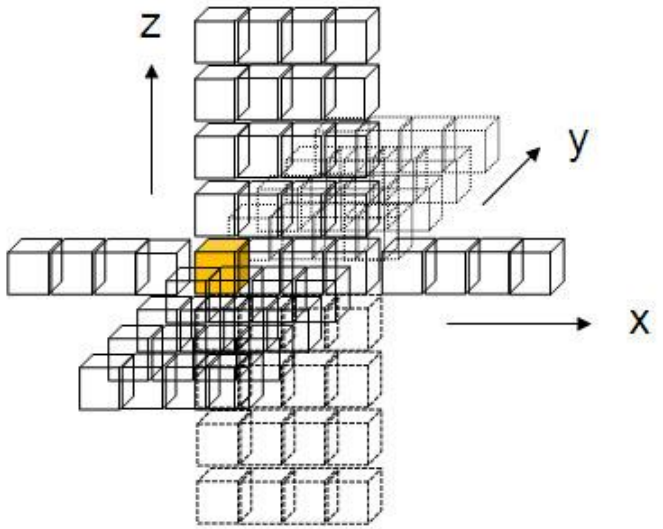
```
for (i over Z){
 for (j over Y){
   for (k over X){
   Lapl_xx= x*wavefield_t1[i][j][k]
         + (wavefield_t1[i][j][k-1]+ wavefield_t1[i][j][k+1])
         + (wavefield_t1[i][j][k-2]+ wavefield_t1[i][j][k+2))
         + (wavefield_t1[i][j][k-3]+ wavefield_t1[i][j][k+3])
         + (wavefield_t1[i][j][k-4]+ wavefield_t1[i][j][k+4]);

   Lapl_yy= y*wavefield_t1[i][j][k]
         + (wavefield_t1[i][j-1][k]+ wavefield_t1[i][j+1][k])
         + (wavefield_t1[i][j-2][k]+ wavefield_t1[i][j+2][k])
         + (wavefield_t1[i][j-3][k]+ wavefield_t1[i][j+3][k])
         + (wavefield_t1[i][j-4][k]+ wavefield_t1[i][j+4][k]);
```



19 SSE registers being used to compute 4 stencils simultaneously. At least four floating-point elements are loaded from each cache-line, and twelve floating point elements belong to the cache-line(s) holding the X-direction elements.

FINITE DIFFERENCES - STENCIL COMPUTATION

```
for (i over Z){
 for (blocked loop j over Y){
   for (blocked loop k over X)

   for (k over X)
   Lapl_xx [k] = x*wavefield_t1[i][j][k]
         + (wavefield_t1[i][j][k-1]+ wavefield_t1[i][j][k+1])
         + (wavefield_t1[i][j][k-2]+ wavefield_t1[i][j][k+2))
         + (wavefield_t1[i][j][k-3]+ wavefield_t1[i][j][k+3])
         + (wavefield_t1[i][j][k-4]+ wavefield_t1[i][j][k+4]);
   for (k over X)
   Lapl_yy [k] = y*wavefield_t1[i][j][k]
         + (wavefield_t1[i][j-1][k]+ wavefield_t1[i][j+1][k])
         + (wavefield_t1[i][j-2][k]+ wavefield_t1[i][j+2][k])
         + (wavefield_t1[i][j-3][k]+ wavefield_t1[i][j+3][k])
         + (wavefield_t1[i][j-4][k]+ wavefield_t1[i][j+4][k]);
```



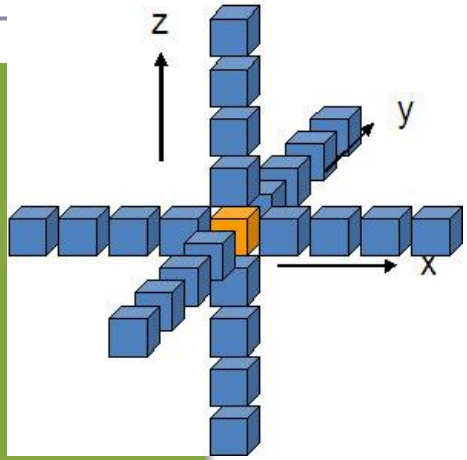
Optimization with:

Block on Y and on X

Use an additional “helper loop” over k called writing into small vectors in a separate function.

INTEL TUNING EXAMPLE

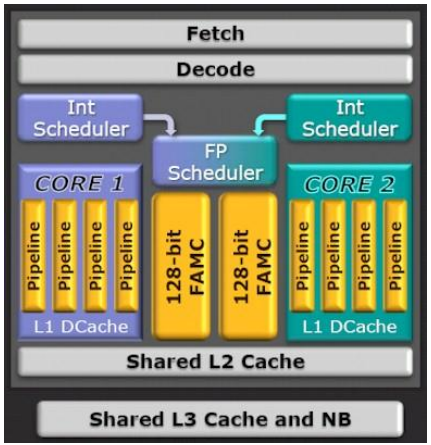
```
V01 Original version base time
./iso3dfd_dev01_cpu_avx.exe 400 300 800 20 10
throughput:   576.88 MPoints/s

V02 with loop blocking
throughput:   646.82 MPoints/s

V04 ivdep and loop unroll directive
throughput:   755.78 MPoints/s

V05 Macros FINITE_ADD, assume_aliged, manual unrolling of inner loop
throughput:  1368.46 MPoints/s

V06 Using Macros and optimizing multiplies with the coeffecients
throughput:  1397.78 MPoints/s

V07 First touch, streaming stores
throughput:  2122.33 MPoints/s

V08 Usage of vector intrinsics and optimal blocking
throughput:  2207.68 MPoints/s
```



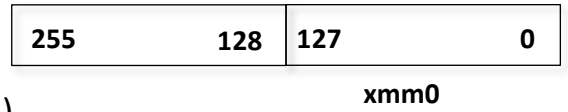
INTEL AND AMD SSE/AVX

2007: Intel (Woodcrest) and AMD (Barcelona) SSE 128bit (data paths and FP units)
2x speed-up of vectorized codes for DP using packed SSE instructions
Chips suffering from peak transfer rate vs peak FP performance (L1,L2,Memory)

2011: Intel (SandyBridge) and AMD (Bulldozer) AVX 256bit
again a possible 2x speed-up of vectorized codes …
Impressive : with even more cores, the peak transfer rate vs peak FP performance has slightly improved (additional ports, faster memory).
VEX prefix 256 SIMD support can run 3 or 4 operand syntax (compared to 2 for x86 ISA)
AVX needs vectorization



The 2x can not be reached for real applications.
From Westmere (Nehalem) to SandyBridge not all components scale 2x (especially the L2/L3 cache bandwidth)
LMBench. Expect a 1.2-1.4x !



AMD shares FP unit for 2 cores. AMD extends FMA4 instructions (fused multiply-add).
AMD can run the threads using floating points in 256-bit AVX mode (scheduling ymm-based operations over the entire FP unit) or can use just one lane of the shared FP unit (xmm 128-bit) with VEX or SSE code.

CONCLUSION COMPILERS OPTIMIZATION

Compiler works well

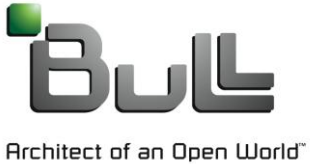
- Local Optimization on basic blocks (branchless statements)
    - common sub-expression elimination, redundant load and store elimination
    - scheduling, strength reduction, peephole optimizations.
- Loop Optimization
    - unrolling, vectorization
- Function Inlining
    - increase code size and generate less efficient code
- Global Optimization (ipo)

Compiler needs guidance

- Loop and “adjacent loops” optimization
    - Peeling, blocking, fusion or fission of adjacent loops :
        - Data locality and re-usage (change from memory bandwidth bound to cpu bound)
    - data alignment
    - Block into L3 to optimize TLB
- Global Optimization using Profile-Feedback Optimization (PFO), profile guided (PGO)

THANK YOU !

GUNTER.ROETH@BULL.NET



STATISTIQUES NUMA

Disponibles via :

- `/sys/devices/system/node/node*/numastat`

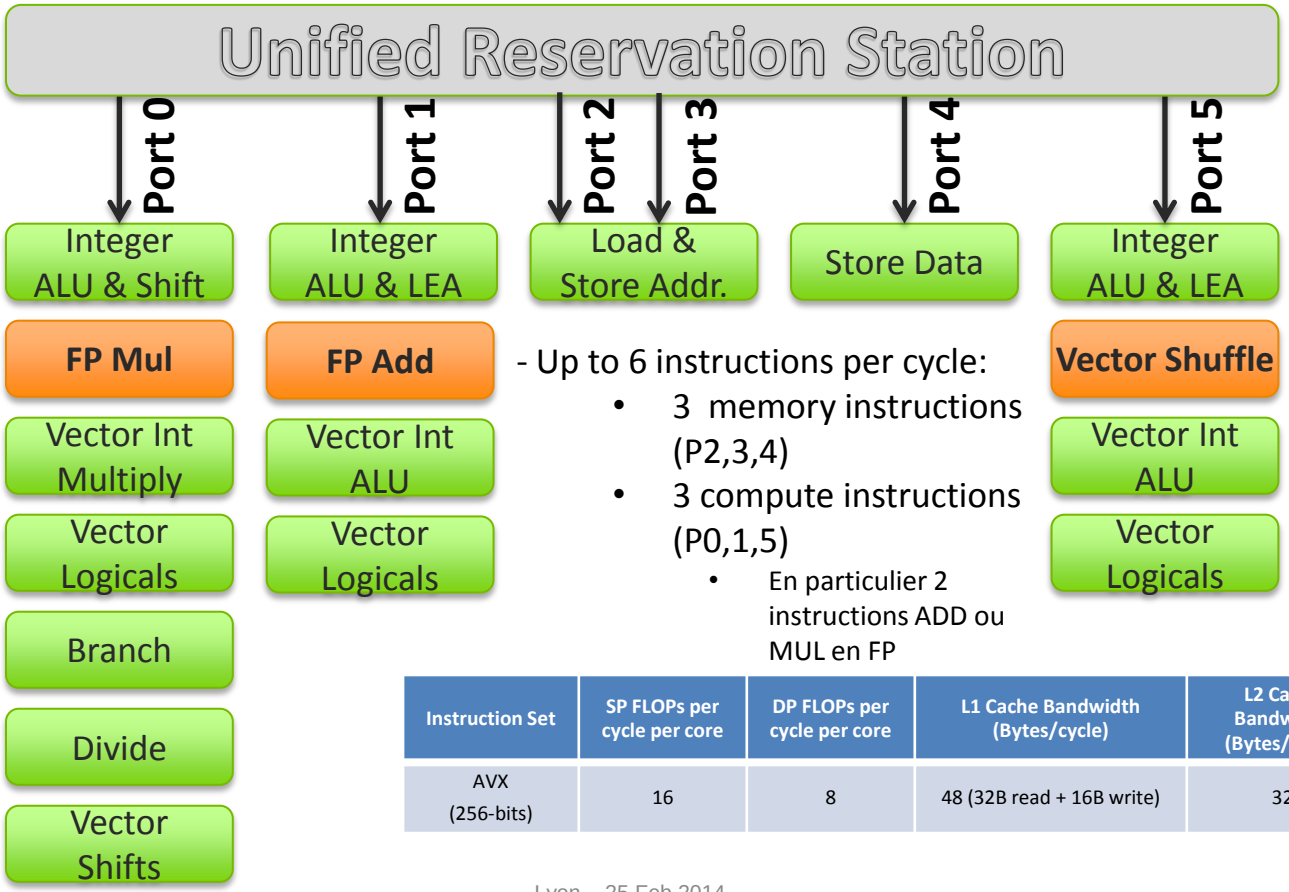
la commande `numastat`

- L'unité est la page (4K)
- Tout ça mérite d'être enrobé

```
> numastat
                           node0           node1
numa_hit            37270443889     41299449921
numa_miss            1738099296      3204337700
numa_foreign         3204337700      1738099296
interleave_hit          1347000         1243207
local_node          37266097532     41259383644
```

variable	Description
`numa_hit`	numa_hit is the number of allocations where an allocation was intended for that node and succeeded there.
`numa_miss`	numa_miss shows how often an allocation was intended for this node, but ended up on another node due to low memory.
`numa_foreign`	numa_foreign is the number of allocations that were intended for another node, but ended up on this node. Each numa_foreign event has a numa_miss on another node.
`interleave_hit`	interleave_hit is the number of interleave policy allocations that were intended for a specific node and succeeded there.
`local_node`	local_node is incremented when a process running on the node allocated memory on the same node.
`other_node`	other_node is incremented when a process running on another node allocated memory on that node.

SANDY BRIDGE: EXECUTION UNITS



- Up to 6 instructions per cycle:
  - 3 memory instructions (P2,3,4)
  - 3 compute instructions (P0,1,5)
    - En particulier 2 instructions ADD ou MUL en FP

Instruction Set	SP FLOPs per cycle per core	DP FLOPs per cycle per core	L1 Cache Bandwidth (Bytes/cycle)	L2 Cache Bandwidth (Bytes/cycle)
AVX (256-bits)	16	8	48 (32B read + 16B write)	32

COMPILER TARGETS : INTEL AND AMD

Intel SandyBridge L5330, dual socket 8 cores per socket
Intel MIC Xeon Phi Co-processor >50 cores on a socket
AMD Bulldozer 2 integer cores sharing a FP unit 16 cores per socket

Very different CPU architecture
all processors need vectorization to run at highest possible speed !