Modern Architectures & Optimization

Luca Tornatore - I.N.A.F.

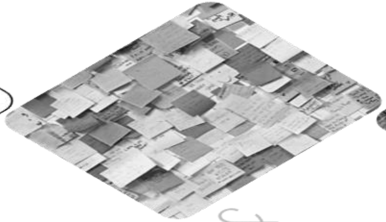
"Foundation of HPC" course

DATA SCIENCE & SCIENTIFIC COMPUTING

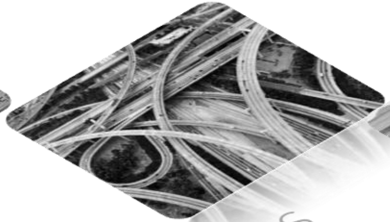
2020-2021 @ Università di Trieste

Outline

First things first

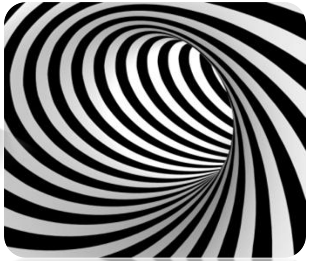


Cache & Memory

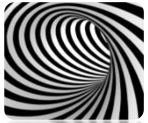


Branches

Pipelines

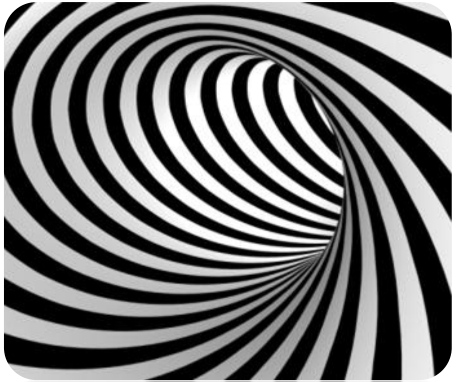


Loops



Outline

Avoid the avoidable inefficiencies

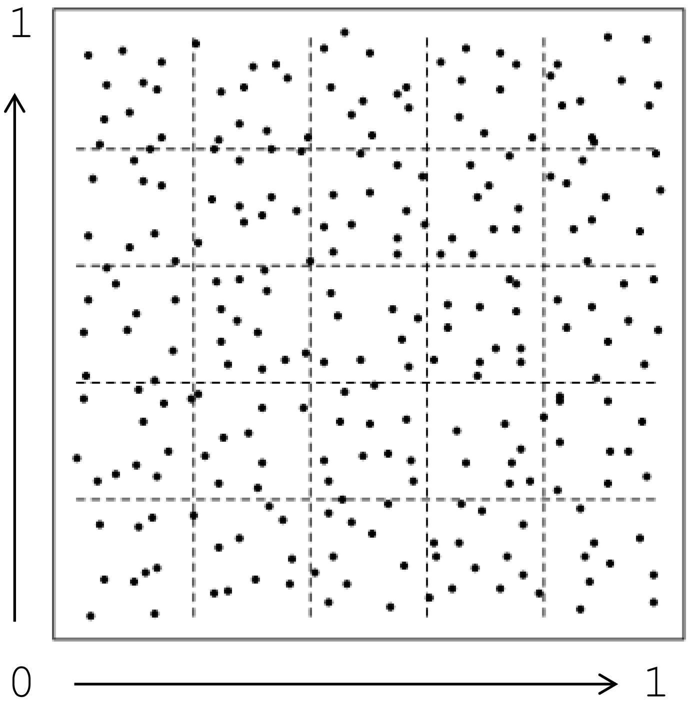


Loops techniques

Prefetching

Let's suppose that

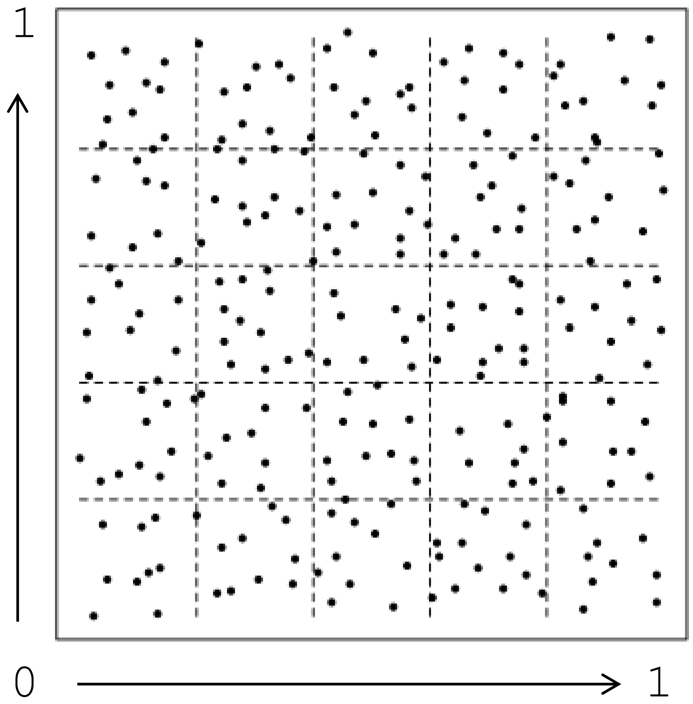
1) we have a distribution of random data points on a 2D plane which we subdivide in sub-regions using a grid.



Let's suppose that

1) we have a distribution of random data points on a 2D plane which we subdivide in sub-regions using a grid.

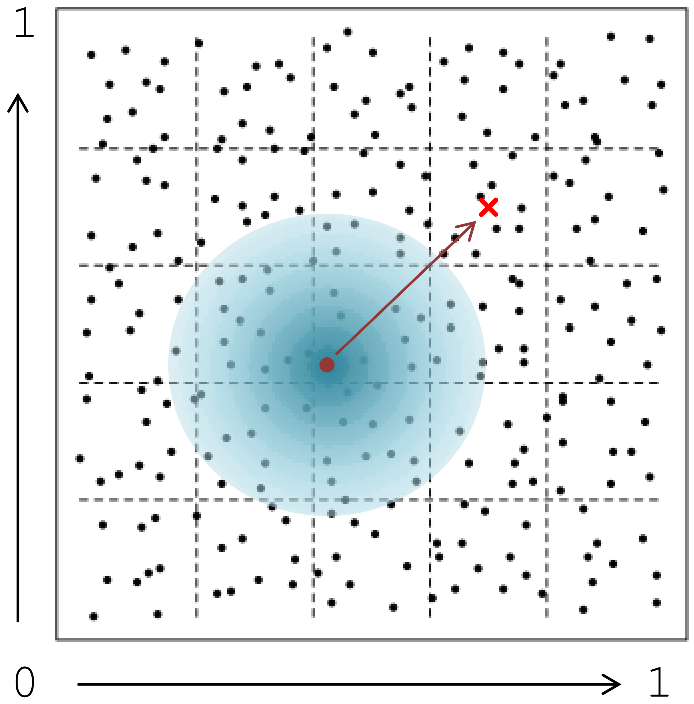
2) for each point *p*, we want to select all the grid cells whose center is closer to *p* than a given radius *r*, and to perform some operations accordingly to our search result.



Let's suppose that

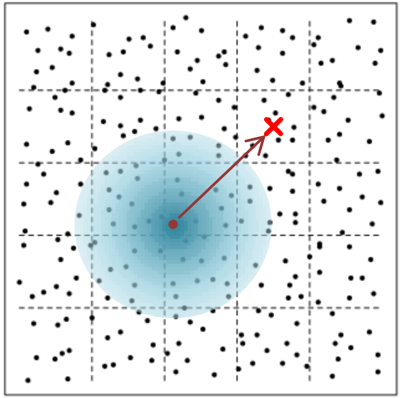
1) we have a distribution of random data points on a 2D plane which we subdivide in sub-regions using a grid.

2) for each point *p*, we want to select all the grid cells whose center is closer to *p* than a given radius *r*, and to perform some operations accordingly to our search result.



We may consider to use a nested loop like this one →

Is there anything you would change in this loop?



```
for(p = 0; p < Np; p++)

    for(i = 0; i < Ng; i++)
      for(j = 0; j < Ng; j++)
        for(k = 0; k < Ng; k++)
          {
            dist = sqrt(
                    pow(x[p] - (double)i/Ng - half_size, 2) +
                    pow(y[p] - (double)j/Ng - half_size, 2) +
                    pow(z[p] - (double)k/Ng - half_size, 2));

               if(dist < R)
               do something;
          }
```

(1) Avoid expensive function calls

Some function calls are particularly expensive. Those include, among others, `sqrt()`, …

Try to avoid them *if possible*.

```
for(p = 0; p < Np; p++)

  for(i = 0; i < Ng; i++)
    for(j = 0; j < Ng; j++)
      for(k = 0; k < Ng; k++)
        {
          dist2 = pow(x[p] - (double)i/Ng - half_size, 2) +
                  pow(y[p] - (double)j/Ng - half_size, 2) +
                  pow(z[p] - (double)k/Ng - half_size, 2));

                  if(dist2 < R2)
                  do something;
        }
```

(1) Avoid expensive function calls

Some function calls are particularly expensive. Those include, among others, `sqrt()`, `pow()`, …

Try to avoid them *if possible*.

```
for(p = 0; p < Np; p++)

  for(i = 0; i < Ng; i++)
    for(j = 0; j < Ng; j++)
      for(k = 0; k < Ng; k++)
        {
          dx = x[p] - (double)i/Ng - half_size;
          dy = y[p] - (double)j/Ng - half_size;
          dz = z[p] - (double)k/Ng - half_size;

          dist2 = dx*dx + dy*dy + dz*dz;
          if(dist2 < R2)
              do something;
        }
```

(1) Avoid expensive function calls

Some function calls are particularly expensive. Those include, among others, `sqrt()`, `pow()`, **`floating point division`**, ..
Try to avoid them *if possible*.

```
for(p = 0; p < Np; p++)

  for(i = 0; i < Ng; i++)
    for(j = 0; j < Ng; j++)
      for(k = 0; k < Ng; k++)
        {
          dx = x[p] - (double)i * Ng_inv - half_size;
          dy = y[p] - (double)j * Ng_inv - half_size;
          dz = z[p] - (double)k * Ng_inv - half_size;

          dist2 = dx*dx + dy*dy + dz*dz;
          if(dist2 < R2)
              do something with sqrt(dist2);
        }
```

(1) Avoid expensive function calls

```
(double)<i,j,k> * Ng_inv + half_size
```

was performed N^3+N^2+N times, always returning the same values.
Hoisting would save
$N(N^2+N^1+1)$ **mul**, **add** and **mem** accesses.

You can do better pre-computing the relevant values:

```
double ijk[Ng];
for(i = 0; i < Ng; i++)
   ijk[i] = i * Ng_inv + half_size
```

```
for(p = 0; p < Np; p++)

    for(i = 0; i < Ng; i++)
      for(j = 0; j < Ng; j++)
        for(k = 0; k < Ng; k++)
          {
            dx = x[p] - (double)i * Ng_inv - half_size;
            dy = y[p] - (double)j * Ng_inv - half_size;
            dz = z[p] - (double)k * Ng_inv - half_size;

            dist2 = dx*dx + dy*dy + dz*dz;
            if(dist2 < R2)
               do something with sqrt(dist2);
          }
```

(2) *Hoisting* of expressions

```
for(i = 0; i < Ng; i++) {
   dx2 = x[p] - (double)i * Ng_inv - half_size;
   dx2 = dx2*dx2;

     for(j = 0; j < Ng; j++) {
        dy2 = y[p] - (double)j * Ng_inv - half_size;
        dy2 = dy2*dy2;
        dist2_xy = dx2 + dy2;

          for(k = 0; k < Ng; k++) {
             dz = z[p] - (double)k * Ng_inv - half_size;
             dist2 = dist2_xy + dz*dz;
             if(dist2 < Rmax2)
               do something with sqrt(dist2);  } } }
```

`(double)<i,j,k> * Ng_inv + half_size`

was performed N^3+N^2+N times, always returning the same values.
Hoisting would save
$N(N^2+N^1+1)$ **`mul`**, **`add`** and **`mem`** accesses.

(2) *Hoisting* of expressions

You could do even better by pre-computing the relevant values:

```
double ijk[Ng];
for(i = 0; i < Ng; i++)
   ijk[i] = i * Ng_inv + half_size
```

```
for(i = 0; i < Ng; i++) {
   dx2 = x[p] - Ng_inv[i] - half_size;
   dx2 = dx2*dx2;

   for(j = 0; j < Ng; j++) {
      dy2 = y[p] - Ng_inv[j] - half_size;
      dist2_xy = dx2 + dy2*dy2;

      for(k = 0; k < Ng; k++) {
         dz = z[p] - Ng_inv[k] - half_size;
         dist2 = dist2_xy + dz*dz;
         if(dist2 < Rmax2)
           do something with sqrt(dist2);  } } }
```

(3) Clarify the variables' scope

All these variables are very local, there's no need for them to have a wider scope.

That will help you in writing the code, and *may* help the compiler in optimizing the stack and perhaps the registers usage.

```
for(int i = 0; i < Ng; i++) {
    double dx2 = x[p] - (double)i * Ng_inv - half_size;
    dx2 *= dx2;

        for(j = 0; j < Ng; j++) {
            double dy2 = y[p] - (double)j * Ng_inv - half_size;
            double dist2_xy = dx2 + dy2*dy2;

            for(k = 0; k < Ng; k++) {
                double dz = z[p] - (double)k * Ng_inv - half_size;
                double dist2 = dist2_xy + dz*dz;

                if(dist2 < Rmax2)
                  do something with sqrt(dist2);  } } }
```

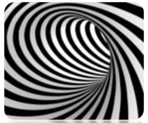
(4) Suggest what is important

These variables are often calculated and reused subsequently.

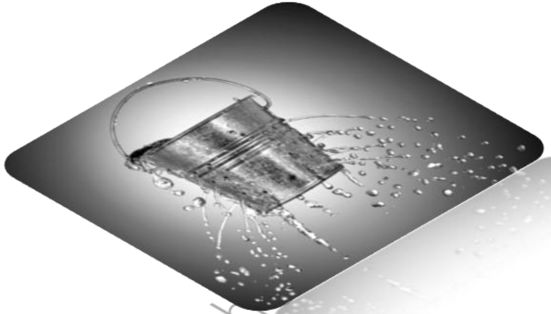
Keeping a register dedicated to them may be useful.

Note: *this is a suggestion, the compiler, after analyzing the code, may decide differently*

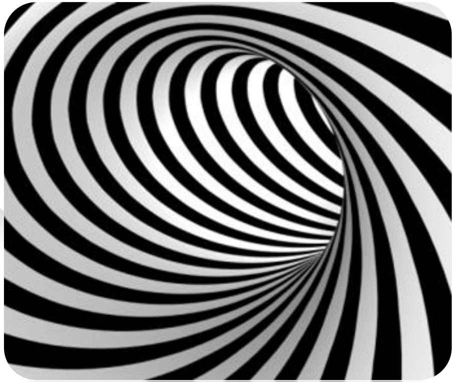
```
double register Ng_inv = 1.0 / Ng;
for(int i = 0; i < Ng; i++) {
   double dx2 = x[p] - (double)i * Ng_inv - half_size;
   dx2 *= dx2;

      for(j = 0; j < Ng; j++) {
         double dy2 = y[p] - (double)j * Ng_inv - half_size;
         dy2 *= dy2;
         double register dist2_xy = dx2 + dy2;

         for(k = 0; k < Ng; k++) {
            double register dz = z[p] - (double)k * Ng_inv - …;
            double register dist2 = dist2_xy + dz*dz;

            if(dist2 < Rmax2)
              do something with sqrt(dist2);  } } }
```



Outline



Avoid th
avoidable
inefficiencies



Loops techniques

Prefetching

Optimizing cache access in loops

Loop classification

$$A_I = \frac{f(n)}{n}$$

Arithmetic Intensity: the **ratio** between the **number of performed operations** and the **amount of the data**.

1. O(N) / O(N)
   optimization potential limited

2. $O(N^2)$ / $O(N^2)$
   some more opportunities for opt.

3. $O(N^3)$ / $O(N^2)$
   significant optimization potential

Cache access in loops: $O(N)/O(N)$

Example

1-level loops: Scalar products, vector additions, sparse matrix-vector multiplication

Inevitably memory-bound for very large *N*; in general, improvements come from *avoiding unnecessary operations* and/or *repeated memory accesses*, and increasing **data reuse**

[check the room for loops fusion]

O(N) / O(N)

```
for(int j=0; j<2; j++)
  A[i] = B[i] × C[i]


for(int j=0; j<2; j++)
  Q[i] = B[i] + D[i]
```

➔

```
for(int j=0; j<2; j++)
  {
    A[i] = B[i] × C[i]
    Q[i] = B[i] + D[i]
  }
```

Loop fusion: in the version on the right, B is recalled from memory only once.

Cache access in loops: $O(N^2)/O(N^2)$

Example

2-levels loops: dense matrix-vector mul, matrix transpos., matrix add, …

Improvements comes again from increasing *data reuse*, exploiting *locality* and *avoiding unnecessary* operations and memory accesses.

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

N×N N×N N×N

→ $3 \times N^2$ memory accesses

Cache access in loops: $O(N^2)/O(N^2)$

Step 1:
Avoid unecessary loads /stores

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

→

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

Now it is clearer for the compiler that `C[i]` need to be loaded and stored only 1 time

→ $2\times N^2+N$ memory accesses

Cache access in loops: $O(N^2)/O(N^2)$

Step 2:
Unroll outern loop and *fuse* in the inner loop; there is potential for *vectorisation*.

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

→

```
for(int i=0; i < N; i += m)
   for(j = 0; j < N; j++){
      b_temp  = B[j];
      C[i]   += A[i][j] * b_temp;
      C[i+1] += A[i+1][j] * b_temp;
      …
      C[i+m] += A[i+m][j] * b_temp; }
```

N×N/m

N×N

N

→ $N^2 \times (1+1/m) + N$

Note: unrolling and register spill

Using a too large m in the previous example while the target CPU does not have enough registers to keep all the needed operands results in a "code bloating".
In this case, the CPU has to spill registers' content to cache and viceversa, slowing down the computation.

→ learn to inspect the compiler's *log*

A too much involute and obscure loop body may hamper the compiler to effectively perform *unroll & jam* optimizations targeted to the CPU it runs on.

→ hand code effort to clarify the code
→ hints / directives to the compiler
(directives are generally not portable across different compilers)

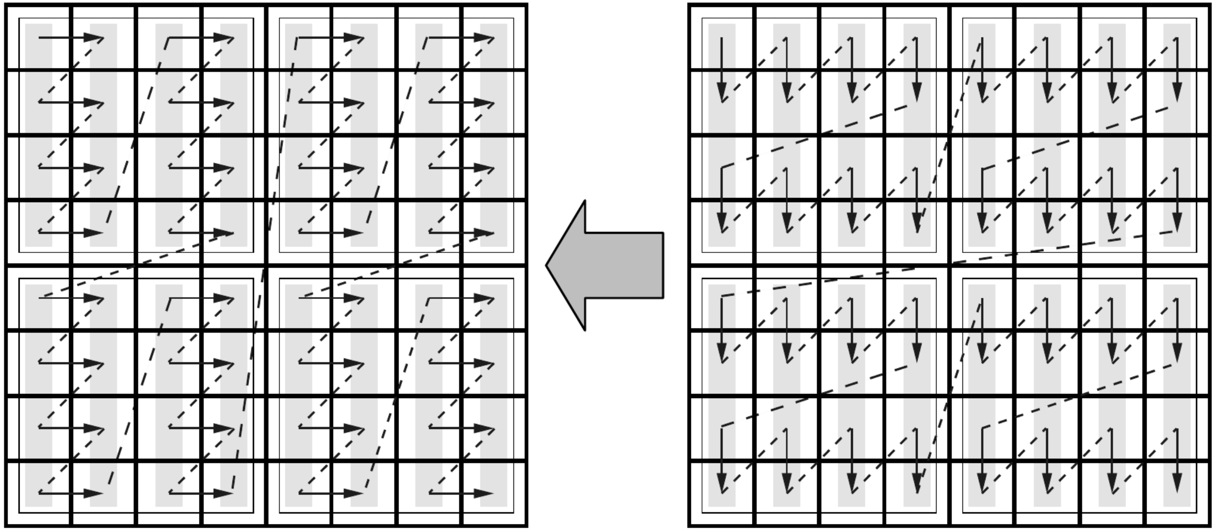
Cache access in loops: $O(N^2)/O(N^2)$

Sometimes no magic wand can cure the fact that you have to access N^2 memory locations.
For instance: in matrix transpose you have to access all the source matrix and all the destination matrix once.

Unroll & Jam strategy can bring benefits as long as the cache can hold N lines.
An L_C-way unrolling is too much aggressive and may easily result in register pressure.

Loop blocking is a good strategy that does not save memory loads but increase dramatically the cache hit ratio

Cache access in loops: $O(N^2)/O(N^2)$



Step 3:
Fully exploit locality of referenced data; cut TLB misses by accessing 2D arrays by blocks

Image taken from: introduction to HPC for scientists and engineers

Loop unrolling

Loop unrolling is a fundamental code transformation which usually helps significantly in improving your code performance:

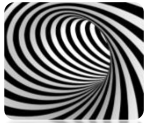
- It reduces the loop overhead (counter update, branching)
- It exposes *critical data path* and dependencies
- It helps in exploiting ILP, especially in case of memory aliasing

Cache access in loops: $O(N^3)/O(N^2)$

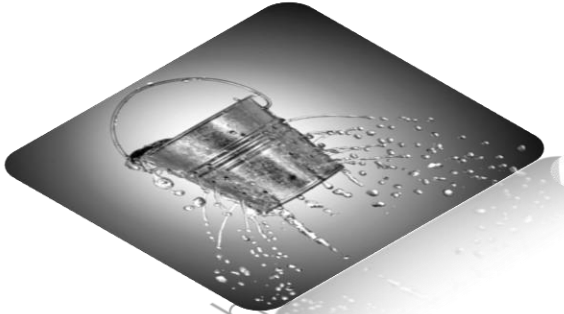
These algorithms (ex: matrix-matrix multiplication or dense matrix diagonalization) are very good candidates for optimizations that lead flop/s performance very close to the theoretical peak (in fact, MMM is at the core of `linpack`).

Blocking, unroll&jam + vectorization of operations, reorganization of ops to exploit CPU's pipelines and out-of-order capability, are all used by extremely specialized libraries.

→ It is a brilliant idea to link those library instead of developing your own algorithm, unless some very special needs must be met.



Outline



Avoid the avoidable inefficiencies

Loops techniques

Prefetching

At the right moment, at the right place

We know that waiting for data and instructions is a major performance killer.

Modern CPUs have the capability of pre-emptively bring from memory into cache levels data that **will be needed shortly afterwards**.
They can do that following some speculative algorithm based on the current execution flow and assuming spatial locality and temporal locality.

Both *data* and *instructions* can be pre-fetched.

Pre-fetching may be both hardware-based and software-based (tipically the compiler insert pre-fetching instructions at compile-time).

At the right moment, at the right place

From the point of view of the programmer, there are 2 possible ways to deal with prefetching:

EXPLICIT

you explicitly insert a pre-fetching directive.
Very difficult to be achieved effectively: the directive must be inserted timely but not too early (data eviction) or too late (load latency).

INDUCED

you consciously arrange data layout and execution flow so that to make it obvious to the compiler what to prefetch.

Explicit prefetching

This is a standard binary search implementation.

Find the median element

Define the next search

```
int mybsearch(int *data, int N, int Key)
 {
   int register low = 0;
   int register high = N;
   int register mid;

   while(low <= high) {
     mid = (low + high) / 2;

     if(data[mid] < Key)
       low = mid + 1;
     else if(data[mid] > Key)
       high = mid-1;
     else
       return mid;
   }
   return -1;
 }
```

Explicit prefetching

We can make it better by simply making sure that the element to be compared for (the `mid`) is in the cache when requested

```
int mybsearch(int *data, int N, int Key)
 {
   int register low = 0;
   int register high = N;
   int register mid;

   while(low <= high) {
     mid = (low + high) / 2;

     if(data[mid] < Key)
       low = mid + 1;
     else if(data[mid] > Key)
       high = mid-1;
     else
       return mid;
   }
   return -1;
 }
```

Explicit prefetching

We can make it better by simply making sure that the element to be compared for (the `mid`) is in the cache when requested

```
int mybsearch(int *data, int N, int Key)
 {
   int register low = 0;
   int register high = N;
   int register mid;

   while(low <= high) {
     mid = (low + high) / 2;
      __builtin_prefetch (&data[(mid + 1 + high)/2], 0, 3);
      __builtin_prefetch (&data[(low + mid - 1)/2], 0, 3);

     if(data[mid] < Key)
       low = mid + 1;
     else if(data[mid] > Key)
       high = mid-1;
     else
       return mid;
   }   return -1; }
```

Explicit prefetching

```
luca@GGG:~/code/HPC_LECTURES/prefetching% ./prefetching_off
performing 13421772 lookups on 134217728 data..
set-up data.. set-up lookups..
start cycle.. time elapsed: 20.7534
luca@GGG:~/code/HPC_LECTURES/prefetching% ./prefetching_on
performing 13421772 lookups on 134217728 data with prefetching enabled..
set-up data.. set-up lookups..
start cycle.. time elapsed: 12.6204
```

Explicit prefetching

```
Samples: 71K of event 'cpu/mem-loads,ldlat=30/P', Event count (approx.): 13901140
Overhead       Samples  Memory access
  71,08%         42196  Local RAM hit
  24,14%         17022  LFB hit
   4,11%         10967  L3 hit
   0,63%          1714  L1 hit
   0,02%            75  L2 hit
   0,01%            15  L3 miss
   0,00%             1  Uncached hit
```

```
Samples: 61K of event 'cpu/mem-loads,ldlat=30/P', Event count (approx.): 11720387
Overhead       Samples  Memory access
  68,74%         29450  LFB hit
  27,04%         28208  L1 hit
   2,72%           909  Local RAM hit
   1,29%          2983  L3 hit
   0,20%           346  L2 hit
```

Explicit prefetching

Usage of direct prefetching directive is highly uncertain, since it is difficult to spot the exact point – both in the code and in the execution – where to place them (also because your C code is different than the generated assembly code).

Moreover, the “exact point” is very likely dependent on the system you run on, and then it is susceptible to change significantly.

It is normally much safer to re-organize your code so to have **prefetching by pre-loading**.

Prefetching by moral suasion

Let’s discuss together this very simple example before putting the hands on the code you find in the `git`

```
elem a = elements[0]
for ( i = 0; i < 4*N_4; i+= 4 )
 {
        elem e = elem[i+4]; // non-blocking miss
        elem b = elem[i+1]; // possible cache-hit
        elem c = elem[i+2]; // possible cache-hit
        elem d = elem[i+3]; // possible cache-hit
        Elaborate(a);
        Elaborate(b);
        Elaborate(c);
        Elaborate(d);
        a = e;
 }
```

Hands-on

You find code snippets with different flavours of prefetching-by-preloading technique on our GitHub, with some comments about compilation.

Compile and run them with different options (and possibly different compilers) and try to understand what happens on your laptop and/or on HPC facility.

```
for ( i = 0; i < N; i++ )
    sum += array[ i ];
```

that’s all, have fun