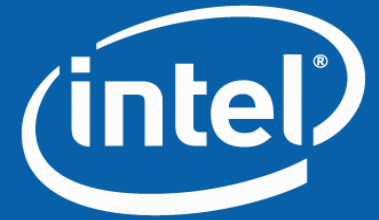
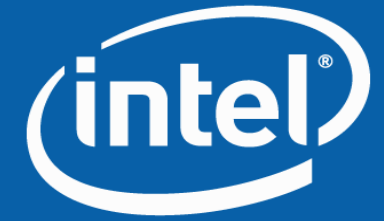


# emboss EBI / open source

Paul Guermonprez  
Sr. Software Engineer  
Software and Solutions Group  
17-06-2006



**Emboss package  
Today's algorithm  
Implementation  
Improvements  
Benchmarks**



# Emboss package

# Presentation

- Package of software for bioinformatics.
- Open Source package under GPL and LGPL.
- Widely used worldwide on servers and clusters.
- Used by pharmas, academics, biotechs ...
- Well maintained, highly portable, extensive documentation, libraries open for external use, various GUI projects.

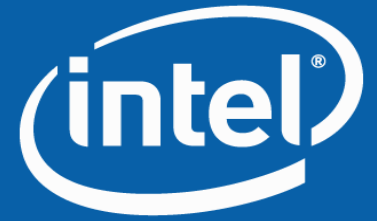


# Focus

- In this package, focusing on a smith–waterman algorithm implementation, « water ».
- This software is using a lot of ressources, both in terms of memory and cpu.
- This same algorithm is very important in bioinformatics and used in more complex implementations.

# Biological problem

- Biologists often have various versions of the same sequence, nearly identical.
- Differences are due to evolution of 2 genes between species, or in time, or between individuals, or between the same sequence but different sequencing operations.
- But this is not directly useful to find where a small sequence (think gene) is located in a big sequence (think genome), (can be part of the solution).



# Today's algorithm

# Hypothesis

- Take two small DNA/RNA/... sequences (size of a gene typically, a few K).
- The two sequences are nearly the same but have local mismatches and not exactly the same size (gaps due to insertions, deletions, ...).
- Goal : find best alignment between the two.
- Parameters : give variable penalty for gap opening, gap extension and mismatch.



# Overview

- Create two 2D matrices `compass` and `path` of dimension :  $\text{length}(\text{seq1}) * \text{length}(\text{seq2})$ .
- For each couple of position (« base ») in the two sequences, `path` will store a score and `compass` will store a direction (null, gap left or gap right).
- Let's launch a double loop foreach base in `seq1` and foreach base in `seq2`, we have  $\text{length}(\text{seq1}) * \text{length}(\text{seq2})$  iterations.

# Iteration – input/output

- We have to get 3 scores from `path`, and set a score and a direction for the iteration.
- The 3 scores are located  $(-1,-1)$ ,  $(-1,-2)$ ,  $(-2,-1)$  in the 2D `path` matrix, here is why :
- For each iteration we have to estimate if the best alignment is direct with no gap, with a gap in sequence 1 or a gap in sequence 2.

# Iteration – matrix accesses

Iteration n

...

Iteration n+1

	Seq1			
Seq2		Gap1		
	Gap2	NoGap		
			X,Y	

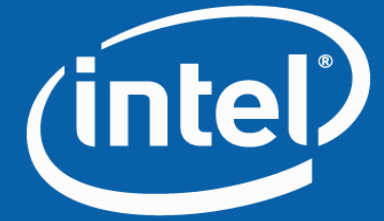
	Seq1			
Seq2				
		Gap1		
	Gap2	NoGap		
			X,Y+1	

# Iteration – 3 cases

- We've seen 3 cases : no gap, gap direction 1 and 2
- These cases are chosen comparing the score of the ungapped alignment (default) with the two gapped alignments.
- But with penalties taken into considerations.
- The selected score is stored in the `path` variable, and the selected case in the `compass` variable.

# Iteration – Border effects

- We need to access data in the `path` matrix from the previous iteration.
- But you can only access the  $(-2,-1)$  cell if you are far enough from the border.
- That's why the code has to create branches in the iteration to handle the case or create a separate loop to do so.



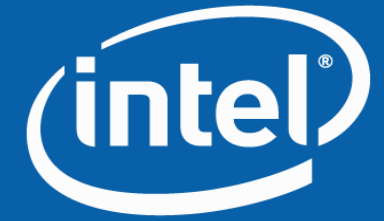
# Implementation

# Starting loops and default Case

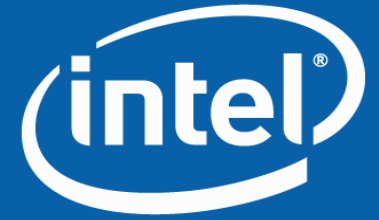
```
1 xpos = 1;
2 while(xpos!=lenb) ← FIRST LOOP
3 {
4     i = 1;
5     bx = maxb[xpos-1];
6     bv = 0;
7
8     while(i<lena) ← SECOND LOOP
9     {
10         /* get match for current xpos/ypos */
11         match = sub[ajSeqCvtK(cvt,a[i])][ajSeqCvtK(cvt,b[xpos])];
12
13         /* Get diag score */
14         mscore = path[(i-1)*lenb+xpos-1] + match; ← Setting default
15                                                     score
16
17         /* ajDebug("\tOpt %d %6.2f \",i*lenb+xpos,mscore); */
18
19         /* Set compass to diagonal value 0 */
20         compass[i*lenb+xpos] = 0;
21         path[i*lenb+xpos] = mscore;
```

# Test Case 1 : gap along X ?

```
23  /* Now parade back along X axis */
24  if(xpos-2>-1) ← border effect ?
25  {
26      fnew=path[(i-1)*lenb+xpos-2]; ← get score for
27      fnew-=gapopen;                (-1,-2)
28      if(maxa[i-1] < fnew)
29      {
30          oval[i-1] = maxa[i-1] = fnew;
31          cnt[i-1] = 0;
32      }
33      ++cnt[i-1];
34
35      if( maxa[i-1]+match > mscore) ← is score different
36      {                               enough from
37          mscore = maxa[i-1]+match;   default case ?
38          path[i*lenb+xpos] = mscore;
39          compass[i*lenb+xpos] = 1; /* Score from left */
40      }
41
42  }
```



# Improvements



# Improvements Memory Access

# Why accessing memory is bad ?

Matrices are big memory blocks, and we need to access the memory :

- aligned with the hardware's layout
- a minimum of times
- highest level (HardDrive<FBD/DDR<L3<L2<L1<...)
- helping the hardware and software prefetchers

# Memory – alignment

Matrices are big memory blocks, often read, so we need to keep in mind the hardware layout and the language memory usage.

This code is accessing memory : `path[i*len+xpos]`, `i` as an inner loop.

Inverting the loops helps because we load hardware-contiguous memory locations.

But languages differs in their memory management : in fortran, it would the right way to access matrices.

(as we are changing the loops, we write them with *for* instead of *while* to help the compiler)

# Memory – alignment

Vertical – foreach seq1, foreach seq2

n \	Seq1			
Seq2		Gap1		
	Gap2	NoGap		
			X,Y	

n+1 \	Seq1			
Seq2				
		Gap1		
	Gap2	NoGap		
				X,Y+1

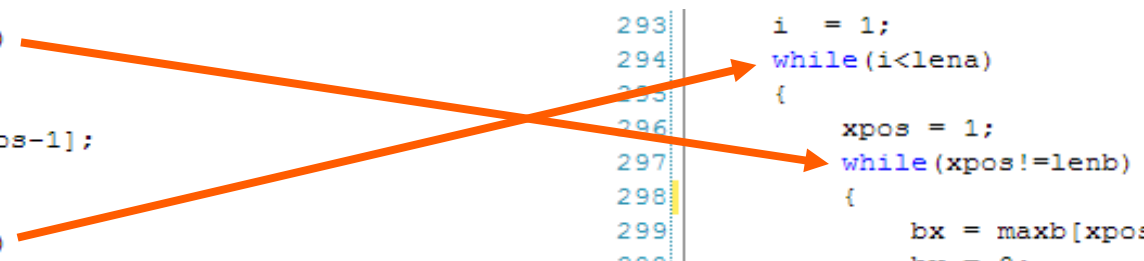
Horizontal – foreach seq2, foreach seq1

n \	Seq1			
Seq2		Gap1		
	Gap2	NoGap		
			X,Y	

n+1 \	Seq1			
Seq2			Gap1	
		Gap2	NoGap	
				X,Y+1

# Inverting loops

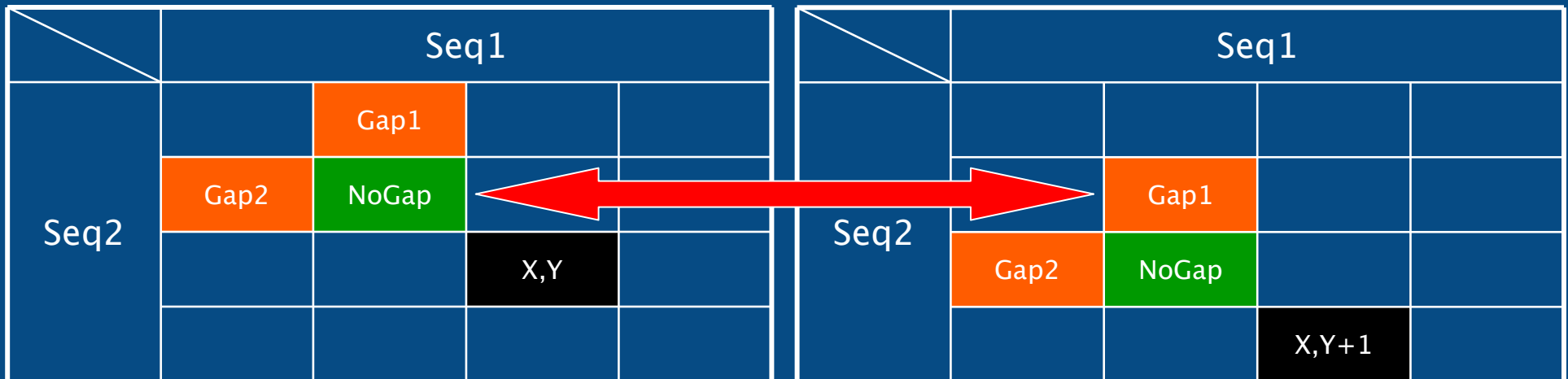
```
293 | xpos = 1;
294 | while(xpos!=lenb)
295 | {
296 |     i = 1;
297 |     bx = maxb[xpos-1];
298 |     bv = 0;
299 |
300 |     while(i<lena)
301 |     {
293 |     i = 1;
294 |     while(i<lena)
295 |     {
296 |         xpos = 1;
297 |         while(xpos!=lenb)
298 |         {
299 |             bx = maxb[xpos-1];
300 |             bv = 0;
```



# Memory – number of accesses

We need to find way to reuse memory accesses, between iterations for example

Same memory access from 2 consecutive iterations  
Iteration n                      ...                      Iteration n+1



# Memory – number of accesses

```
466 | palast = path[(i-1)*lenb];  
467 | for ( xpos=2 ; xpos<lenb; xpos+=1 )  
468 | {  
469 |     ajint co = compass[i*lenb+xpos] ;  
470 |     float pa00 = path[i*lenb+xpos] ;  
471 |     float pa11 = path[(i-1)*lenb+xpos-1] ;  
472 |     float pa12 = palast ;  
473 |     // float pa12 = path[(i-1)*lenb+xpos-2] ;  
474 |     float pa21 = path[(i-2)*lenb+xpos-1] ;  
475 |     palast = pa11 ;
```

Temporary  
variable access  
from iterations

# Remove array accesses

```
456     int subai = suba[i] ;
457     float ma = maxa[i-1] ;
458     float cn = cnt[i-1] ;
459     float ov = oval[i-1] ;
460
461     for (xpos=2;xpos<lenb;xpos++)
462     {
463         subsub[xpos] = sub[ subai ][ subb[xpos] ] ;
464     }
465
466     for ( xpos=2 ; xpos<lenb; xpos++ )
467     {
468         bx = maxb[xpos-1];
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529     maxa[i-1] = ma ;
530     cnt[i-1] = cn ;
531     oval[i-1] = ov ;
```

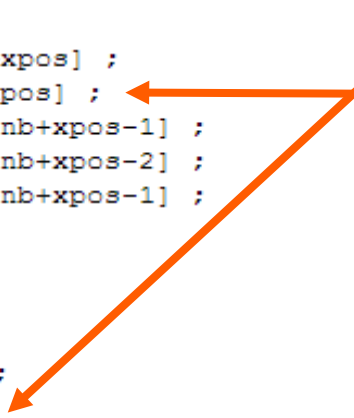
Temporary  
variable instead  
of array

# Remove matrix accesses

```
468 |         ajint co = compass[i*lenb+xpos] ;  
469 |         float pa00 = path[i*lenb+xpos] ;  
470 |         float pa11 = path[(i-1)*lenb+xpos-1] ;  
471 |         float pa12 = path[(i-1)*lenb+xpos-2] ;  
472 |         float pa21 = path[(i-2)*lenb+xpos-1] ;
```

```
538 |         compass[i*lenb+xpos] = co ;  
539 |         path[i*lenb+xpos] = pa00 ;
```

Temporary variable instead of matrix, only one cell has to be written back to the matrix



# Memory – Highest Level

High memory levels have better access times, but are smaller. (The smaller the fastest). You usually take care not to use structures bigger than your main memory to avoid « swapping ».

The same problem happens between memory levels, and with the same relative impact on speed.

Solution : To avoid swapping, you've learnt to use series of independant structures loaded one at a time from hard drive to memory. Just do the same, recursively, to use efficiently the various levels of memory. Between memory and L3, L3 and L2, L2 and L1 ... or to keep it simple from memory to L1.



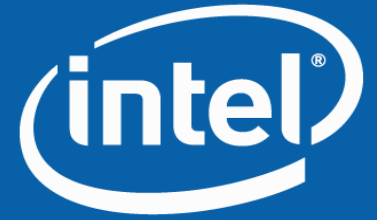
# Memory – Prefetching

Of course higher caches are not only used if you have small structures.

The processor and the compiler are working together to help minimise accesses to lower cache by fetching some data to higher caches and register automatically.

The criterias to do so may vary : contiguous data in memory, usage patterns discovery during execution, ...

Not enough ? You can even code in your software (using intrinsics) what data are about to be accessed to let the processor prefetch this dat.



# Improvements Branches

# Why branches are bad ?

- When you code conditionnal statements with alternative executions, the processor is using branches to do some calculations until he is sure this is the right direction.
- But if wrong, the processor has to unload all the useless information from his pipeline and reload the right data. And wait.
- Conclusion : Avoid contitionnal statements when possible.

# Removing branches

- In this code, we've seen that for each iteration, we are checking if we are far enough from the border to access  $(-1,-2)$  and  $(-2,-1)$  cells.
- Most time the test is true, and this is easy to predict.
- A solution : change the inner loop to start iterations at 2 instead of 1, the test will always be true and can be removed. And create two copies of the loops to handle the first iteration for the two dimensions.

# Removing branches

*Before*

...

*After*

One big double loop  
with a lot of false branches  
(yellow, orange and red)  
 $len1 * len2$  iterations

1	2	3	4	5
2	2	3	4	5
3	3			
4	4			
5	5			

One big double loop  
plus two simple loops  
no branches at all  
 $len1 * len2 - 1$  iterations

2	3	4	5
2	3	4	5
3			
4			
5			

# Removing branches

- You had  $\text{len1} * \text{len2} * 2$  branches in the original code.  $\text{len1} + \text{len2} - 1$  branches false, the rest true.
- In the second code, no branches left as the result of the test is known from the loop index.

# Split borders

```
293 | for ( i=1 ; i<lena ; i++ )  
294 | {  
295 |     for ( xpos=1 ; xpos<lenb; xpos++ )  
296 |     {
```

```
441 |  
442 |  
443 |  
444 |
```

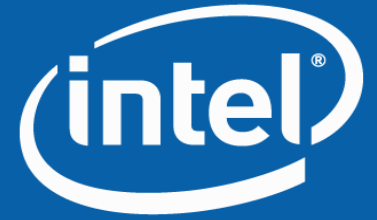
```
for ( i=2 ; i<lena ; i++ )  
{  
    for ( xpos=2 ; xpos<lenb; xpos++ )  
    {
```

```
368 |  
369 |  
370 |  
371 |
```

```
for ( i=1 ; i<lena ; i++ )  
{  
    for ( xpos=1 ; xpos<2; xpos++ )  
    {
```

```
295 |  
296 |  
297 |  
298 |
```

```
for ( i=1 ; i<2 ; i++ )  
{  
    for ( xpos=1 ; xpos<lenb; xpos++ )  
    {
```



# Improvements CPU

# Function calls

- Function calls are not only time consuming, but also prevent the compiler from applying vectorisation to your loops.
- The Intel Compiler can optimise this with IPO (interprocedural optimisation) and inlining (replacing the function call by the content of the function).
- But the best method is to avoid function calls.

# Function calls

You can see the functions in the main iteration being called  $\text{len1} * \text{len2} * 2$  times.

But the two function calls could be called  $\text{len1} + \text{len2}$  times only, and outside the main loop to help vectorisation, results are stored in small ( $\text{len1}$  and  $\text{len2}$ ) arrays.

# Function calls

One more thing can be done : we are now calling a 2D array in the main iteration and the compiler is not happy with it (see logs).

A solution would be to create a temporary 1D array for each iteration of the outer loop, and copy the data from one line of the 2D array in the 1D array. The inner loop could then access a simple 1D array.

But we are hackers : why not use a secondary pointer to a line of the 2D array ? because it would create 2 pointers to the same memory location, we'll cover that later.


# Function calls

```
302 |         /* get match for current xpos/ypos */
303 |         match = sub[ajSeqCvtK(cvt,a[i])[ajSeqCvtK(cvt,b[xpos])]];

440 |     int *suba = ( int* )calloc( lena , sizeof(int) );
441 |     int *subb = ( int* )calloc( lenb , sizeof(int) );
442 |     float *subsub = ( float* )calloc( lenb , sizeof(float) );
443 |     for ( i=2 ; i<lena ; i++ )
444 |     {
445 |         suba[i] = ajSeqCvtK(cvt,a[i]) ;
446 |     }
447 |     for ( i=2 ; i<lenb ; i++ )
448 |     {
449 |         subb[i] = ajSeqCvtK(cvt,b[i]) ;
450 |     }

456 |     int subai = suba[i] ;
457 |
458 |     for (xpos=2;xpos<lenb;xpos++)
459 |     {
460 |         subsub[xpos] = sub[ subai ][ subb[xpos] ] ;
461 |     }

470 |     match = subsub[xpos];
```



# Type conversions


- Conversions between type are costly operations in term of CPU.
- We usually use smaller types to save memory, but is it optimal ?

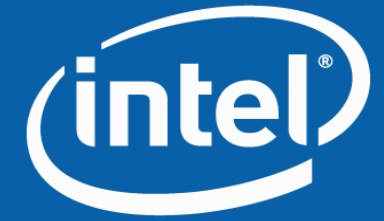
# Type conversions

```
518 |         maxa[i-1]= oval[i-1] - ((float)cnt[i-1]*gapextend);
519 |         maxb[xpos-1]= bx - ((float)bv*gapextend);

251 |     float *cnt;
252 |
253 |     static AjPStr outstr = NULL;
254 |     float bx;
255 |     float bv;
256 |
257 |     ajDebug("embAlignPathCalcSW\n");
258 |
259 |     /* Create stores for the maximum values in a row or column */
260 |
261 |     maxa = AJALLOC(lena*sizeof(float));
262 |     maxb = AJALLOC(lenb*sizeof(float));
263 |     oval = AJALLOC(lena*sizeof(float));
264 |     cnt = AJALLOC(lena*sizeof(float));

430 |         maxa[i-1]= oval[i-1] - (cnt[i-1]*gapextend);
431 |         maxb[xpos-1]= bx - (bv*gapextend);
```





# Benchmarks

# Benchmark – Workload

2 sequences of the Lake Victoria Marburg Virus have been compared (16 kba, different strains) from local fasta files with default water parameters (gapopen=10 gapextend=0.5), 20 runs per measure :

>gi|91177767|gb|DQ447652.1

*Lake Victoria marburgvirus*

*DRC1999 strain 09DRC99, complete genome*

>gi|91177759|gb|DQ447651.1

*Lake Victoria marburgvirus*

*DRC1999 strain 05DRC99, complete genome*



# Benchmark – Platforms

2 EM64T platforms have been tested, both with SuSE 10.1 64 bits to have the last GCC 4.1.0 rpm :

**Giraglia** : **Nocona** HP Workstation xw8200

(NOCONA cpufamily=15 model=4)

1 Xeon 3.6 Ghz, 1Mo cache

4Go RAM ddr2 400Mhz, SATA disk.

**Capraia** : **Woodcrest** Intel Super Micro server

(WOODCREST cpufamily=6 model=15)

2 Xeon Dual Core 2.66 Ghz, 4Mo cache

4Go RAM ddr2 667MHz, SATA disk. Pre production SDV.



# Benchmark – Compilers

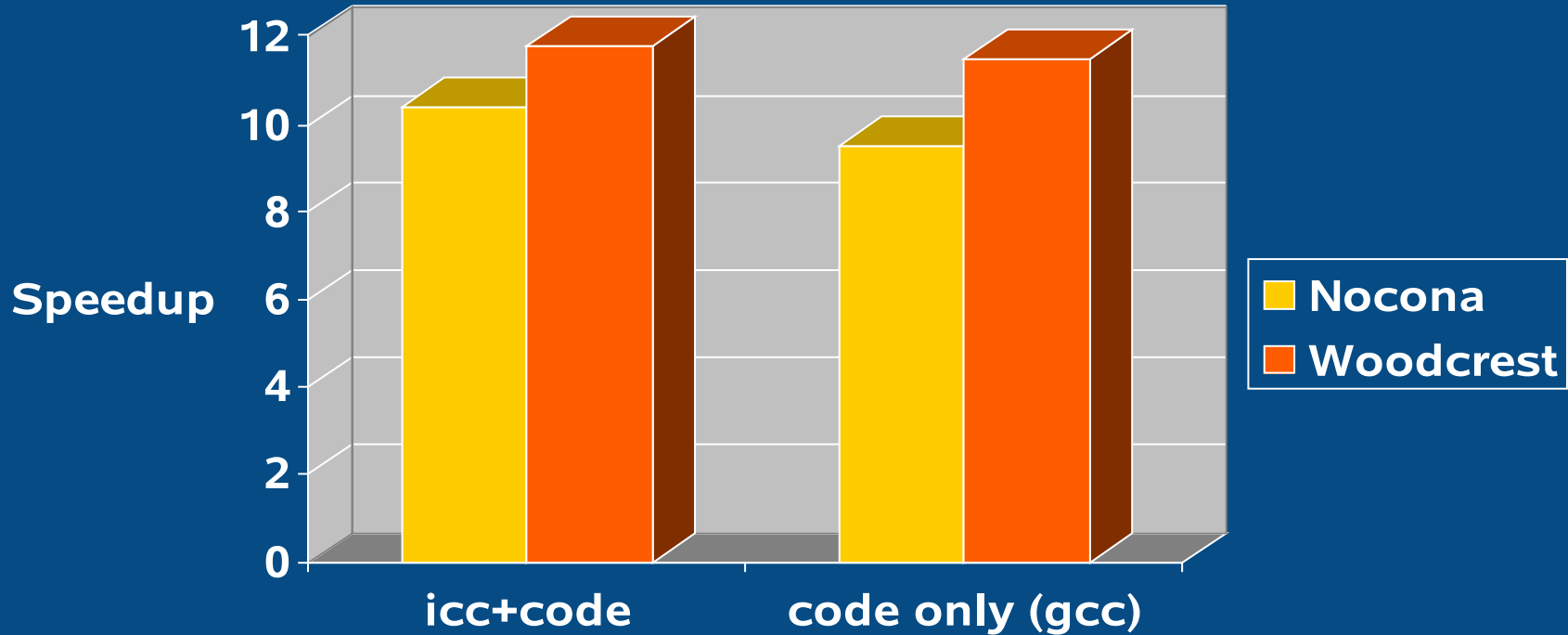
The goal is not to compare compilers, but to verify the optimization process with gcc and intel compiler. The two compilers are :

gcc : 4.1.0 rpm from SuSE 10.1,  
*compiled with -O2 or -O3 depending on tests.*

icc : Intel Compiler 9.1.039 for EM64T,  
*with various compiler switches.*



# Optimization Speedups Complete (compiler+code) vs. Code Only



Get the maximum speedup from your optimization with the new Xeon Woodcrest and Intel Compiler 9.1

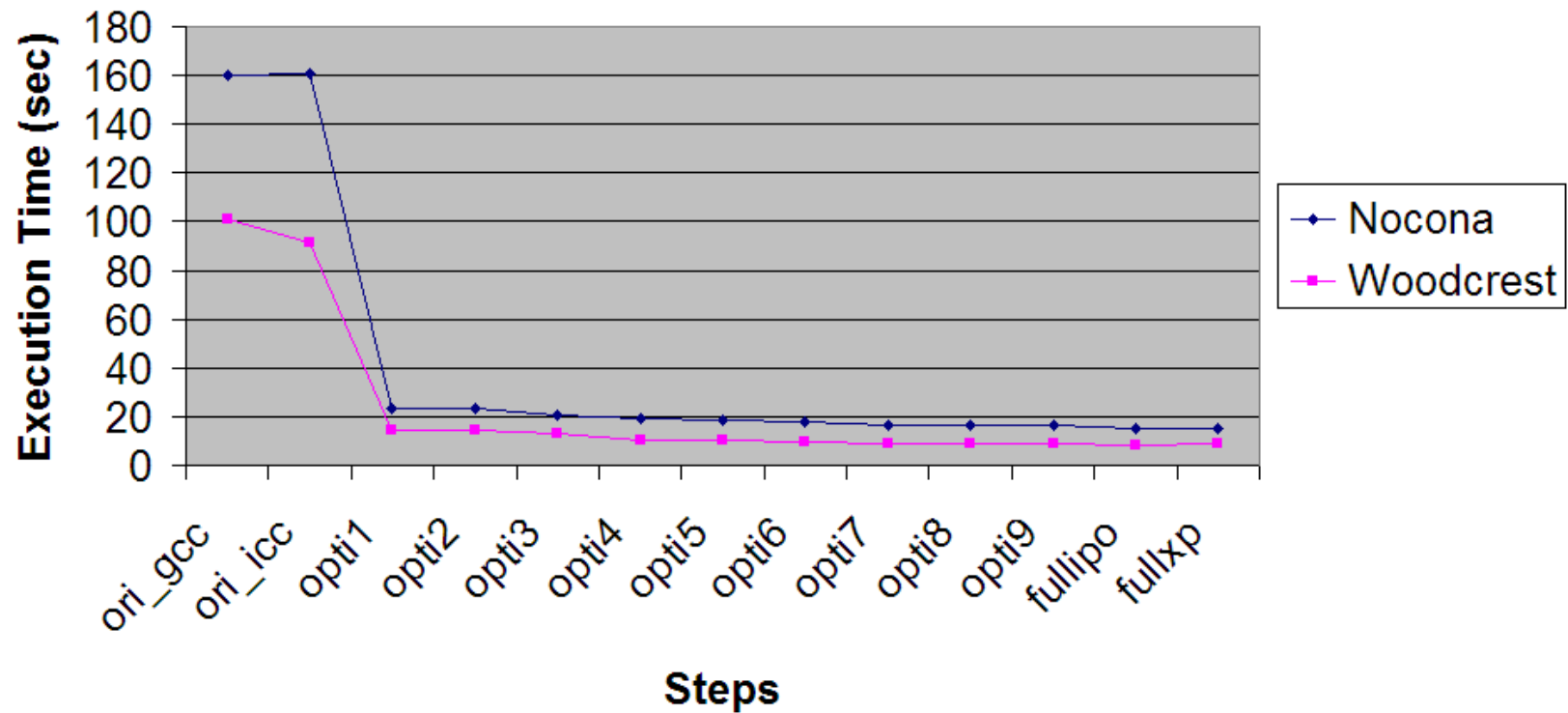


# Optimization Speedups Complete (SW+ICC) and Software Only

	Complete Speedup (Code + ICC)	Software Speedup (gcc -O3)
Nocona	10,4	9,5
Woodcrest	11,8	11,5

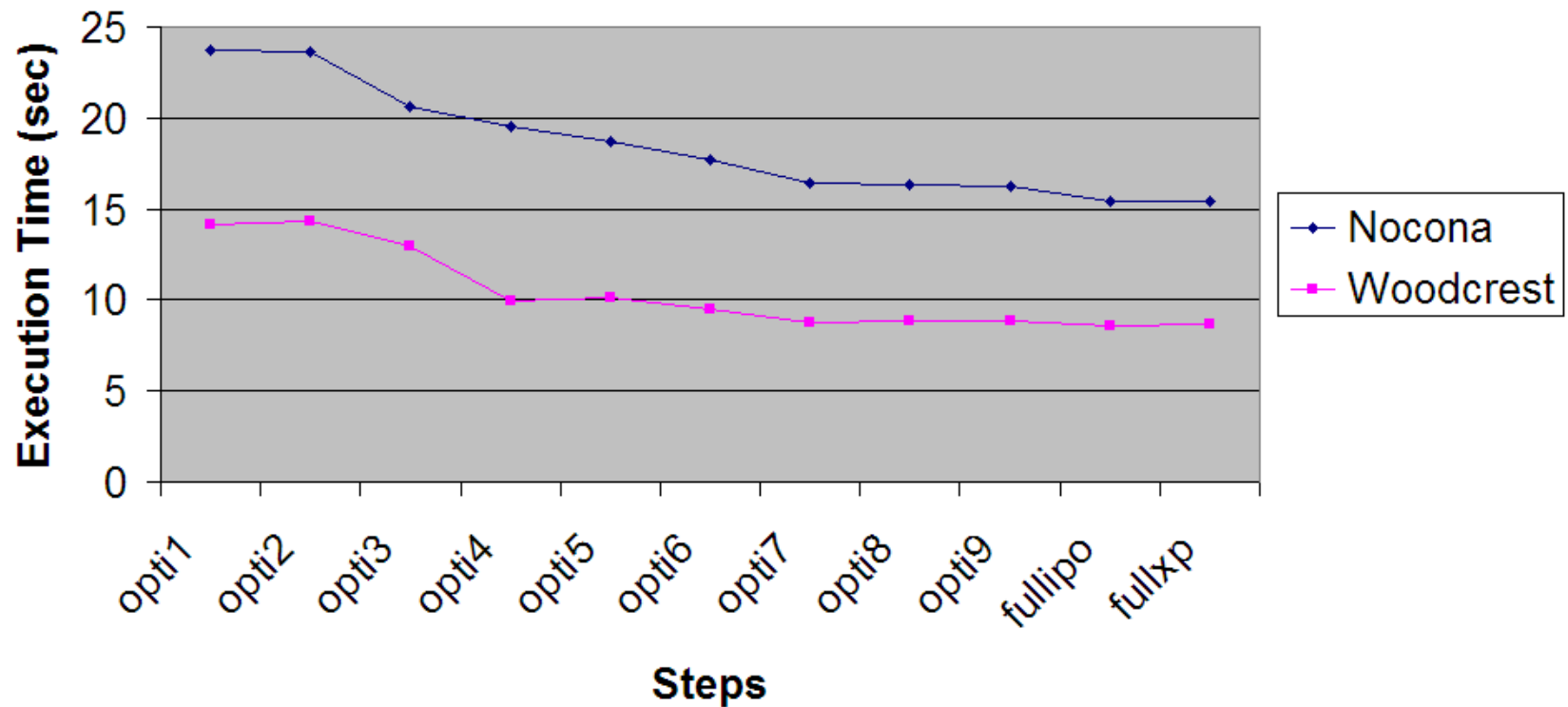
# Complete optimization

Complete optimization process  
(marburg workload)

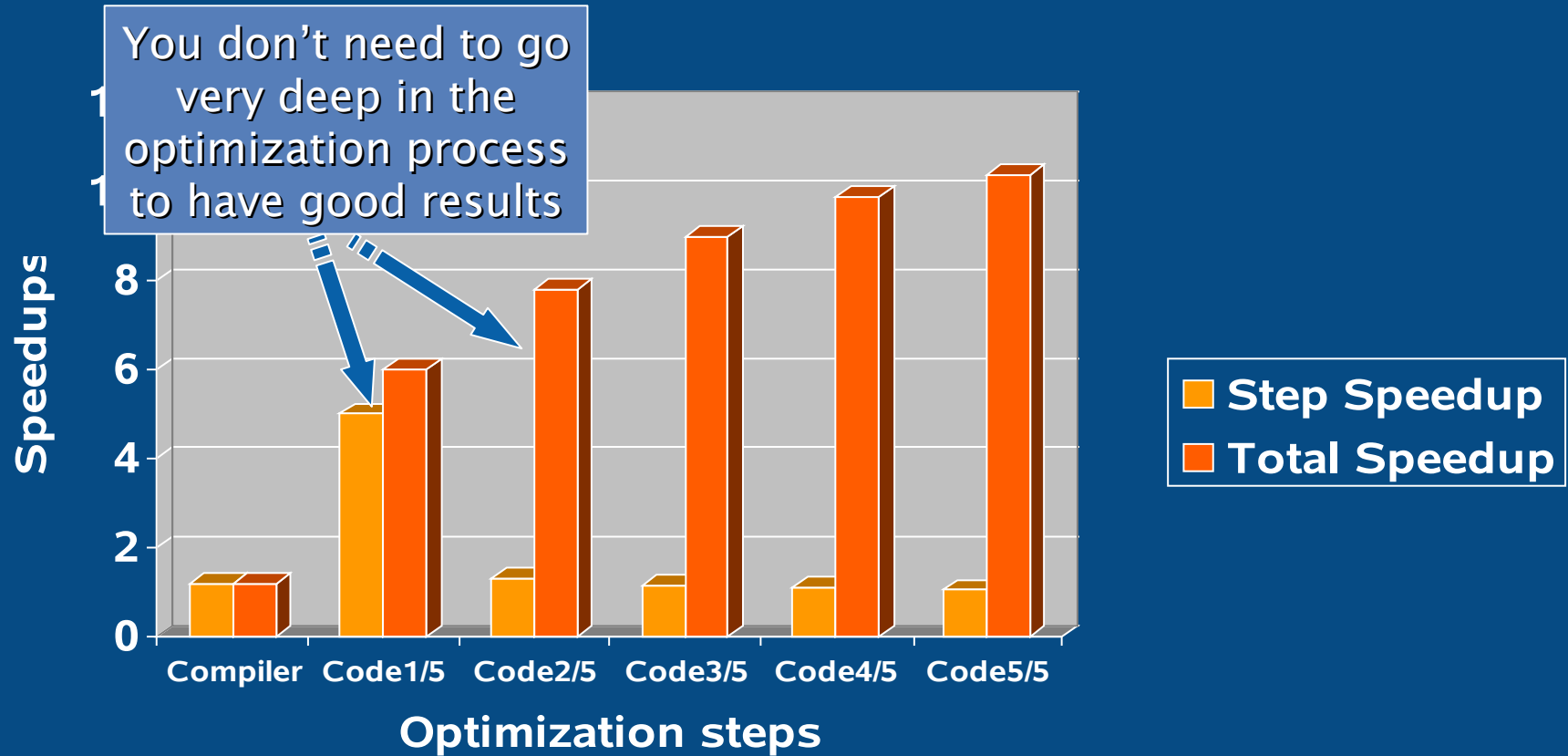


# Focus on the last steps

## Focus on the last steps of optimization (Marburg workload)



# Example Effort vs. Speedup



# Results – Nocona

## 20 runs per configuration

Giraglia-NCN	avg	min	max	std
original_gcc	<b>160,36</b>	158,37	161,14	0,593
original_icc	160,56	158,89	161,3	0,599
opti-01_icc	23,7	23,21	24,85	0,636
opti-02_icc	23,66	23,2	24,8	0,652
opti-03_icc	20,65	20,63	20,68	0,014
opti-04_icc	19,55	19,31	19,84	0,206
opti-05_icc	18,66	18,48	18,87	0,104
opti-06_icc	17,71	17,62	17,8	0,045
opti-07_icc	16,44	16,36	16,49	0,033
opti-08_icc	16,34	16,29	16,43	0,036
opti-09_icc	16,28	16,21	16,36	0,04
opti-09_icc_fullipo	15,4	15,35	15,5	0,036
opti-09_icc_fullxp	<b>15,38</b>	15,32	15,42	0,03
opti-09_gcc_o3	16,83	16,77	16,92	0,038



# Results – Woodcrest

## 20 runs per configuration

Capraia-WDC	avg	min	max	std
original_gcc	<b>101,07</b>	100,5	101,67	0,29
original_icc	91,4	90,91	91,8	0,222
opti-01_icc	14,14	14,12	14,2	0,019
opti-02_icc	14,28	14,12	16,76	0,569
opti-03_icc	12,95	12,9	13,32	0,087
opti-04_icc	9,98	9,97	9,99	0,007
opti-05_icc	10,14	10,12	10,16	0,011
opti-06_icc	9,49	9,48	9,51	0,009
opti-07_icc	8,76	8,75	8,78	0,008
opti-08_icc	8,88	8,86	8,9	0,01
opti-09_icc	8,87	8,85	8,88	0,009
opti-09_icc_fullipo	8,58	8,57	8,6	0,009
opti-09_icc_fullxp	<b>8,65</b>	8,64	8,67	0,008
opti-09_gcc_o3	8,84	8,83	8,86	0,009



# Optimization steps

see source codes for details

original_gcc	Original code compiled with <code>gcc -O2</code>
original_icc	Original code compiled with intel <code>icc -O2</code>
opti-01_icc	Inverting <code>loops</code> , <code>icc -O3</code>
opti-02_icc	Changing while with <code>for</code> loops, <code>icc -O3</code>
opti-03_icc	Moving <code>borders</code> in separate loops, <code>icc -O3</code>
opti-04_icc	Moving <code>function calls</code> in a separate loop, <code>icc -O3</code>
opti-05_icc	Avoiding <code>conversions</code> , <code>icc -O3</code>
opti-06_icc	Using temporary <code>variables for arrays</code> , <code>icc -O3</code>
opti-07_icc	Using temporary <code>variables for matrices</code> , <code>icc -O3</code>
opti-08_icc	Avoiding <code>matrix access</code> , <code>icc -O3</code>
opti-09_icc	Using a temporary variables for <code>maxb</code> array, <code>icc -O3</code>
opti-09_icc_fullipo	Opti-09, plus <code>compiler switches and -ipo</code>
opti-09_icc_fullxp	Opti-09, plus <code>compiler switches and -xP</code>
opti-09_gcc_o3	Opti-09, compiled with <code>gcc -O3</code>