Performance Engineering

Bruno Cuevas

21st Nov 2016

1 Introduction

In this report, the final goal is to enhance the performance of a C program. This program uses the Laplace approach to solve a problem through the use of a 5-points gradient.

2 Modifying loops

One of the keys aspects that must be taken into account is that matrixes in C are stored in a memory which is linear (so data is actually organised by rows that are placed one after each other). Since processors must move through the memory directions to find the data instance they are going to process, the process can get speed up only by changing the way used to access the memory values of the matrix.

In the original code, the matrix was accessed firstly by columns:

```
float A[n][m];
float Anew[n][m];
float y[n];
...
for( i=1; i < m-1; i++ ) for( j=1; j < n-1; j++){
  Anew[j][i]=(A[j][i+1]+A[j][i-1]+A[j-1][i]+A[j+1][i])/4;
  error=fmaxf(error,sqrtf(fabsf( Anew[j][i]-A[j][i])));

}
```

A change in the loop order to access firstly by rows was performed:

```
for( j=1; j < n-1; j++) for( i=1; i < m-1; i++ ){
  Anew[j][i]=(A[j][i+1]+A[j][i-1]+A[j-1][i]+A[j+1][i])/4;
  error=fmaxf(error,sqrtf(fabsf( Anew[j][i]-A[j][i])));
}
```

Although this could seem trivial, the speed up was actually pretty high, as it will be observed in the perf measures section.

3 Calculation changes

Computers perform operations in a different way than us. For instance, they do not perform division as we do but employing a method that only requires the fundamental operations that computers perform.

A possible way to speed up the process is using multiplication by 0.25 instead of division by 4 within the Anew[j][i] calculation.

```
for( j=1; j < n-1; j++) for( i=1; i < m-1; i++ ){
  Anew[j][i] = ( A[j][i+1]+A[j][i-1]+A[j-1][i]+A[j+1][i])*0.25;
  error=fmaxf(error,sqrt(fabsf( Anew[j][i]-A[j][i])));
}
```

Moreover, it can be observed that $\sqrt{\|An_{j,i} - A_{j,i}\|}$ is being calculated within the loop. Although we need that the error is root square of the substraction values, it is known that the largest error will also have the highest square root, so it is not necesary to calculate the square root until flow gets out of the loop.

```
for( j=1; j < n-1; j++) for( i=1; i < m-1; i++ ){
  Anew[j][i] = ( A[j][i+1]+A[j][i-1]+A[j-1][i]+A[j+1][i])/4;
  error=fmaxf(error,fabsf( Anew[j][i]-A[j][i]));
...
error = sqrt(error);
}
```

4 Double Buffer

Until now, the process has consisted in calculating the values of $An_{i,j}$ from the values of $A_{i,j}$, and then update the A matrix as $A_{i,j} = An_{i,j}$. This process is time consuming since it needs to iterate all over the values of An matrix to update A.

A double buffer can avoid this loop without need to consume more resources. Instead of calculating the values of An and then copy those values to A, the program will calculate the values for A or An depending on the iteration number, using the values of the other matrix.

```
if(iter % 2 == 0){
 for( j=1; j < n-1; j++) for( i=1; i < m-1; i++ ){
   Anew[j][i] = ( A[j][i+1]+A[j][i-1]+A[j-1][i]+A[j+1][i])*0.25f;
   error = fmaxf( error,  fabsf( Anew[j][i]-A[j][i] )  );
 }
} else {
 for( j=1; j < n-1; j++) for( i=1; i < m-1; i++ ){
   A[j][i] = (
↪  Anew[j][i+1]+Anew[j][i-1]+Anew[j-1][i]+Anew[j+1][i])*0.25f;
   error = fmaxf( error,  fabsf( Anew[j][i]-A[j][i] )  );
 }
}
```

As we can see, when the iter value (which is the iteration value) is odd, values will be calculated for An from A, and when iter is even, values will be calculated for A from An.

5 Parallelization through OpenMP

By now, the program has been using only one thread of execution. The use of OpenMP libraries can help to enhance the performance through calcuating different An values at the same time. A parallel section was introduced.

```
#pragma omp parallel default(none)\
 shared(Anew,A) private(i,j) \
reduction(max:error){
 #pragma omp for nowait
  for( j=1; j < n-1; j++) for( i=1; i < m-1; i++ ){
    Anew[j][i] = ( A[j][i+1]+A[j][i-1]+A[j-1][i]+A[j+1][i])*0.25f;
    error = fmaxf( error,  fabsf( Anew[j][i]-A[j][i] )  );
  }
 #pragma omp barrier
 #pragma omp for nowait
  for( j=1; j < n-1; j++) for( i=1; i < m-1; i++ )
    A[j][i] = Anew[j][i];
}
```

There are two aspects that worth attention in this implementation:

- There is a reduction statement. It allows us to keep the error value updated to the maximum error within the calculation without the need of storing this value inside a vector that should be iterated to find maxima.
- The use of double buffer is depreciated for parallelization.

6 Compilation

The program can be compiled in different ways: using the default gcc compiler that is included in all normal Linux distros, or the icc compiler, property of Intel Corporation.

Booth versions will be tested to compare if they produce any enhance of the performance, using as code the fastest version.

As it could be seen later, the enhance might be due to a better usage of the memory. Most of the Miss Cycles depend on the lag of DRAM memory, and those compilers could have internal tools to administrate in a better way memory, so it can move from the DRAM to the processor Caché without having the process waiting.

7 Statistics

Name	Improvements
Exec1	None
Exec2	Inverted loops.
Exec3	Inverted loops, multiplication, square root out
Exec4	Double Buffer, inverted loops, multiplication, square root out
Exec5	Inverted loops, multiplication, square root out, openMP
Exec6	Inverted loops, multiplication, square root out, openMP, 0fast
Exec7	Inverted loops, mult, square root out, openMP, icc

8 Conclussion

Three conclussions can be made:

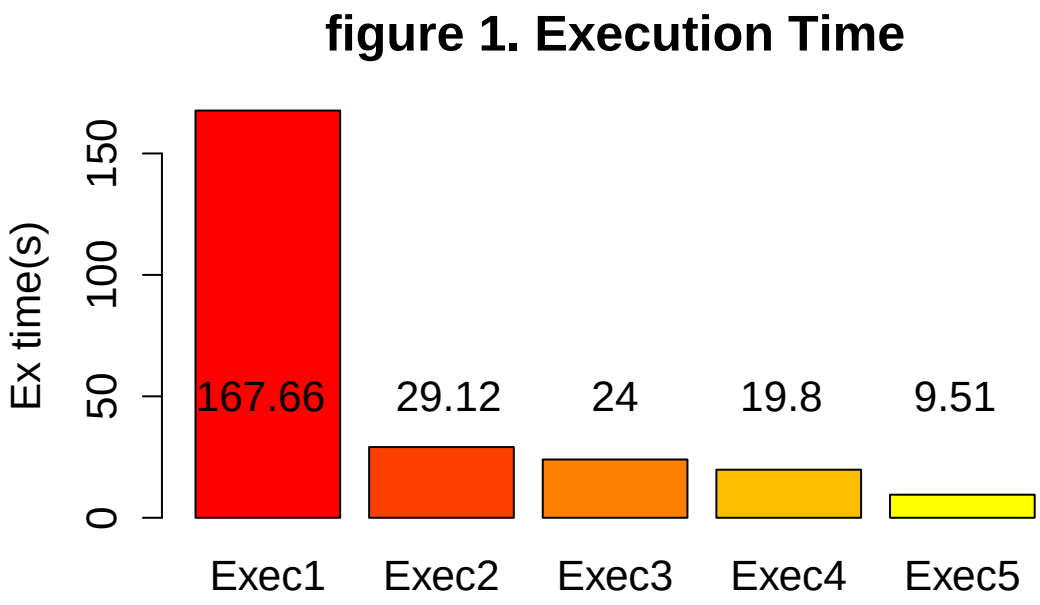

Figure 1: Execution time. Execution was performed in aolin21 at Monday 21st Night. Results for 10 measures. Since variance was extremely low, bar errors were depreciated.

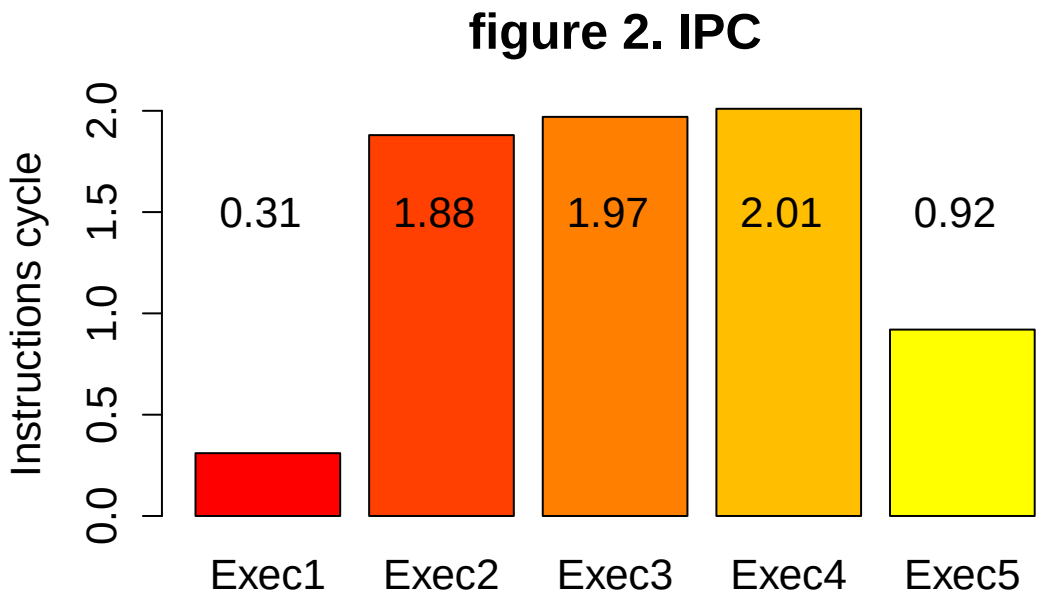

Figure 2: IPC (Intructions per cycle). Measured at the same time that data in figure 1.

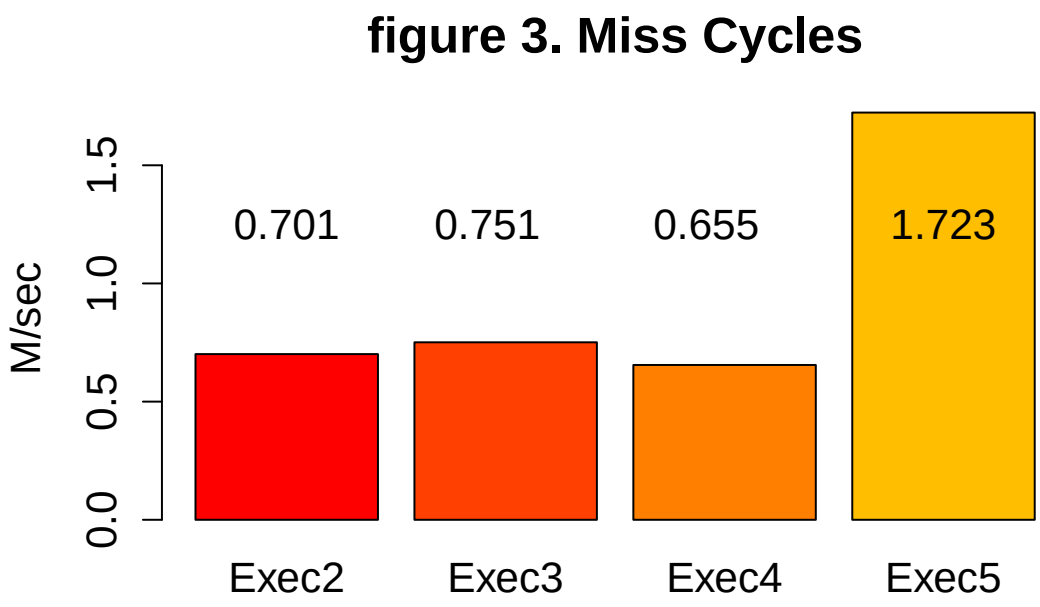

Figure 3: Miss Cycles. Measured at the same time that data in figure 2. Value of Exec1 was removed due to its much higher value in relation to all the other measures.

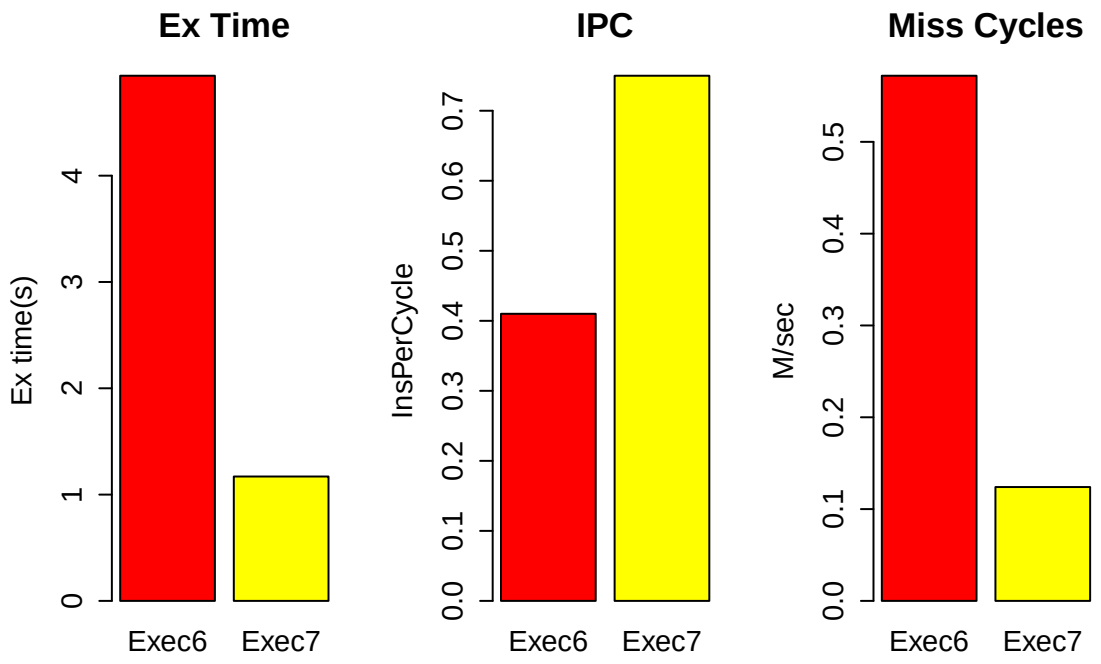

Figure 4: Execution time, IPC and Miss cycles. Booth Exec6 and Exec 7 have been compiled in different compilers, in a smaller grid (512x1024) and for an max iter of 1000.

- The use of parallelization tools, the control of the way we access to the memory directions of a matrix and the use of the right tools within the compiler can lead to an speed up of more than 500 times (data not shown).
- However, this speed up doesn't imply a better usage of resources. Double buffer implementation got the highest IPC and the lowest number of Missed Cycles, even if it was slower.
- Finally, the use of the compiler icc, with the option -O3 makes a larger speed up than gcc with the option -Ofast.