Performance Engineering: Assignment

Sofia Almirante (1527718), Pau Reig (1527816)

November 2022

Q1. Why is the strength reduction transformation providing no performance improvement?

Running the program with strength reduction transformation (concretely substituting division for multiplication, which are faster operations) presents no performance improvement. This is because we compile the base progam with -Ofast, which enables different kind of optimizations. One of the transformations implemented by the compiler is to provide faster, though sometimes less precise, mathematical operations, like substituting division for multiplication.

Q2. Why is the loop interchange transformation improving performance so much?

To answer this question we need to understand the basic idea of cache memory and how the data in our program is being stored. In the cache memory we store temporarily a copy of the data in some memory locations. This allows us to access very fast a small concrete amount of information. When the program asks for a memory location, it first checks the cache memory. If it is stored there, it doesn't have to access the main memory, so it results in a faster performance. On the contrary, when the location is not in the cache memory, it access the main memory and replaces the cache with the new information used. With this main idea, we can understand why working by blocks of information results in a better performance.

In Fig. 1, we have represented an scheme to intuitively understand this concept. In the Figure, we have stored a 2x8 matrix like a vector, and the cache memory is able to store the data of 5 spaces of memory. We can see that accessing the matrix by rows provides a better performance, being able to access five positions of the matrix with the same cache memory, and only replacing it every 5 iterations. In the other hand, when accessing the matrix by columns, it is necessary to replace the cache memory every iteration, so we are always accessing the main memory, without taking advantage of the cache memory properties.

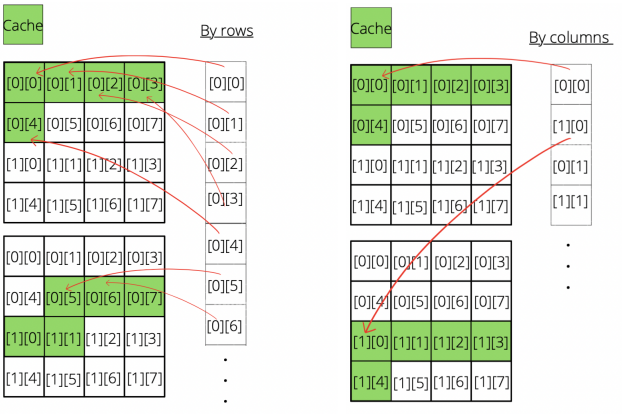

Figure 1: Cache memory scheme accessing by rows (left) and columns (right). We supose a cache memory able to store 5 spaces, and a 2x8 matrix stored like a vector.

Q3. Which is the explanation for the performance improvement due to loop fusion?

When doing the loop fusion, we are providing to the computer two independent operations that it can do at the same time, what we call paralelization. For this reason, we expect an increase in the IPC and a decrease of time execution and cycles needed. In addition, since the program has to access the memory spaces less times, we also expect a decrease in the instructions needed.

Q4. The double buffer strategy avoids executing the function *laplace_copy*(). From the previous table, compute the total number of instructions and cycles that takes the execution of *laplace_copy*() and estimate the IPC for that function. State a conclusion from this value.

Looking at the difference between the cycles and instructions with and without the copy function, we can deduce it needs 5.6G cycles and 1.1G instructions, which results in a function of 0.2 IPC. This number is very small, which makes the function very unefficient. Since it represented the 40% of the total cycles of the program, a big increase in the IPC after removing this function (from 0.81 to 1.20) is expected.

Q5. Explain the effect on performance of the code motion modification. Is it the execution of square root operations having a strong effect on performance?

By reducing the times we calculate the square root, we observe a reduction of the instructions and cycles needed to complete the program, resulting in an improvement of 1.2x execution time. Despite this, since this operation was done inside a loop among other operations, we observe a decrease of the IPC from 1.20 to 0.93, which makes us realize it is having a strong effect on performance.

Q6. Using the performance results obtained before, which is the performance bottleneck of the program lap2.c?

	elapsed time (s)	**Instructions (G)**	**IPC**
n = 4096, max_iter = 100	2.62	11.8G	1.4
n = 1024, max_iter = 1600	2.5	11.9G	1.44
n = 256, max_iter = 25600	2.46	12.2G	1.52

We have observed that, for a constant number of N*M*max_iter, we get a slower performance for bigger matrices. This is because the smaller the matrix, the more of it fits in the cache memory, which provides a faster performance. This results in a higher IPC, since we are able to compute more instructions per cycle.

Because of this reason, we can conclude the performance bottleneck of our program is the effect of the size of the matrix.

Q7. Explain the optimization idea implemented in lap2B.c and how much and why it works.

In this program, the strategy of using a second matrix *Anew* is abandoned. Instead, we are going to have an auxiliary vector TPM with space to store up to two matrix rows. In order to do that, we are going to take advantage on the fact that we only need three rows to compute the value and error of a determinate position.

Given three consecutive rows $j-1$, j and $j+1$ of the matrix, the new error function is going to calculate the new values of the middle row, and its going to store it in a row of the auxiliary vector TPM. In the next iteration, we are going to calculate de new values for the row $j+1$ using also j and $j+2$, and we are going to store it in the free row of the auxiliary vector. The main difference between the programs appears now, realizing that, since we are not going to need the row j anymore, we can replace the values of that row with the ones stored temporarily in the auxiliar vector, that now is going to be available to be overwritten

with the new information of next iteration. In this step, we also obtain the current maximum error of the system.

In the Table, we have gathered the performance results of each program used in order to compare them.

	Elapsed time (s)	**Instructions** (G)	**IPC**
lap2.c	2.5	11.8	1.40
lap2B.c	2.39	11.6	1.50
lap2C.c	1.76	9.3	1.62

Using lap2B.c, we have observed an improvement of 1.05x in the execution time, a decrease of 0.2G of instructions and an increase of 0.1 in the IPC. Since now the copy of the new elements can be done simultaniously to the calculation of new values, we can associate this slight immprovement to paralelization in this step. In addition, working with a smaller array also benefits the performance because it can be stored more easily in the cache memory, whiich allow us to acces to it faster and execute more instructions per cycle.

Q8. Explain the optimization idea implemented in lap2C.c and how much and why it works.

This program shares the concept of an auxiliary array that we analyzed in lap2B.c, with the difference that in this case it is going to be able to contain up to 3 rows of the matrix. We are going to call it T.

We are going to supose the matrix A contains the data of the system at time t. The program calculates and stores in T three consecutive rows $j-1$, j and $j+1$, that are going to be in time $t+1$. The main difference between the other programs is that now we are not going to copy this information in the matrix A. Actually, the whole matrix is never going to store the information of the system for the time $t+1$. Instead, the program uses the rows $j-1$, j and $j+1$ in time $t+1$ in the array T to calculate the data for the row j in time $t+2$, that it stores in A again. For the next step, the calculates the information for the row $j+2$ in time $t+1$, and stores it in the row of T that was occupied by $j-1$, already in $t+2$ in the matrix A and no longer needed for the following operations.

Looking at Table, we observe that using lap2B.c provides, compared to lap2.c, an improvement of 1.4x in the execution time, a decrease of 2.5G of instructions and an increase of 0.22 in the IPC.

Working this way, we avoid the copy of the data from the auxiliary vector to the original matrix, which results in a reduction of operations needed. In addition, we are able to advance 2 time steps in a single loop iteration. We saw in Q4 that the copy function was unefficient and represented a big part of the cycles of the program. Even though the function is not the same, we can extrapolate the concept and understand why lap2C.c presents a big improvement in the IPC.