Hardware and Code Optimization HW 1

Michel Gonzalez

November 13 2021

1 Logon and 'idev'

In this homework assignment I explored the *idev* tool to see compiler optimizations as well as the time differences in each optimization. I opened up a compute node by calling *idev–pdevelopment*, where the $-p$ command sets a queue to a named queue. Once in, I used the **hostname** command which just displays the computer's current hostname's domain. In my case the hostname command gave,

```
c202-020.frontera.tacc.utexas.edu
```

this was the only compute node I used in this assignment.

2 Setup

To begin this assignment I first made a directory with a path of,

```
/home1/08376/michel_g/Hw1
```

I copied the vector.c file into my directory and began analyzing the programs temporal behaviour. I first determined the size of the arrays a, b, and c in terms of number of elements and in number of bytes. The table 1 gives the sizes of each array in these terms.

The total amount of bytes from all three arrays is $8,397,824$ bytes or 8.398 MB. On the Frontera Cascade Lake node, an $L3$ cache block holds 38.5 MB per socket. Comparing these two values the three arrays take up $\approx 21.18\%$ of the total space on the $L3$ cache block.

	a	b	c
N	1024	1024^2	1024
$bytes$	1024	8388608	8192

Table 1: Table of element and byte size of arrays a, B, and c

The for loops to process the computation in the vector.c code are,

```
// Compute a = B * c
for (n = 0; n < iter; n++) {
  a[0] = 0.;
  for(j = 0; j < N; j++)  // inner loops
    for(i = 0; i < N; i++)
      a[i] = a[i] + B[i][j]*c[j];
```

The purpose of the two innermost loops can be explained by studying what data is being accessed by each loop. Now, looking at the outer most loop it will go through the columns of matrix B and match the row on the c vector. The inner, or fast loop will traverse through the rows of matrix B and it will also calculate a fraction of the total value of the a_i position of vector a through each iteration. In all this matrix vector product is actually being calculated column by column and not row by row. In a diagram setting this is what the two inner loops are doing.

$$\begin{bmatrix} b_{11} * c_1 \\ b_{21} * c_1 \\ b_{31} * c_1 \\ \vdots \end{bmatrix} + \begin{bmatrix} b_{12} * c_2 \\ b_{22} * c_2 \\ b_{32} * c_2 \\ \vdots \end{bmatrix} + \begin{bmatrix} b_{32} * c_3 \\ b_{32} * c_3 \\ b_{32} * c_3 \\ \vdots \end{bmatrix} + \cdots$$

These loops are important this is waht lets up compute vector a by accesing all the data wihtin B and c. With this method the only new data that will be coming in will be the values from matrix B while the value from vector c is held constant every j iteration. However since the vector c will be fully accessed one time after all loops are done, the loops cannot vectorize the data which limits the potential speed in computation.

Now, the outer most loops acts as a repeater for the computation. It is there since I will be studying the temporal behaviour of this computation while running optimization commands. Therefore it is really there to test the efficiency of the code by telling us how fast it took to compute the same computation *iter* times. If I removed the outermost loop it would preform the computation only once and move on to perform the rest of code.

3 Compiling and Executing

Normally the code is compiled using **icc –xcore-avx512 vector.c**, but I ran through the **-O1, -O2, -O2 -no-vec** optimization commands to see the time differences between each optimizer. The table below shows the result with he minimum and average time of execution for each optimization. Note that each optimization was tested 3 times.

(sec)	T_1	T_2	T_3	T_{avg}
norm	3.366751	3.370363	3.343685s **m**	3.360266
-O1	30.670408 **m**	32.402119	31.830988	31.634505
-O2	3.369080	3.343951 **m**	3.350200	3.354410
-O2 -no-vec	6.469891	6.469789 **m**	6.469994	6.469891

Table 2: Table of time test using the standard compiler and the optimization commands **-O1, -O2, -O2 -no-vec**.

The values with the **m** indicate the minimum time from the three trials. Next, I applied the following optimization; –O2 –qopt-report-phase=vec –qopt-report=3, which created a report based on the optimizations it did. The purpose of the -qopt-report-phase=vec flag is to specify the phase that reports are generated against, in this case it was generated against the phase for vectorization. The purpose of the qopt-report=3 flag is to give report details based on level 3 details. These details include reporting the loops that were vectorized, reporting the loops that were not vectorized, along with reason preventing vectorization, and an overall loop vectorization summary within each vectorized loop.

After running the optimization the report file **vector.optrpt** was created. The loops of interest had the line numbers 45, 48, 47 in that order. Note that line 48 comes before line 47 meaning that the optimization reorder the j for loop and the i for loop (the innermost loops). The optimized code would look like the following,

```
// Compute a = B * c
for (n = 0; n < iter; n++) {
  a[0] = 0.;
  for(i = 0; i < N; i++)
      for(j = 0; j < N; j++)  // inner loops
      a[i] = a[i] + B[i][j]*c[j];
```

The computation of $a = B * c$ is now reorganized. In terms of accessing the data in B it will now access all the data in a single column by going row by row.

Additionally vector c will also be accessed multiple times through each column. Now vector a is being computed position by position, meaning it will find the value of a_i fully and then move on to the next position. The following shows how the multiplication is now taking place.

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} & \dots \\ b_{21} & b_{22} & b_{23} & \dots \\ b_{31} & b_{32} & b_{33} & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} * \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \end{bmatrix} =$$

$$\begin{bmatrix} b_{11} * c_1 + b_{12} * c_2 + b_{13} * c_3 + \dots \\ b_{21} * c_1 + b_{22} * c_2 + b_{23} * c_3 + \dots \\ b_{31} * c_1 + b_{32} * c_2 + b_{33} * c_3 + \dots \\ \vdots \end{bmatrix}$$

This allows the program to vectorize the inner loop which greatly increases the speed of computation. This is why the optimization reorders the loops so it can vectorize between the data from matrix B and vector c to quickly compute the values of a_i.

The **-no-vec** disables vectorization from occurring which is why in the time test the **-O2 -no-vec** flag takes longer to compute. The flag still allows the compiler to reorder the loops, however without vectorization the computation time almost doubles. Additionally the **-O1** flag had the worst times being nearly 10 times longer than the **-O2** flag. This occurs because the **-O1** flag does not allow for loop optimizations so in this case it runs the code as is.