2022 COMP3320 Manual Vectorization Report

A. Introduction and Analysis on the Original Code

This report records experimenter's attempts on optimizing a piece of code that simulates the activity of one cloth falling on a static ball. The code initializes a set of nodes in a manner of SoA, that nine arrays were established for recording x, y, z vectors on position, velocity and force. The style of the code to update those vectors are "nested loops", it conducts calculation on each node row by row, and within rows, column by column.

The experiment asserted with four types of optimizations on the code, they are: trivial optimization, Single Instruction Multiple Direction (SIMD) register optimization, OpenMP optimization on the inner loop, and SIMD optimization packed with OpenMP, that the outer loops were distributed among several threads.

The experimenter decides to only focus on attributes of the number of nodes, and the depth of nodes' interaction, which affect the run time at most. Experimenter also think the radius of the ball could also affect the interaction, for its ability of affecting the branches during calculating vectors, that whether one node is contacting the ball. A few measurements later the experimenter consider this attribute doesn't have this affect. The experimenter can't create a specific situation that most of the time cloth is contacting the ball, or not.

Experiment results:

-n 200 -d 3 29643041 (us)

-n 200 -d 2 15221173 (us); -n 150 -d 2 6745592 (us); -n 100 -d 2 2839055 (us);

-n 200 -d 1 6232193 (us)

The relationship between runtime and -n is O(n^2), the relationship between runtime and -d is O(n).

In a gadi calculating node, the CPU and total time for running -n 150 -d 7 is 21.51s, this would be the comparison for any later optimization.

B. Trivial Optimization

The original code has a few design flaws, they have probability to cause the program to be cache inefficient and increasing the number of branches.

Cache inefficient design:

The flaws fall under this category looks like below

```
for(j=0; j<n; j++){
        for(i=0; i<n; i++){
                a[i*n+j] = .....;
                }
        }
```

Simply reverse the j and i in the loop will fix this.

Branch increasing design:

```
for(i=0; i<n; i++){
        x[i] = ....;
}
for(i=0; i<n; i++){
        y[i] = ....;
}
for(i=0; i<n; i++){
        z[i] = ....;
}
```

Experimenter decides to merge them into one loop, to decrease the number of branches.

Doing so will decrease the runtime to around 5447823 for a -n 200 -d 1 run, it also reduces the number of level 1 cache misses from 4.354*10^7 to 8.02*10^6.

B.1 An Interesting observation

The program has two branches that the experimenter thinks can be hard for one computer to predict. One is in velocity calculation that checks weather the cloth node is contacting the ball surface; one is in force calculation that checks weather two node the program is computing are the same one. During later optimizations that cannot handle branches, the experimenter assaulted few tricks to remove them, and test the correctness of his trick on the trivially optimized code. For the first one, the experimenter calculated the velocity of not contacting and contacting at once then add them with masks that is generated by the first bit of the subtraction between ball surface and node position. For the second one, he dissembles the loop into four, that avoids the central node. However, doing so will increase the runtime, make the optimized version become slower than the original. The takeaway for experimenter is that don’t do branch removing until necessary. After this event, he would think the second branch would be easy for the computer to predict, for -d 2, the branch history would be T T T T T T T N T T T T T T T for one loop, easy to predict.

C. SIMD Optimization

By using intel AVX2 registers, the experimenter has vectorized all the floating operations in the code to process four doubles at once, therefore, all the loops that used to calculating attributes of one node, can calculates attributes of four nodes.

C.1 Branches

The SIMD vectorization cannot handle branches, one computer couldn’t know the second or the third or the fourth nodes are contacting sphere or not, while processing the first node. To overcome this, the experimenter uses the AVX2 compare function to create masks, by subtracting ball surface’s position and nodes. The mask will become either 0x1 or 0x0 based on comparison. Then, the program needs to calculate the velocity of two situation, and apply add operation with them and the masks, adding them together, what is left are the correct result.

Another branch exhibits in the potential calculation function, where the code calculates distances between nodes, the function will not calculate the distance, when the “adjacent node” is the same with the one we want to know its force, the distance would be zero, and the calculation would be NAN. Experimenter removes this branch via the trick discussed in section B.1.

C.2 Result

On a gadi computational node, the run time for -n 150 -d 7 is 7.67s, compare to the original version or trivially optimized version, the improvement is around three times. The intel advisor’s comment on vectorized loop’s effect is “Average estimated speed-up of vectorized code compared to scalar version: 3.10x”

C.3 Pipelining

All the instructions in one loop are all pipelined by the experimenter, all the memory operations and ALUs are grouped with themselves.

The experimenter didn’t bother to try the effect of not doing instruction pipelining, neither measure the performance of it. From his experience in the COMP3320 lab 5, he anticipates the result of such would be negative to the performance.

C.4 Analysis

The experimenter initially anticipates AVX2 SIMD vectorized code to improve itself four times faster, the observation is three times. Experimenter's explanation to this is that he used unaligned store and unaligned load instructions, this would harm the performance.

D. OpenMP

D.1 Static value

Experimenter assault OpenMP vectorization by adding a line of pragma that inform the intel compiler to do "vectorized for loop" of the following loop, more precisely, "#pragma omp for reduction(+: value)".

The major challenge of doing so is to solve race conditions, which exhibit in two loops that calculate the overall kinetic energy and potential energy. His trick is firstly set the energy value to static, so shared by all the threads, then add a reduction clause in pragma, that compiler knows to add a synchronization barrier for such values.

Branches are handled like SIMD version.

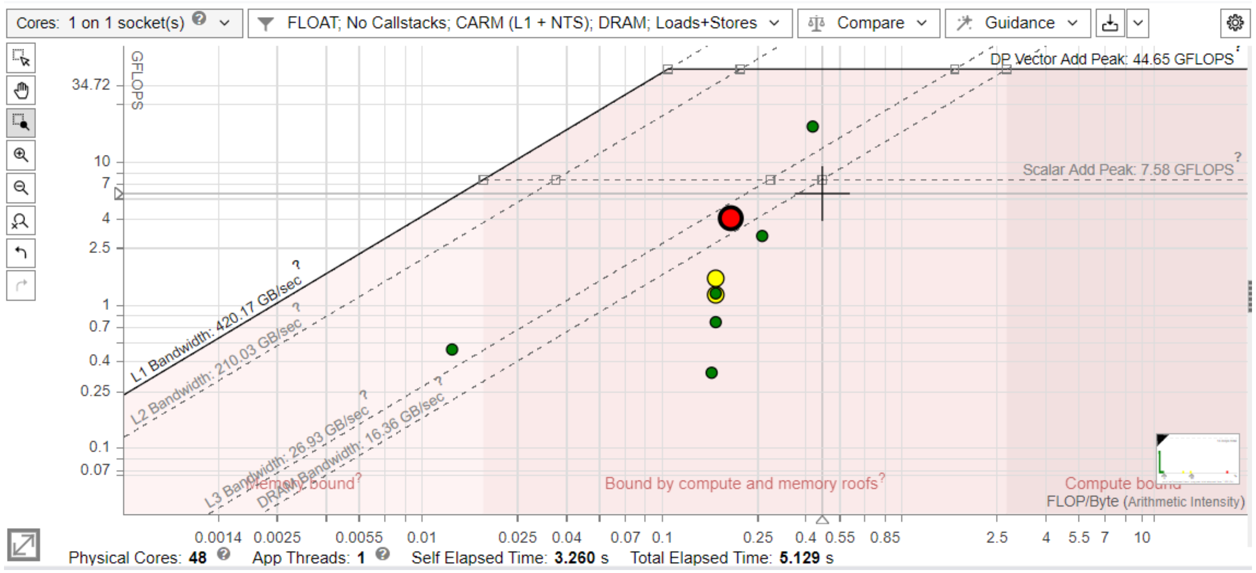
D.2 Result

For binary that is vectorized by OpenMP from the trivially optimized version, the run time is 9.01s, for -n 150 -d 7;

For binary that is vectorized by both OpenMP and SSE, the runtime for -d 150 -d 7 -p 10 is 8.03;

D.3 Analysis

The roofline plot for binary, kernel_omp looks like this:



Cores: 1 on 1 socket(s)

FLOAT; No Callstacks; CARM (L1 + NTS); DRAM; Loads+Stores

Physical Cores: 48 App Threads: 1 Self Elapsed Time: 3.260 s Total Elapsed Time: 5.129 s

The loop that spends most of the runtime is positioned on the region, that is both bounded by computational and L3 bandwidth roof, it is colored with red, in the code, it locates at line 345. Experimenter thinks this loop's memory access pattern has a vast space for improvement.

In the meantime, the green dot located at the very top of this graph is loop 110, it is a major loop that updates all node's velocity, both contacting or not contacting. Its memory accessing pattern is much greater than any others, even it has more FLOPs processed.

D 3.1 loop red analysis

The red loop is for calculating forces between nodes. The whole function where red exhibit work like this:

Set origin node.

Go through all the nodes that are adjacent to the origin, calculate the distance, save its distance and index into two arrays.

The above loop is segmented into a manner that it doesn't access origin node again, in order to removing a branch.

Go through the two arrays with adjacent nodes' info in a SSE vectorized manner.

The problem is clear now, all those "adjacent node info collection" loop is messing up the cache! When the real computationally intensive loop arrives, it will find that the info is all evicted to level 3 caches. In comparison, the green loop doesn't have this problem, therefore have a great performance.