Explanation of Varying Time Results

Alright, so I basic cache information that set size alternated with 512 bytes of data (size of cache divided by block size) so it made sense that stride sounds of around 512 should produce optimal results; of course, when the array is too much smaller than stride size, not much gets done (as it doesn’t have too many steps in the array before returning to the next logical block). As we can see, any stride size offered an improvement over the sample program, with a stride of 1024 performing very well on the largest data sets while a size of 256 excelled in general (256 is the size used for all the 2d graphs). Most likely the greater stride size for larger arrays allowed us to more thoroughly load forward loading and even miss forward loading in the TLB cache once every blue moon. For smaller ones, however, it wasn’t really as powerful most unfortunately.

As we can see, I also tried running a stride that wasn’t a multiple of 2; while this violates the ‘hop to next in same set’ so that we can make the cache change as much as possible, I thought that it might be able to ‘fool’ the cache by breaking the multiple of 2 standard. As we can see, that was not the case, however, and we ended up just limiting the number of times that we caused a cache miss because we weren’t constantly accessing blocks with the same set bits.

Model Name: Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz

Explanation of My Code

First, let’s start with what did not work. I initially tried to further fool the cache by running a program that swapped every k points in mem (i.e. mem[start] with mem[mem[mem[start]]] for k = 2) and using this to mix up points that were of the same set. However, this did not end up working too well and was shown to lower the speed of results, hence why I cast it aside.

Next, I tried to include a little bit of random sorting into the algorithm (utilizing modulo of high powers to determine the randomness) to see if the cache could be fooled in that way; once again, the results showed that this did not work, likely because the consequences of randomness introduced paled in comparison to the consequences of not stacking blocks with the same set bits.

Now, for what did work. First, the changeBy variable is used to group blocks with the same set number together (i.e. access mem[i] then mem[j] so that I and j have the same set bits so it becomes more likely something has to be kicked out of cache). Using this, I initialized the array so that when traversing itself it is constantly hitting the same set.

Next, we began swapping to try to prevent forward loading. What the next segment of code does is swap parts at the beginning of the array with parts of the same set at the end of the array (this is to combat the cache loading memory addresses that are sequentially after the last ones accessed).

Moving forward, we then swap elements with the element two after itself [access to 4,3,2,1 becomes 2,1,4,3] so that we could further pull apart addresses that were close in memory. Finally, we performed the same procedure on every other element which on the test cases resulted in the time being slower [2,1,4,3 becomes 1,2,3,4]. I am genuinely confused on why this worked but hey – don’t look a gift horse in the mouth.

Now, for a results summary – in tabular (and in graph – ooh ) forms:

3D comparison Visualizations (Stride Size vs Run Time)

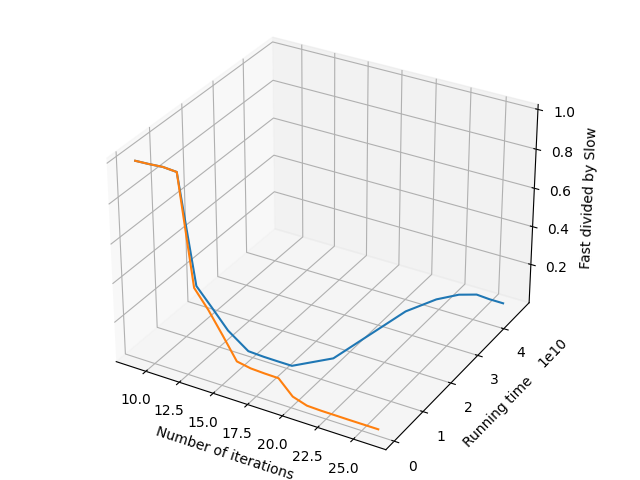
Chart, surface chart

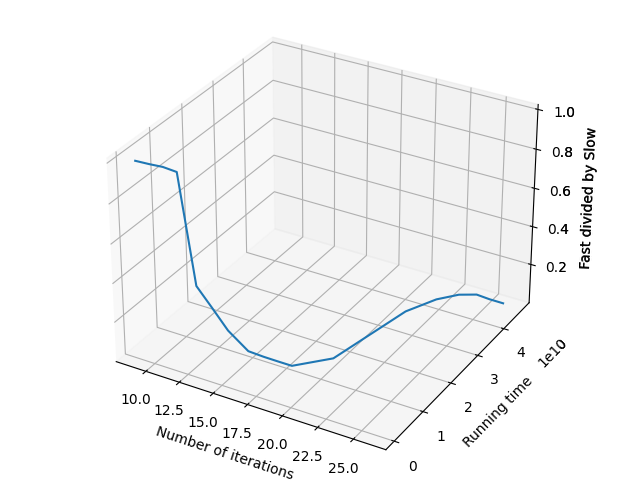
Description automatically generated

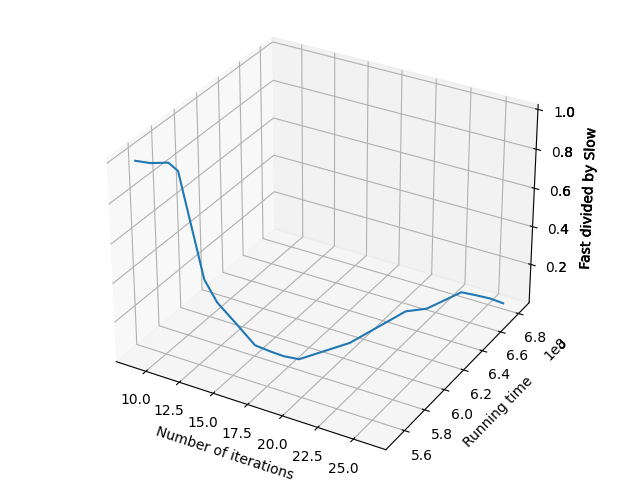
Chart, surface chart

Description automatically generated

256 Step Size Zoom In

Both Graphs

Slow Graph

Fast Graph

Further 2D Zoom Ins