CSC 485c Project, All-Pairs Shortest Path

Aiden Bull, Gareth Marriott

March 2020

1 Introduction

This is our second interim report for our project on optimizing a solution to the all-pairs shortest paths problem. The motivation for our project is specifically to compute the solution to all-pairs shortest paths between Wikipedia articles, treating hyperlinks on the articles as outgoing one directional edges. In the first interim report, we described our choice of algorithm, as well as optimizations we made to improve single threaded performance. Now we will treat that optimized single threaded solution as our baseline and describe what how we further improved it by adding parallelism. In this report we will first describe our single threaded baseline. We will then describe the parallel optimizations we made, and analyze the effectiveness of the optimizations. GitHub Link

2 Quality of Baseline

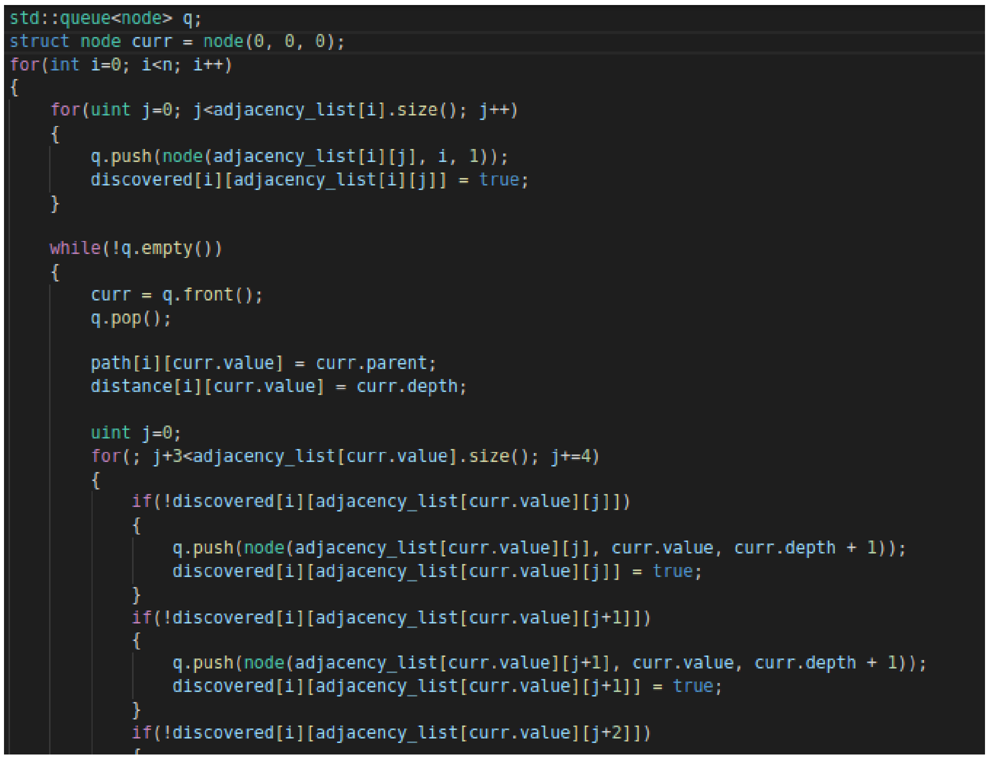
For our choice of algorithm, we allowed some assumptions based on our goal. Because our edges for the Wikipedia graph would represent hyperlinks, we made the assumption that our edges would be unweighted, as one click isn't more expensive than another. With this assumption, our problem is reduced to solving all-pairs shortest paths on an unweighted, directed graph. The algorithm that we chose for the problem was running breadth first search on each node. All-pairs BFS offers $O(mn + n^2)$ time complexity, with m representing number of edges and n representing number of nodes. This time complexity is competitive when compared to more robust algorithms for all-pairs shortest paths. This complexity is typically dominated by the mn term, which improves greatly as the graph becomes more sparse. This improvement with sparsity of the graph is another appealing feature for our goal. Each Wikipedia page will typically have a number of hyperlinks on the order of tens or hundreds, both of which pale in comparison to the total number of articles: around 6 million at the time of writing. These reasons led us to choose BFS as our baseline algorithm.

The input for our algorithm is a text file storing an adjacency matrix, where for each row i and each column j, the value stored at (i, j) is either -1, to represent no outgoing edge from i to j, or 1, to represent an edge from i to j. The data structures used an adjacency list initialized from the input text file, a distance matrix to store path distances, a predecessor matrix to keep track of paths, and a discovered matrix to determine whether an (i, j) pair has been added to a BFS FIFO queue. Each of the matrix data structures and the adjacency list were stored using vectors.

The bulk of our programs compute time comes from two functions. The first function is get_data(). This function performs all necessary preprocessing before actually running all-pairs BFS. In get_data() the adjacency list is initialized from the input text file, and the matrix data structures are initialized. The vast majority of our runtime, however comes from the bfs() function. This function consists of a large for loop, iterating N times, one for each node of the graph. In each of these iterations, single-source BFS is computed with each of the N nodes as the starting point. One iteration of single-source BFS starts with initializing a FIFO queue with the start node's immediate outgoing neighbours. Then each node is popped off the front of the queue. The popped node's corresponding distance and predecessor matrix positions are set to the values indicated by the popped node. Then each outgoing neighbour of the popped node are pushed to the back of the FIFO queue, unless they were already added to the queue, in which case they are ignored. This process repeats until the queue is empty.

Due to some misleading perf stats from our first interim report, we believed that our program was heavily front end bounded. Perf was reporting front end stall rates from 40% to 70%, while reporting very low cache miss rates. Because of this we spent little time focusing on back end optimizations, but we also struggled to find methods of improving the front end. In the end our only successful optimization and the only one to make it into this baseline was performing loop unrolling on the loop of single-source BFS that pushed the current node's neighbours to the FIFO queue. Applying loop unrolling was able to get us a 20% improvement over our single threaded baseline. The single threaded BFS baseline is displayed in figure 1, but part of the loop unrolling was cut off to save space.

Since the first submission, we looked again at our CPU usage statistics again, but this time using Intel's VTune profiler. The CPU usage statistics this time reported very few front end stalls, but a 38% back end stall rate, with most of the stalls be-



```
std::queue<node> q;
struct node curr = node(0, 0, 0);
for(int i=0; i<n; i++)
{
    for(uint j=0; j<adjacency_list[i].size(); j++)
    {
        q.push(node(adjacency_list[i][j], i, 1));
        discovered[i][adjacency_list[i][j]] = true;
    }

    while(!q.empty())
    {
        curr = q.front();
        q.pop();

        path[i][curr.value] = curr.parent;
        distance[i][curr.value] = curr.depth;

        uint j=0;
        for(; j+3<adjacency_list[curr.value].size(); j+=4)
        {
            if(!discovered[i][adjacency_list[curr.value][j]])
            {
                q.push(node(adjacency_list[curr.value][j], curr.value, curr.depth + 1));
                discovered[i][adjacency_list[curr.value][j]] = true;
            }
            if(!discovered[i][adjacency_list[curr.value][j+1]])
            {
                q.push(node(adjacency_list[curr.value][j+1], curr.value, curr.depth + 1));
                discovered[i][adjacency_list[curr.value][j+1]] = true;
            }
            if(!discovered[i][adjacency_list[curr.value][j+2]])
```

Figure 1: BFS code for single threaded baseline

ing core bound. This indicates that there are still single-threaded optimizations to be made, most likely improvements to instruction level parallelism, but also potentially some improvements to cache efficiency. The high number of core bound stalls would explain the improvements due to applying loop unrolling, which can help expose instruction level parallelism.

3 Parallel Algorithm Description

Our parallel optimizations take heavy advantage of the nature of our algorithm and our data structures used. Because the algorithm is simply running single source BFS on each node, it follows that parallelism should be attainable by simply running multiple single-source BFS iterations simultaneously. Attaining that was our target for achieving parallel speedup. Because each iteration of single-source BFS could logically be run independently, the issue facing us was avoiding race conditions. Inside the main all-pairs BFS loop, the distance, predecessor, and discovered matrices are all written to and thus are potentially sources of race conditions. Each data structure, however, is only written to at the index of their respective for loop iteration. Furthermore, each of the mentioned data structures are only *ref-*

erenced at the index of their loop iteration. The adjacency list is potentially referenced in any all-pairs loop iteration, but the adjacency list is never written to after being initialized. Because of these access patterns, it is impossible for errors to occur due to race conditions on the four main data structures. That doesn't mean that every data structure could be easily switched to parallel. The FIFO queue and a temporary node variable were both declared outside of the all-pairs BFS loop but were shared for each single-source iteration. To remedy this, the declarations for the FIFO queue and temporary node were moved inside the parallelised all-pairs BFS loop. This adds a bit of overhead, but placed each queue outside of scope of other threads. With that change, the all-pairs BFS loop was fully parallelisable and a "#pragma omp for" was used to parallelise the loop.

A small amount of parallelism was also added to the get_data() function. A loop initializing values in the each of the matrix data structures was parallelised due to each data location being written to only once. The running time of this loop was dwarfed, however, by the loop that initialized the adjacency list from the input text file. We were unable to parallise this loop due to the lines being read in order using the getline() function. A potential speedup is, however, possible by converting the input from adjacency matrix form to adjacency list form, which will be explored in the next submission.

4 Experiment Setup

For our second milestone we moved our tests to the Azure server and scaled up the size of our data set. With the increased speed we achieved in our parallel implementation we were able to test on larger data sets that would better relate to the Wikipedia data set. Our test graphs are categorized by the their sparsity levels and size below. The sparsity level is a fraction and determines the likelihood of any directional edge existing. Timing results were calculated using Chrono timers wrapped around our BFS algorithm call. The times below are how long the bfs algorithm takes to calculate the All-Pairs shortest path problem for a given size and sparsity level. Each recorded number is the average of two tests run on the Azure server. The variability in length was extremely small so it was determined that two tests were enough. The blank indicate no tests were done. For a complete view of all the data visit: Google Spreadsheet link

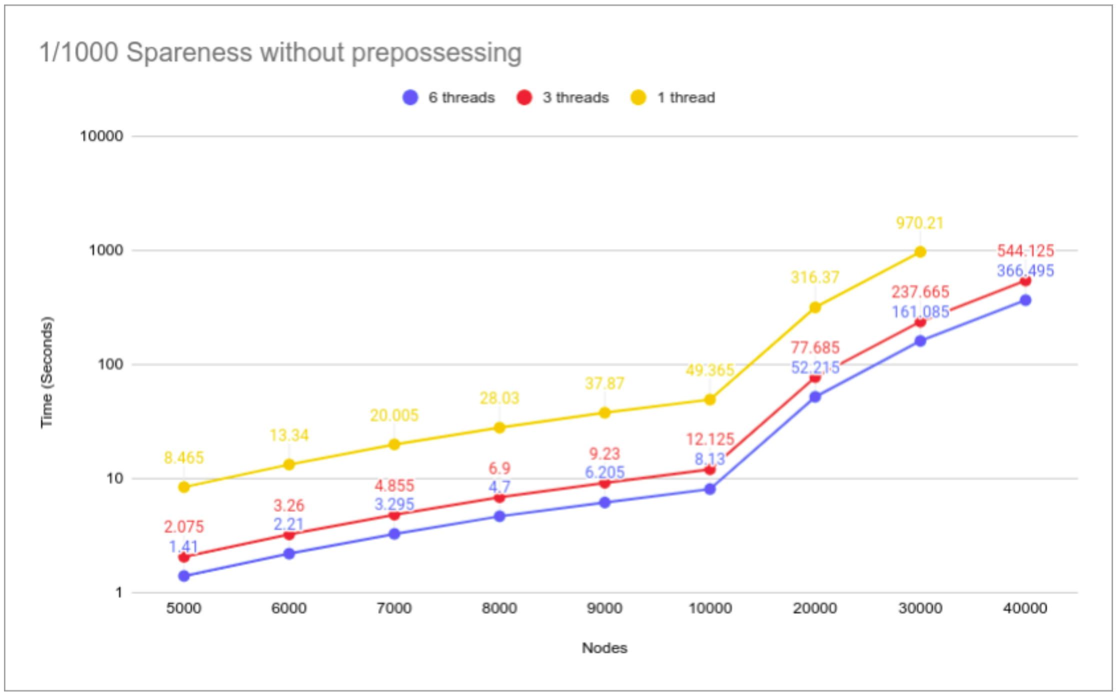


Figure 2: The speed of the 1, 4, and 6 thread optimizations on a log scale. (The increase in rate of growth is due to an increase in the jumps between data size)

6 Threads(in seconds)		
nodes	1/100	1/1000
5000	6.775	1.41
6000	11.44	2.21
7000	17.805	3.295
8000	26.305	4.70
9000	37.135	6.205
10000		8.13
20000		52.215
30000		161.085
40000		366.495

4 Threads(in seconds)		
nodes	1/100	1/1000
5000	10.035	2.078
6000	16.935	3.26
7000	26.45	4.86
8000	39.06	6.90
9000	55.105	9.23
10000		12.125
20000		77.685
30000		237.655
40000		544.125

1 Thread(in seconds)		
nodes	1/100	1/1000
5000	40.845	8.465
6000	69.08	13.34
7000	108.165	20.005
8000	159.445	28.03
9000	225.745	37.87
10000		49.365
20000		316.37
30000		970.21
40000		

The results showed an average of 517% speed up of the 6 thread implementation compared to the single threaded version. The improvements remained consistent between small and large data sets implying a small or no memory stalls.

5 Parallel Scalability

Through testing our algorithms with three different numbers of threads we found that our gains from parallelism scale extremely well. We saw an average of 300% increase going from 1 to 4 threads and an average of 500% increase in speed going from 1 thread to 6.(See table below) This means those 2 final threads contributed to 200% of that growth. This puts our algorithm in an extremely good position when it comes to paralleling to GPU's. From our profiling and testing we see no hindrance in paralysing our code up to the size of the number of nodes.

Percent Increase		
nodes	1 to 6 Threads	1 to 4 Threads
5000	500.35%	307.95%
6000	503.62%	309.20%
7000	507.13%	312.05%
8000	496.38%	306.23%
9000	510.31%	310.29%
10000	507.20%	307.13%
20000	505.90%	307.25%
30000	502.30%	308.23%

6 Evidence of Work Efficiency

Through using VTune we have gathered a much stronger understanding of the bottlenecks in our code. While using Vtune we used a C++ header file to explicitly declare all the data and compile it into the executable. This let us more closely examine the algorithmic components of our code. It is important to note this profiling was done on Intel(R) Core(TM) i7-3520M CPU which was different than the Azure server the timing tests were performed on.

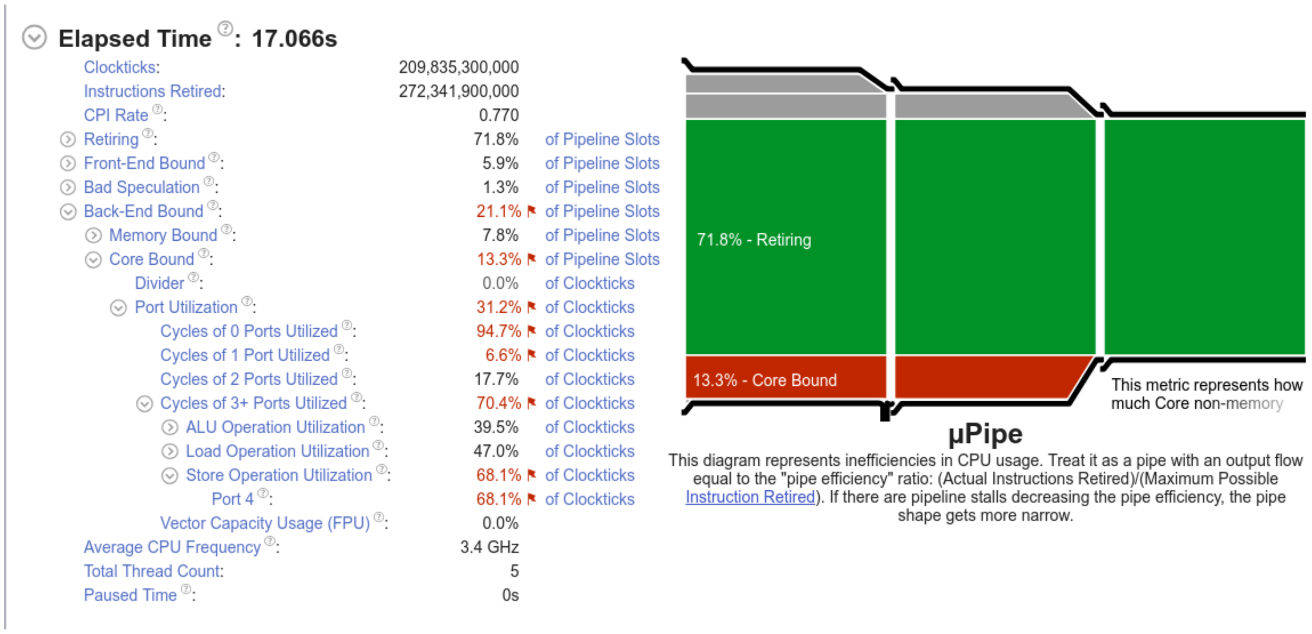
Our parallel implementation had 71.8% of all possible instructions retire and a core bound stall rate of 13.3% with an effective Logical Core utilization of 3.624 out of 4. This was a large increase from our single threaded implementation with our single threaded implementation having only 58.1% of possible instructions retire and 29.9% be core bound.

The next steps would be to look at the port usage and see if we could get better ILP.

7 Analysis and Evidence to Support Design Decisions

The parallel improvements to our single threaded baseline include the parallelisation of the outer for loop wrapping the single source BFS iterations and a minor improvement from parallelising a loop that initialized the matrix data structures. The parallelism for both loops were implemented using a call to "#pragma omp for" for each loop. Additionally, in order to parallelise the outer BFS loop, the shared FIFO queue and temporary node was changed to be reinitialized for each single source iteration. Of the improvements, the most important and impactful change was by far the parallelisation of the outer loop in all pairs BFS. The algorithm and data structures both lent themselves very well to the introduction of parallelism, with the algorithm essentially being repeated computation of another independent algorithm, and the data structures naturally avoiding the need for locks.

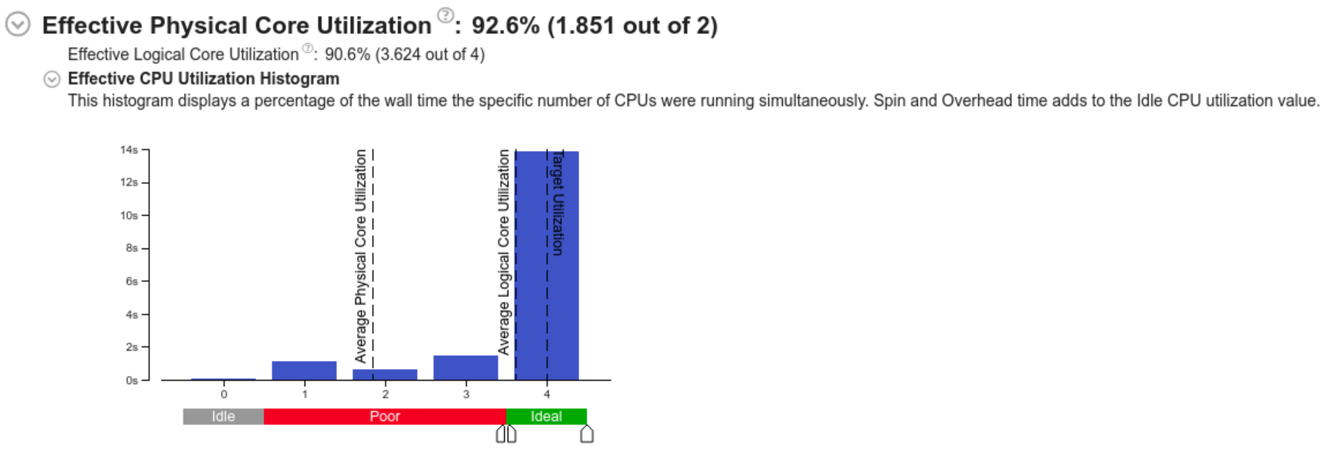
Our adjustments to introduce parallelism almost certainly led to the improvements described in the previous sections. The mere inclusion of the call to #pragma omp for" in bfs() causes the runtime of the function to decrease to 19% of the same function with that line excluded. The overhead added by reinitialising the FIFO queue and temporary node are expected to be negligible due to the simplicity of the queue and small size of the node struct.



```
Elapsed Time: 17.066s
    Clockticks:                              209,835,300,000
    Instructions Retired:                    272,341,900,000
    CPI Rate:                                          0.770
    Retiring:                                          71.8%    of Pipeline Slots
    Front-End Bound:                                    5.9%    of Pipeline Slots
    Bad Speculation:                                    1.3%    of Pipeline Slots
    Back-End Bound:                                    21.1%    of Pipeline Slots
        Memory Bound:                                   7.8%    of Pipeline Slots
        Core Bound:                                    13.3%    of Pipeline Slots
            Divider:                                    0.0%    of Clockticks
            Port Utilization:                          31.2%    of Clockticks
                Cycles of 0 Ports Utilized:            94.7%    of Clockticks
                Cycles of 1 Port Utilized:              6.6%    of Clockticks
                Cycles of 2 Ports Utilized:            17.7%    of Clockticks
                Cycles of 3+ Ports Utilized:           70.4%    of Clockticks
                    ALU Operation Utilization:         39.5%    of Clockticks
                    Load Operation Utilization:        47.0%    of Clockticks
                    Store Operation Utilization:       68.1%    of Clockticks
                        Port 4:                        68.1%    of Clockticks
                Vector Capacity Usage (FPU):            0.0%
    Average CPU Frequency:                           3.4 GHz
    Total Thread Count:                                    5
    Paused Time:                                          0s
```

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Figure 3: Pipeline and breakdown of clock cycles: 4 logical cores



Effective Physical Core Utilization: 92.6% (1.851 out of 2)

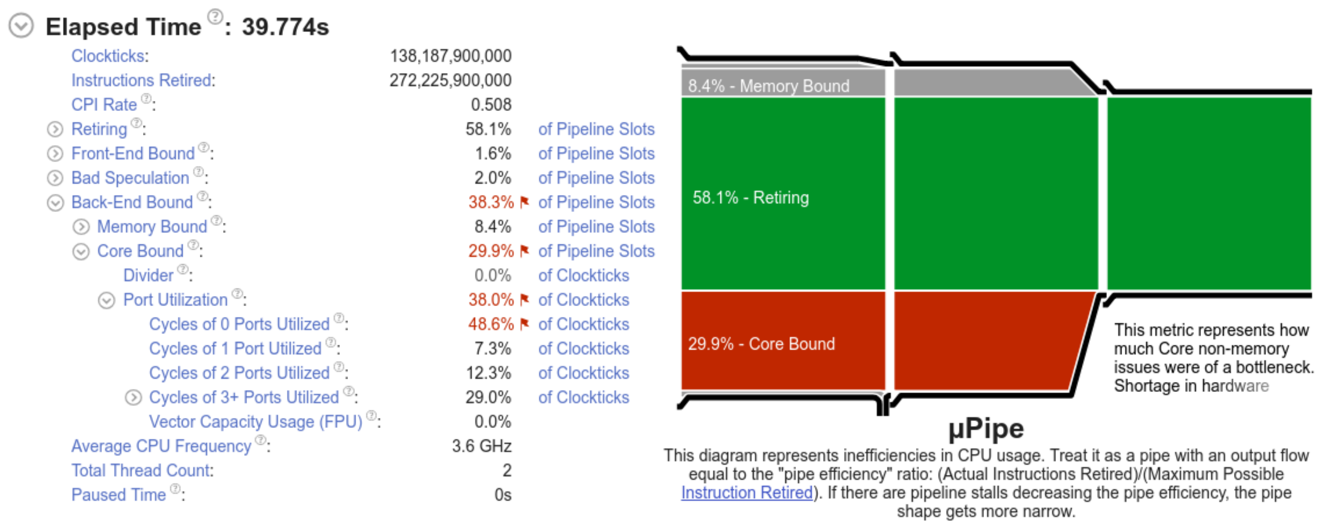
Effective Logical Core Utilization: 90.6% (3.624 out of 4)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.

14s
12s
10s
8s
6s
4s
2s
0s
0
1
2
3
4
Average Physical Core Utilization
Average Logical Core Utilization
Target Utilization
Idle
Poor
Ideal

Figure 4: The effective number of CPU's used (multithreaded implementation)



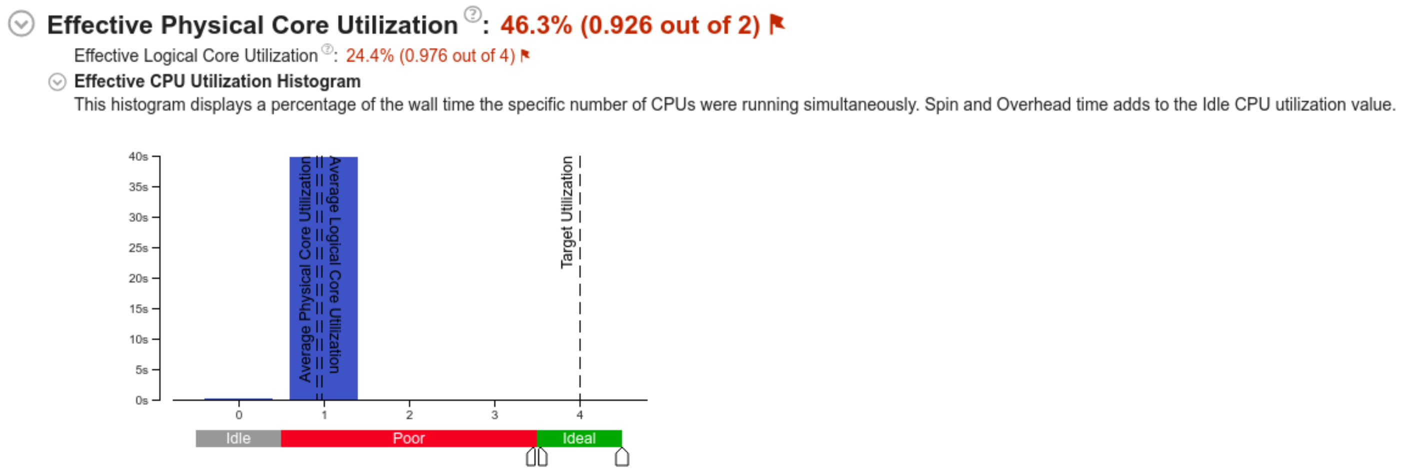
Elapsed Time: 39.774s

Clockticks:	138,187,900,000	
Instructions Retired:	272,225,900,000	
CPI Rate:	0.508	
Retiring:	58.1%	of Pipeline Slots
Front-End Bound:	1.6%	of Pipeline Slots
Bad Speculation:	2.0%	of Pipeline Slots
Back-End Bound:	38.3%	of Pipeline Slots
Memory Bound:	8.4%	of Pipeline Slots
Core Bound:	29.9%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	38.0%	of Clockticks
Cycles of 0 Ports Utilized:	48.6%	of Clockticks
Cycles of 1 Port Utilized:	7.3%	of Clockticks
Cycles of 2 Ports Utilized:	12.3%	of Clockticks
Cycles of 3+ Ports Utilized:	29.0%	of Clockticks
Vector Capacity Usage (FPU):	0.0%	
Average CPU Frequency:	3.6 GHz	
Total Thread Count:	2	
Paused Time:	0s	

This metric represents how much Core non-memory issues were of a bottleneck. Shortage in hardware

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Figure 5: Pipeline and breakdown of clock cycles: 1 logical core



Effective Physical Core Utilization: 46.3% (0.926 out of 2)

Effective Logical Core Utilization: 24.4% (0.976 out of 4)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.

40s
35s
30s
25s
20s
15s
10s
5s
0s
Average Physical Core Utilization
Average Logical Core Utilization
Target Utilization
0
1
2
3
4
Idle
Poor
Ideal

Figure 6: The effective number of CPU's used (single threaded implementation)