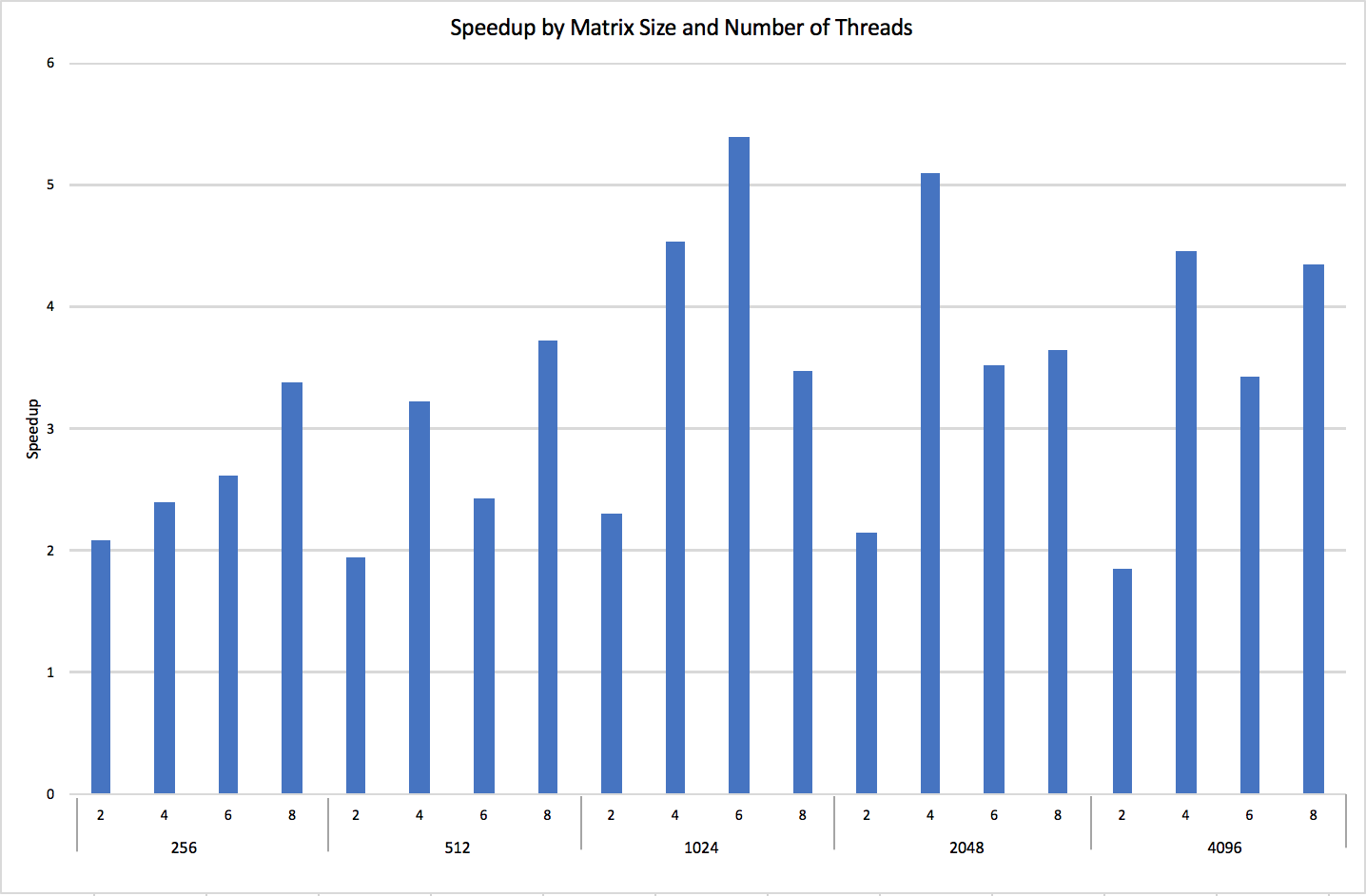
Project 4

1.

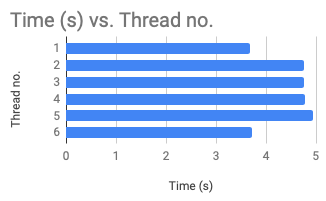
**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Matrix Size (n)* | *Number of Threads* | *Speedup* | *Matrix Size (n)* | *Number of Threads* | *Speedup* |
| 256 | 2 | 2.07865 | 2048 | 2 | 2.14397 |
| 4 | 2.40257 | 4 | 5.09953 |
| 6 | 2.61268 | 6 | 3.52957 |
| 8 | 3.38805 | 8 | 3.65047 |
| 512 | 2 | 1.95195 | 4096 | 2 | 1.85488 |
| 4 | 3.22792 | 4 | 4.45559 |
| 6 | 2.43135 | 6 | 3.421 |
| 8 | 3.73149 | 8 | 4.34563 |
| 1024 | 2 | 2.30983 |  | | |
| 4 | 4.53283 |
| 6 | 5.39658 |
| 8 | 3.46891 |

The graph above shows that while in some cases the speedup is linear with the increase in the number of threads, the matrix size 256, it does not hold true in all cases. In matrix size 512, there is similar linearity but only with thread counts 2 and 4, which repeats with thread numbers 6 and 8. In matrix size 1024 there is linearity from thread counts 2, 4 and 6 but the speedup drops back down for thread count 8, indicating an upper limit on speedup. With matrix size 2048, there is essentially no linearity at all. However, matrix size 4096 shows a similar pattern to thread counts in a matrix size of 512, just with greater overall speedup values.

**Hypothesis:** The speedup is not linear with the total number of threads because theoretical speedup is limited by the serial part of the program. There is a limit to the speedup on any given function no matter the number of threads used. The upper bound of parallel execution speedup is limited by the number of processors on the machine, a parallel fraction and the serial code

2. As an example, in the scenario dimension of the matrix (n) = 1024 & the total number of threads (t) = 6, this is the amount of time each thread took to finish the assigned task.



|  |  |
| --- | --- |
| **Thread no.** | **Time (s)** |
| 1 | 3.68613 |
| 2 | 4.75072 |
| 3 | 4.76667 |
| 4 | 4.78682 |
| 5 | 4.93003 |
| 6 | 3.71994 |

The average time to complete the task was 4.4401 with a variance of 0.3301 and a range of 1.2439. They completed in order of thread 1, 6, 2, 3, 4, & 5. While these are not all executing at the exact same amount of time, it is very similar and within a little over one second of each other. This proves the hypothesis presented in problem 1 because the threads are all concurrent executions that share an address space. It makes sense that they would all share similar execution times. However, as it is seen, the first and the last threads have the shortest execution time. This could be due to the scheduling of these threads having fewer cache misses and data dependencies because they are both the first and the last ones to execute. The first thread does not have to stall to wait on any previous data while the last thread does not have to stall to allow for the execution of following threads.

**3. Best speedup:**

|  |  |  |
| --- | --- | --- |
| **Dimension of the matrix (n)** | **total number of threads used (t)** | **speedup (s)** |
| 1024 | 6 | 5.39658 |

The speedup for this (n = 1024, t = 6) combination outperformed the speedup achieved elsewhere because it is an optimal number of threads for this size matrix and this processor. With too many threads, there is more overhead in the process and additional cache/TLB conflicts. A lower number of threads is better because it hides occasional cache miss latencies. Additionally, the processor used is i7 8th gen processor, which has an effect on the processing power and the number of threads speeds processed at once.

4. The restrictions initially faced with using a basic parallel version of the matrix multiplication involves the cache, loops, and operations. With the cache, when we run into the matrices that start exceeding a usable level, the cache becomes increasingly small and ends up generating cache misses. Also, since the matrix multiplication with a parallel version is memory consuming, it would be a great improvement to the overall design of the algorithm to include an organization of the cache hierarchy that prioritizes the input accordingly. One thing to consider for optimization when it comes to memory consumption is storing memory in Stack vs. Heap. Stack has limited memory, but it can provide for more efficient intermediate calculations if we use a stack with predefined memory allocations. As for the loops, when a loop run into there are increasing usages of instructions on the branch, and with this, we are slowing down the algorithm. To fix this, we need to lessen the number of operations that are being run through. So, we want our nested loops to find the one that is deepest in the function and to have it run through a few iterations as possible. By doing so, we then have fewer branch instructions and ultimately are able to optimize the algorithm. Another way of optimizing matrix multiplication is storing the redundant calculations and you can access the answer to that calculation when it is required.