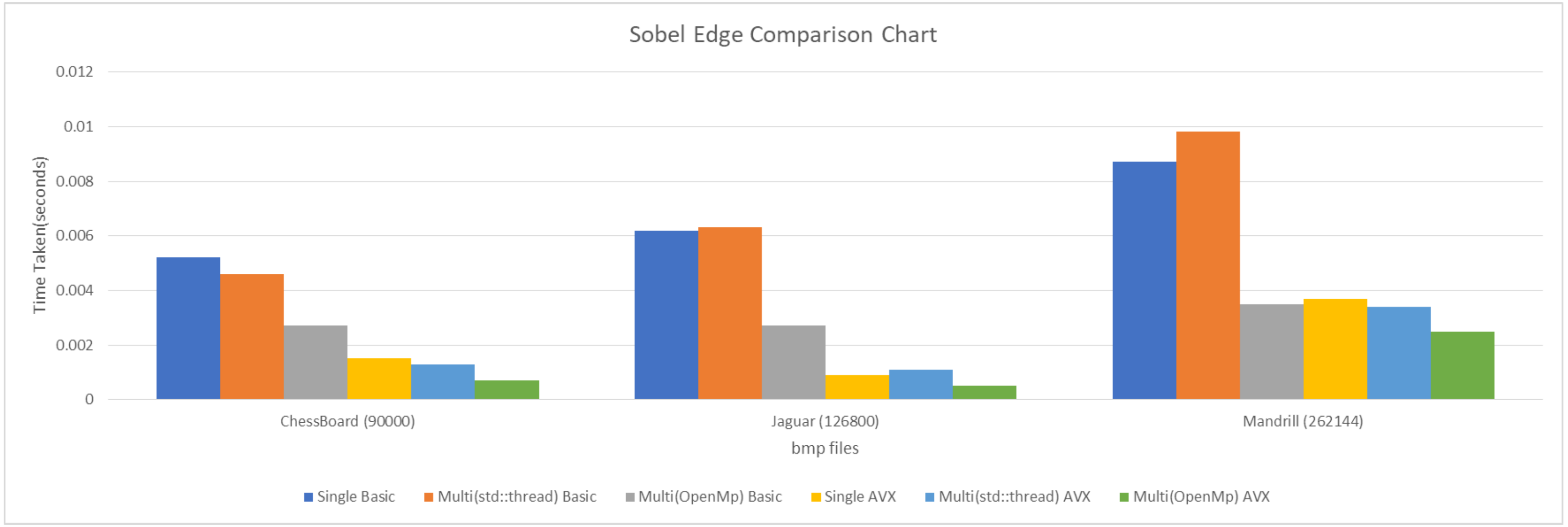
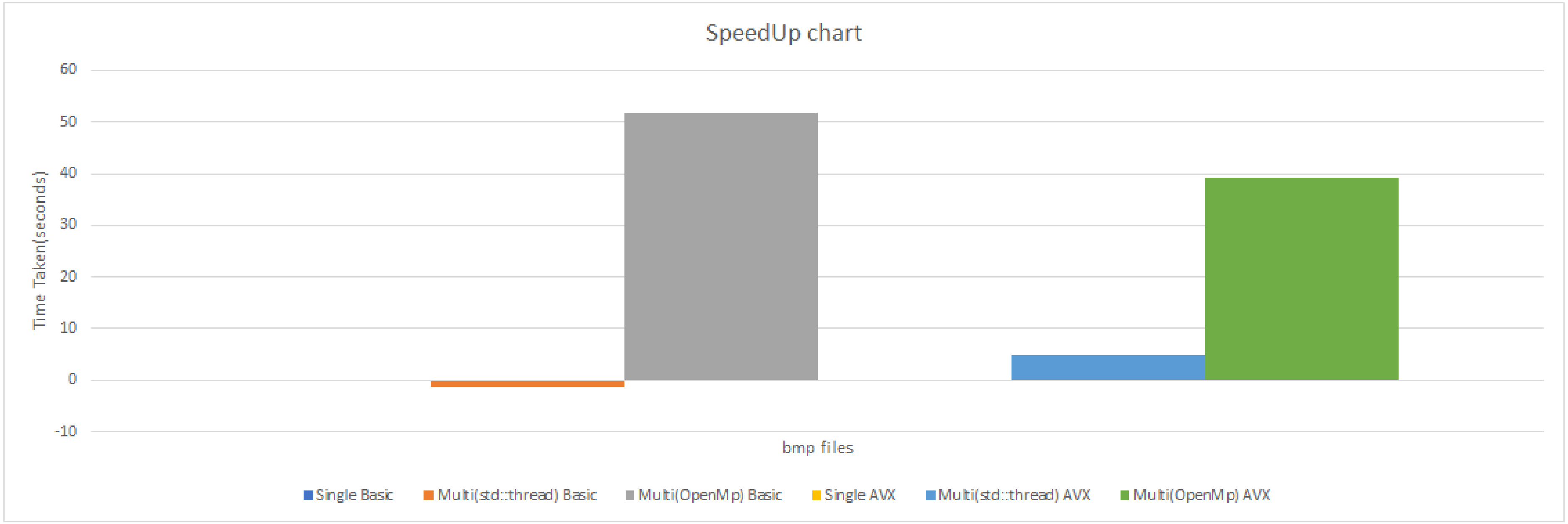


	Single Basic	Multi(std::thread) Basic	Multi(OpenMp) Basic	Single AVX	Multi(std::thread) AVX	Multi(OpenMp) AVX
ChessBoard (90000)	0.0052	0.0046	0.0027	0.0015	0.0013	0.0007
Jaguar (126800)	0.0062	0.0063	0.0027	0.0009	0.0011	0.0005
Mandrill (262144)	0.0087	0.0098	0.0035	0.0037	0.0034	0.0025

These are the values I obtained on my 4 core computer. The results actually do fluctuate quite a bit so I took the estimated average from my runs.



	Single total	Multi(std::thread) Basic	Multi(OpenMp) Basic	Single AVX	Multi(std::thread) AVX	Multi(OpenMp) AVX
Total time for all bmp(s)	0.0262	0.0265	0.0126	0.0061	0.0058	0.0037
Speedup(%)	0	-1.145038168	51.90839695	0	4.918032787	39.3442623

The speedup is comparing basic sobel with basic sobel multithreaded and AVX sobel with AVX sobel multithreaded.

Report:

The technique I have chosen to use for my submission is to use openMp. OpenMP directive is extremely simple and easy to use as it handles the creation, destruction and synchronization of threads for me.

At first, I tried to parallelise my code using std::thread. I did it by creating worker threads and storing them inside a worker queue before the function is ran.I modified the code to split the entire image to process into different chunks of block and let one thread process each. As you can see in the charts above, my implementation for the basic function became slower while my implementation for the AVX function was only 5% faster. I believe the downfalls are attributed to my work queue having too much overhead and the modification of the code created more checks which slowed down the code even more compared to the original code.

However, I was able to achieve 52% speedup using openMP for the basic sobel and 40% for the AVX version. The modification of the original code to the openMP code is really minute. My idea was to split the heights in each layer into blocks for openMP to parralize. I did this by using the **schedule(static)** operative which will evenly split the number of iterations for each thread created by openMP. With openMp handling the creation and destruction of threads, there are much less overhead compared to my implementation of the worker queue. By allowing threads to each run a portion of the height also ensures the threads will not write memory in the same cache block, minimizing false sharing.