COSC7502 – Assignment 2
Joel Thomas 44793203

Introduction:

Using the provided Serial implementation as a template, the $MatrixVectorMultiply()$ function used by the $ARPACK$ library was parallelised using OpenMP, AVX and CUDA. Due to other commitments, there was insufficient time available to successfully complete and test an MPI implementation.

Implementation Methodology:

Commencing with the OpenMP implementation, the most sensible approach to parallelisation was to calculate each $Y_i, 1 \leq i \leq N$ on a separate thread. This resulted in just a single $\#pragma\ omp\ for$ clause on the outer loop. Synchronisation in terms of a critical/barrier clause was not required as each Y_i is completely independent from each Y_j for $i \neq j, 1 \leq i, j \leq N$.

Regarding the AVX implementation, it was too difficult to implement by casting both vectors Y and X into AVX-based vectors of four 64-bit packed doubles and then working with these new types because this methodology would not carry over well for the matrix M. Instead, for each Y_i, the summation $\sum_{j=1}^{N} M_{i,j}X_j$ was processed in blocks of 4 i.e. for a given j, process $\left[M_{i,j}X_j, M_{i,j+1}X_{j+1}, M_{i,j+2}X_{j+2}, M_{i,j+3}X_{j+3}\right]$ before incrementing $j \mathrel{+}= 4$. This enabled easy and natural use of the fused-multiply add operation. For example, the first element in the packed doubles vector would eventually store $M_{i,j}X_j + M_{i,j+4}X_{j+4} + M_{i,j+8}X_{j+8} + \cdots$. Next, horizontal add and add operations ensured that in the end each element of this vector would almost store $\sum_{j=1}^{N} M_{i,j}X_j$. The term "almost" is used because any remainder due to the specific choice of N being not divisible by 4 was handled directly in a manner similar to the serial implementation (without use of any AVX/FMA intrinsics).

With the CUDA implementation, handling every single $M_{i,j}X_j$ on a separate GPU thread was avoided to prevent race conditions from arising. Given a Y_i, dedicating each thread to calculating $M_{i,j}X_j$ for some j would mean simultaneous loads and saves to register when incrementing Y_i resulting in unintending non-deterministic program behaviour (when the random seed is set). Instead, a solution similar to the OpenMP approach was used - each Y_i was calculated independently and completely on a single GPU thread. GPU memory was allocated just once for Y, X and M since any further allocations would be completely unnecessary (their dimensions don't change). While M was copied to GPU memory just once after it was initialised, it was required that Y and X were re-copied every single time a call to $MatrixVectorMultiply()$ was made in order to update their GPU counterparts.

Results Discussion:

During thorough testing, several key metrics from the output of each test run were saved and used to produce the figures seen below and these figures shall now be discussed briefly. The parameters set when tested on the Slurm workload manager were – 16 CPU threads for OpenMP, 10 GPU threads per block for CUDA and 8 GB of RAM common to all tests.

The first diagram for display to the reader is in figure 1 below. This is essentially a scatterplot of the average largest eigenvalue based on 10 runs for each of $N = 1000$ to $N = 10000$ in increments of 1000 for the Serial/OpenMP/AVX/CUDA implementations. With the random seed set to 42 during testing, the purpose of this diagram is to show that all parallel implementations are stable and consistent with the serial implementation in that the results are always deterministic and reliable when reproducibility is allowed (random seed fixed). This is to ensure that the parallel implementations are bug-free and are not causing unintended program behaviours.

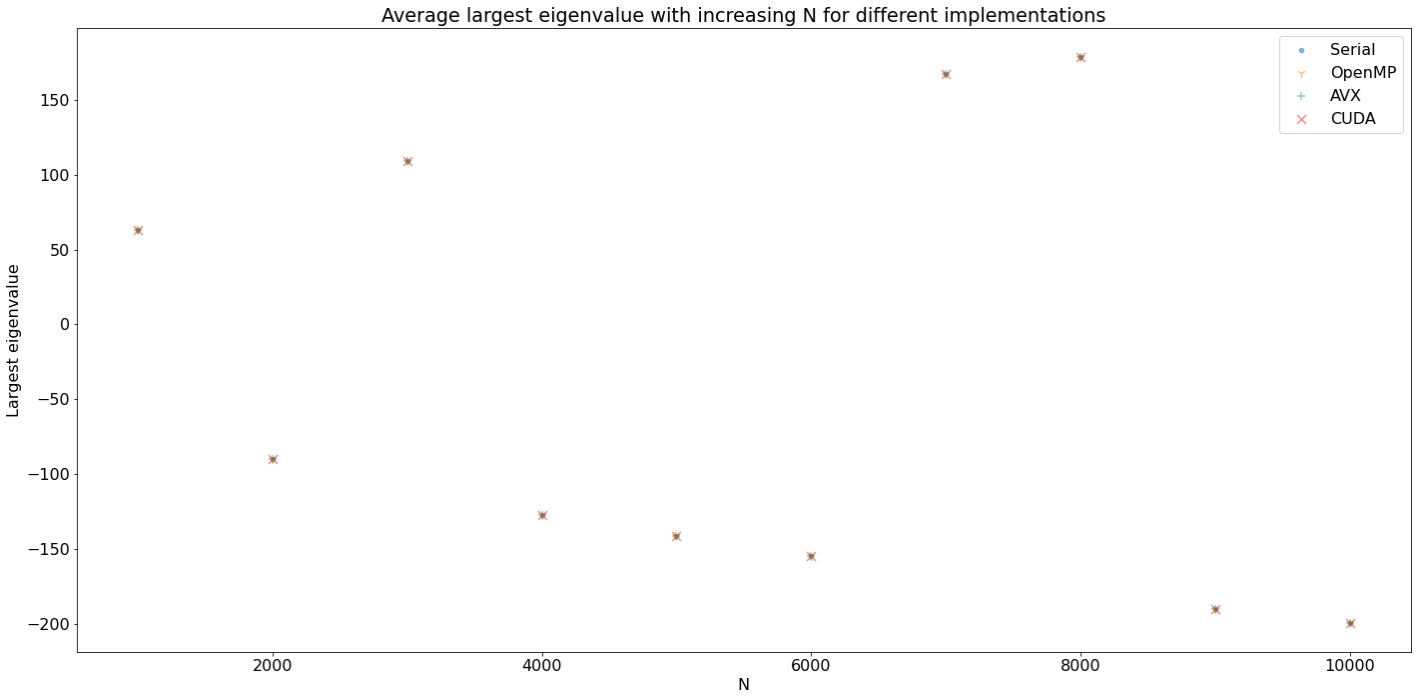

Figure 1: Average largest eigenvalue based on 10 runs for each of $N = 1000$ to $N = 10000$ in increments of 1000 for each implementation.

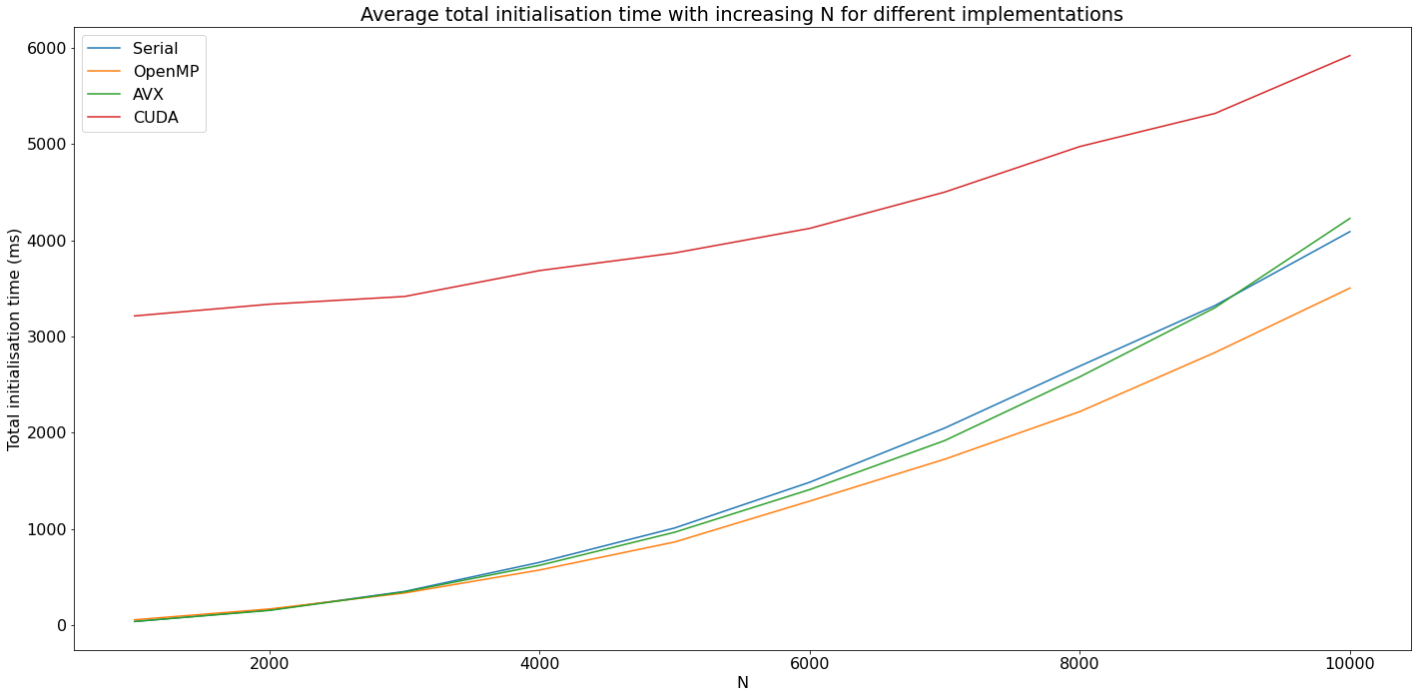

Figure 2: Average total initialisation time based on 10 runs for each of $N = 1000$ to $N = 10000$ in increments of 1000 for each implementation.

The second diagram available for display is in figure 2 below and is a plot of the average total initialisation time based on 10 runs for each of N=1000 to N=10000 in increments of 1000 for each implementation. The CUDA implementation naturally takes longer to initialise since in addition to allocating memory and initialising the matrix M on the CPU, it needs to allocate memory and copy it over to the GPU as well. On average, this is roughly a factor of 84 times slower at $N = 1000$ to just under a factor of 1.5 times slower at $N = 10000$ – the slowdown reduces massively as N grows larger. It is unclear why the OpenMP implementation takes reduced time to initialise compared to the Serial and AVX implementations for large N since the initialisation code for all three of these implementations is exactly identical.

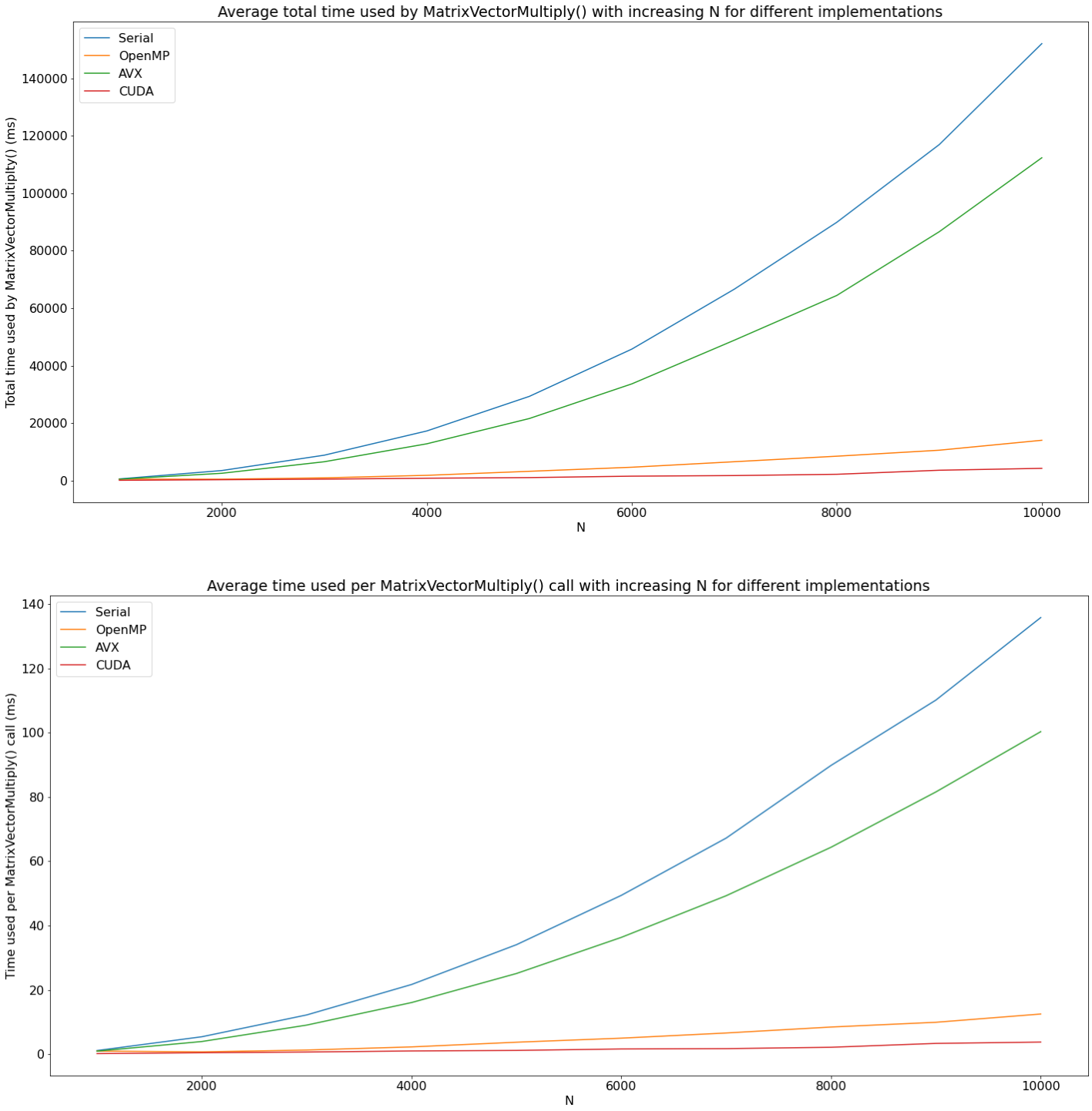

Figure 3: Average total time used by and time used per $MatrixVectorMultiply()$ call based on 10 runs for each of $N = 1000$ to $N = 10000$ in increments of 1000 for each implementation.

Figure 3 above consists of two very similar diagrams – the first represents the average **total** time used by and the second represents the average time used **per** $MatrixVectorMultiply()$ call based on 10 runs for each of $N = 1000$ to $N = 10000$ in increments of 1000 for each implementation. These are the standard diagrams one expects to view when concerned with the speed-ups brought about by the parallelisation of a serial implementation. On average, at $N = 10000$, the OpenMP, AVX and CUDA implementations have a speed-up of roughly 11 times, 1.35 times and 35 times faster respectively **just for** the matrix-vector multiply function. Overall, it appears that good performance scaling is brought about by parallelisation compared to the serialisation.

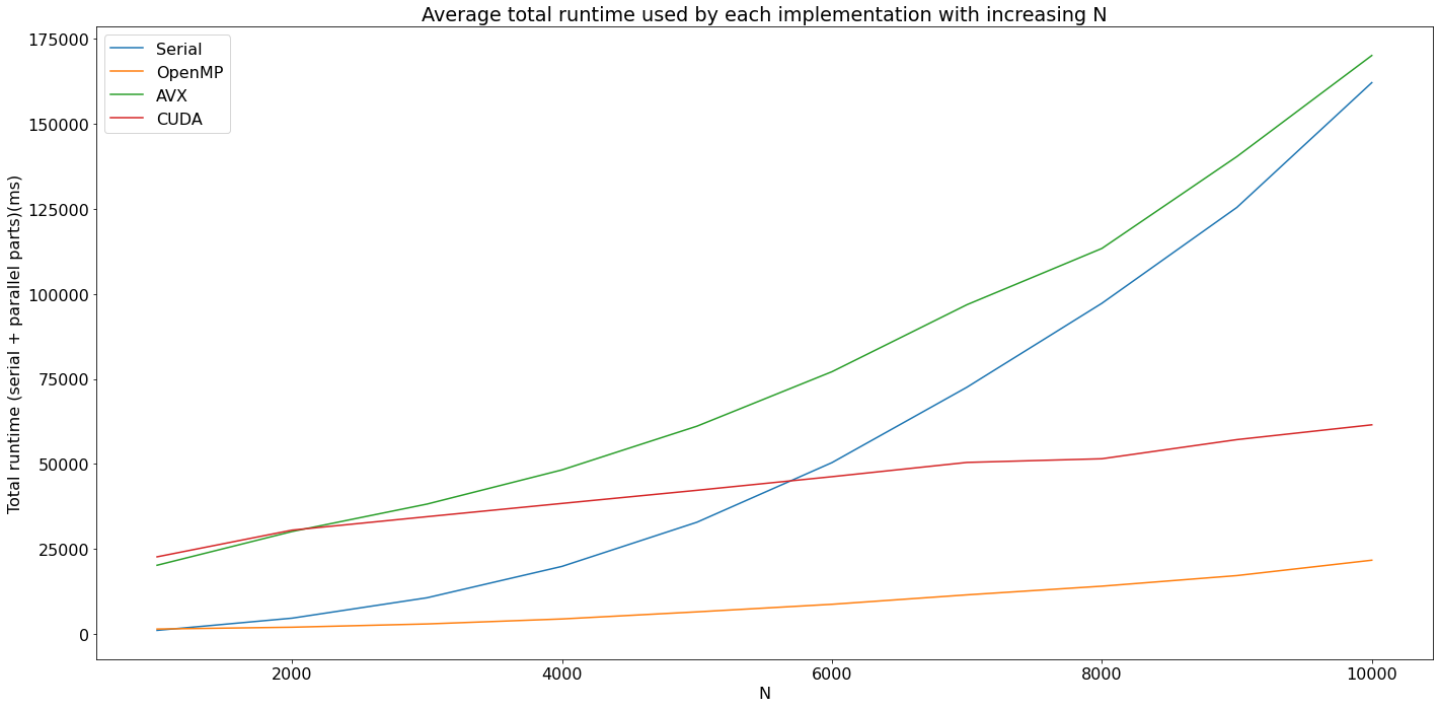

Figure 4: Average total runtime used by each implementation based on 10 runs for each of $N = 1000$ to $N = 10000$ in increments of 1000.

The diagram in figure 4 above is highly interesting as it roughly shows the problem size values where the serial implementation's overall runtime dominates other implementations' overall runtimes. Total runtime in this case is given by the sum of the times used for $MatrixVectorMultiply()$, the initialisation segment and the parts of the $eigensolver$ library that do not involve the parallelised function. In other words, figure 4 is given by adding figures 2 and 3 together. Observe that this figure reflects the high initialisation cost for the AVX and CUDA implementations but also shows the good scalability of the OpenMP and CUDA implementations for increasing problem size N.

Relation to Amdahl's and Gustafson's Laws:

For this section, the parallelisable fraction of the program is represented by $MatrixVectorMultiply()$ whereas the serial fraction is represented by the initialisation segment and the parts of the $eigensolver$ library that do not involve this parallelised function.

Amdahl's law states that if some fraction p of a program is parallelisable, then with a large number of processors, the unparallelisable (serial) fraction $(1-p)$ dominates the runtime. This theory supports viewing parallel performance by making a program run faster for a fixed problem size N^* (use N^* instead of N for this section to avoid confusing notation). The speedup formula is given by $S(N) = \frac{N}{N(1-p)+p}$ where N reflects the number of processors. Clearly as N gets very large, taking the limit, the theoretical maximum speedup is given by $\frac{1}{1-p}$ (upper bound). Ignoring the AVX implementation (since the packed doubles vector sizes are fixed and can't be increased unlike the notion of processors in Amdahl's law), it is expected that increasing the number of CPU threads used in OpenMP or GPU threads used in CUDA (respective equivalents of more processors) would produce a diagram similar to the one seen in figure 5 (next page) taken directly from the Week 8 lecture slides. In other words, the initialisation segment and parts of the eigensolver library that do not involve $MatrixVectorMultiply()$ would end up resulting in the maximum possible speedup for fixed N^*.

On the contrary, Gustafson's law states that if some proportion s of a program is unparallelisable, the speedup formula is instead given by $S(N) = N(1-s) + s$. This theory instead supports viewing parallel performance through scalable parallelism which is using additional processors to solve a bigger problem, usually within some fixed wall-time budget (e.g. 1 hour/day/etc.). The key assumption made here is that this fraction s does not grow with problem size N^* but instead remains fixed throughout. In this case, with a growing problem size and provided there are sufficient computational resources, increasing the CPU threads for OpenMP or GPU threads for CUDA would yield an almost linear relationship in terms of higher processors → higher speedup. This is because the serial portion is completely fixed – the resulting speedup is guaranteed to be increasing with increasing problem size N^* because both implementations are able solve much bigger problem sizes than a serial implementation could in the same fixed timeframe. Finally, this phenomenon is expected to appear very similar to figure 6 below which was provided in the Week 8 lecture slides.

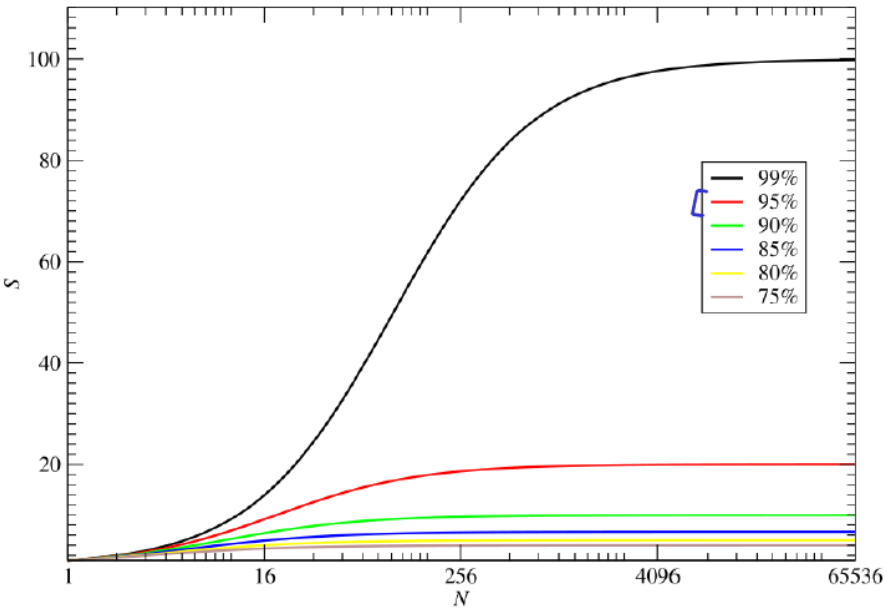

Figure 5: Amdahl's law in theory.

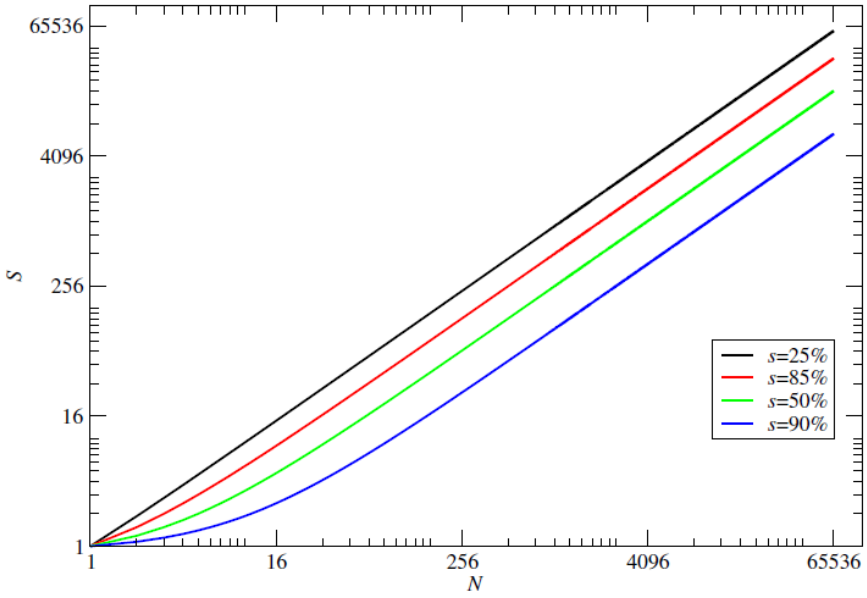

Figure 6: Gustafson's law in theory.