Gabor Hosszu
ID: 8070815

COMP35112: Chip Multiprocessors Lab1 – Report

Expectations & Observations

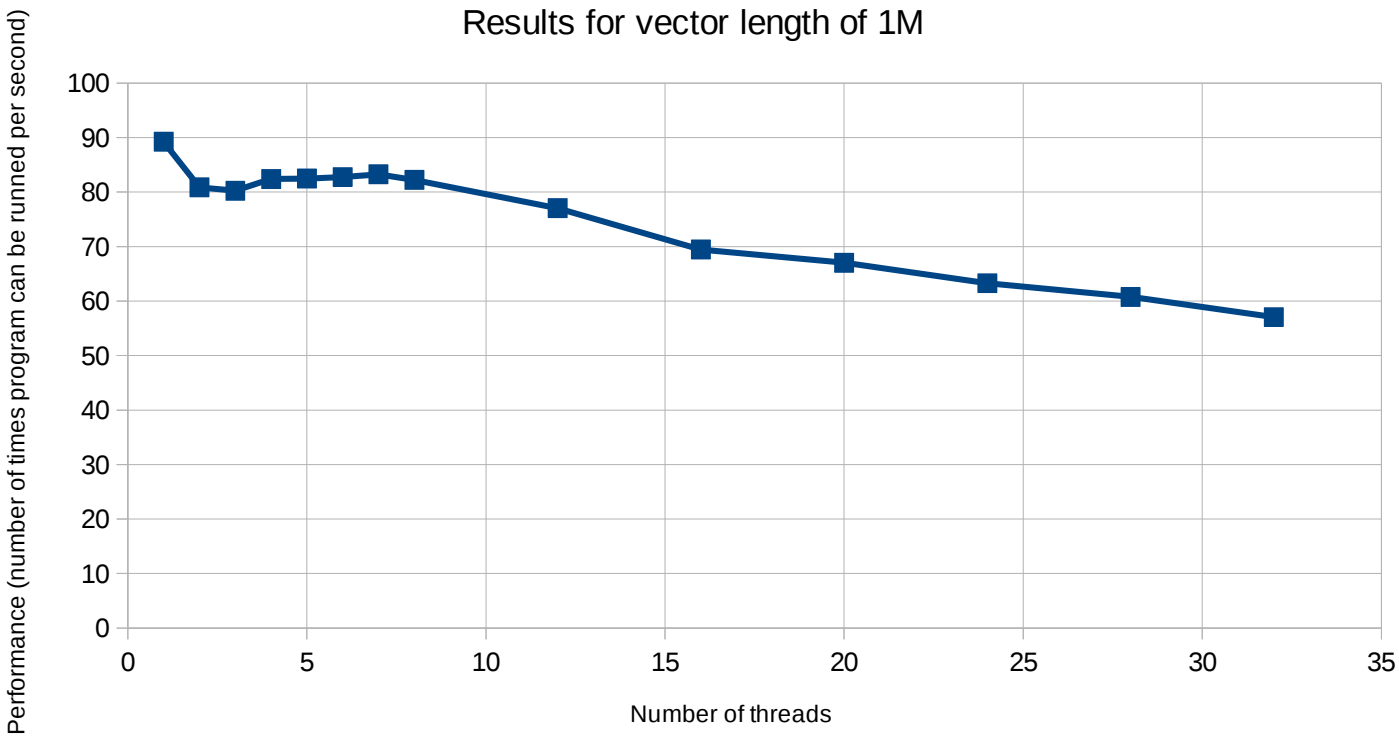
The first experiment I have tried was running the script 32 number of times with n threads every time and vector length constantly 32 (e.g. run1: 1 – 32, run2: 2 – 32, ..., run32: 32-32).

The expectations I had, have more or less come true, as running multi-threadded additions for a vector length this small has virtually no benefits whatsoever. Some might think think that if, for example, you run the addition with parameters 1 – 32 (threads – vector length) and then you run it with 2 – 32, you will 'basically' double your speed. This is clearly not true, as even if in the first case, 32 additions are executed iteratively and in the second one, 16 additions are executed in 2 threads, the operation is way too simple to benefit from this. This is what I have observed so far.

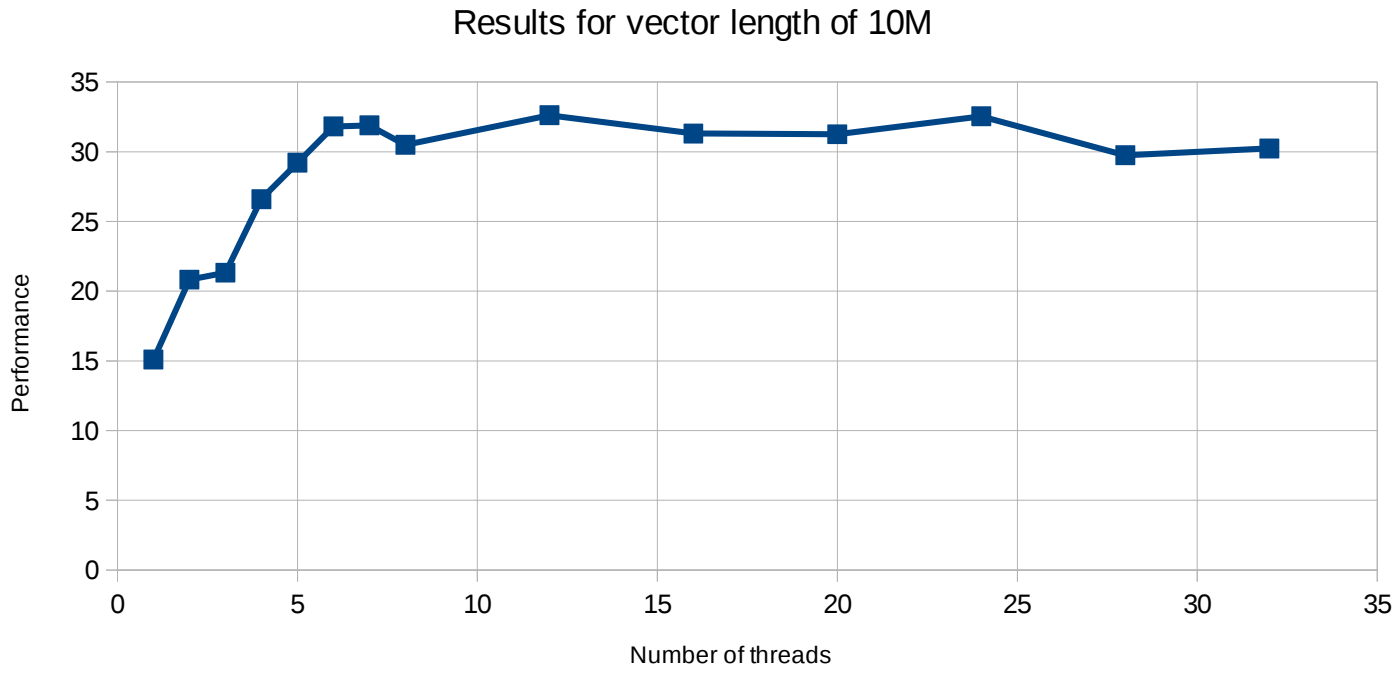
Now for the main experiment, I wanted to find the point where increasing the number of threads is actually useful. So I have started running multiple of 10 for the vector length, each time running it with almost all thread number possibilities (1 … 32). The point where things become 'interesting' are after vector sizes of 10 million. Therefore I ran 4 variations of the program around this value to capture the speed increase/decrease (1, 10, 100 and 1000 million for vector length). Following this are the results plotted on graphs. The recorded values are, of course, all averaged across multiple runs to decrease possible 'noise' from the system.

Gabor Hosszu
ID: 8070815

Performance

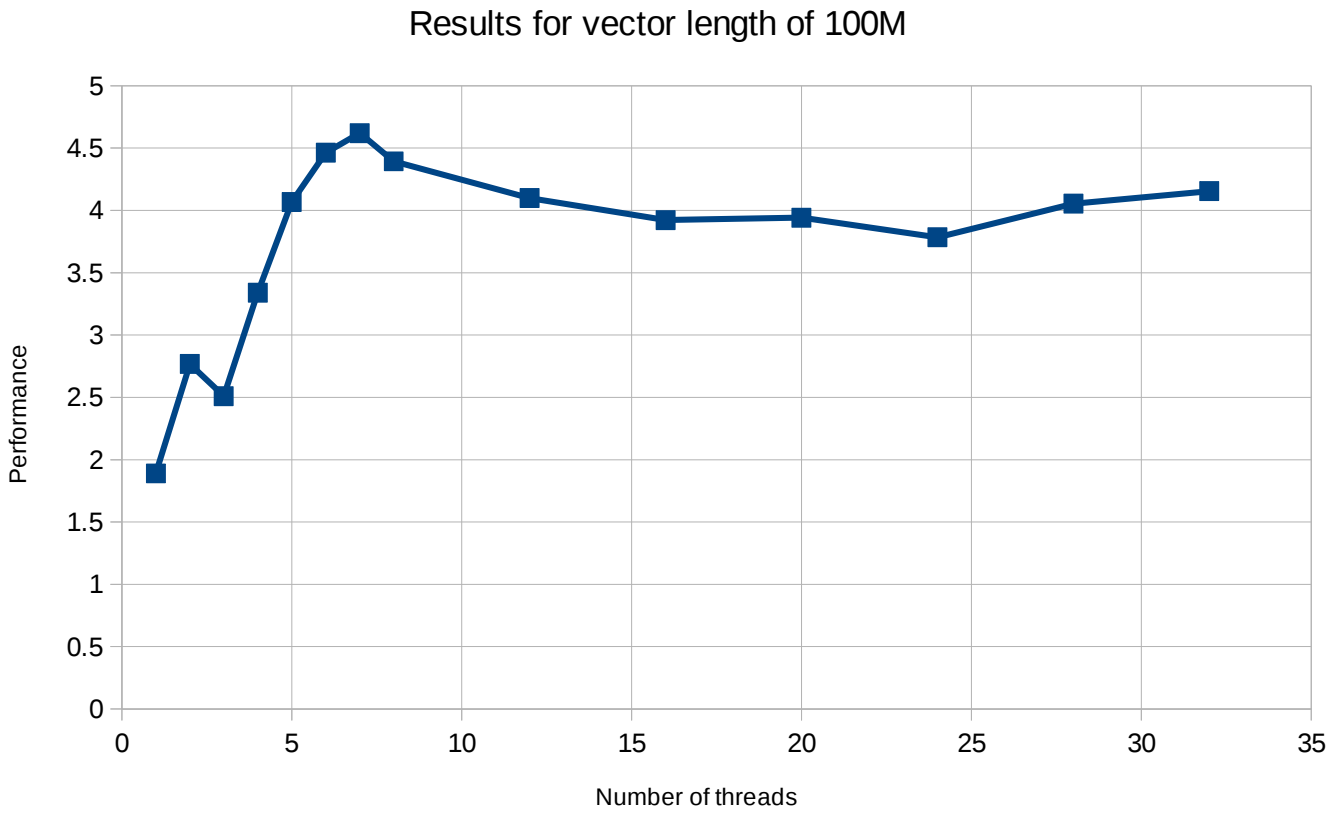


As you can see, for a vector length of 1 million, there is still virtually no benefit from multi-threading. In fact, the more threads are used, the slower the program gets. This is due to the overhead of creating threads.



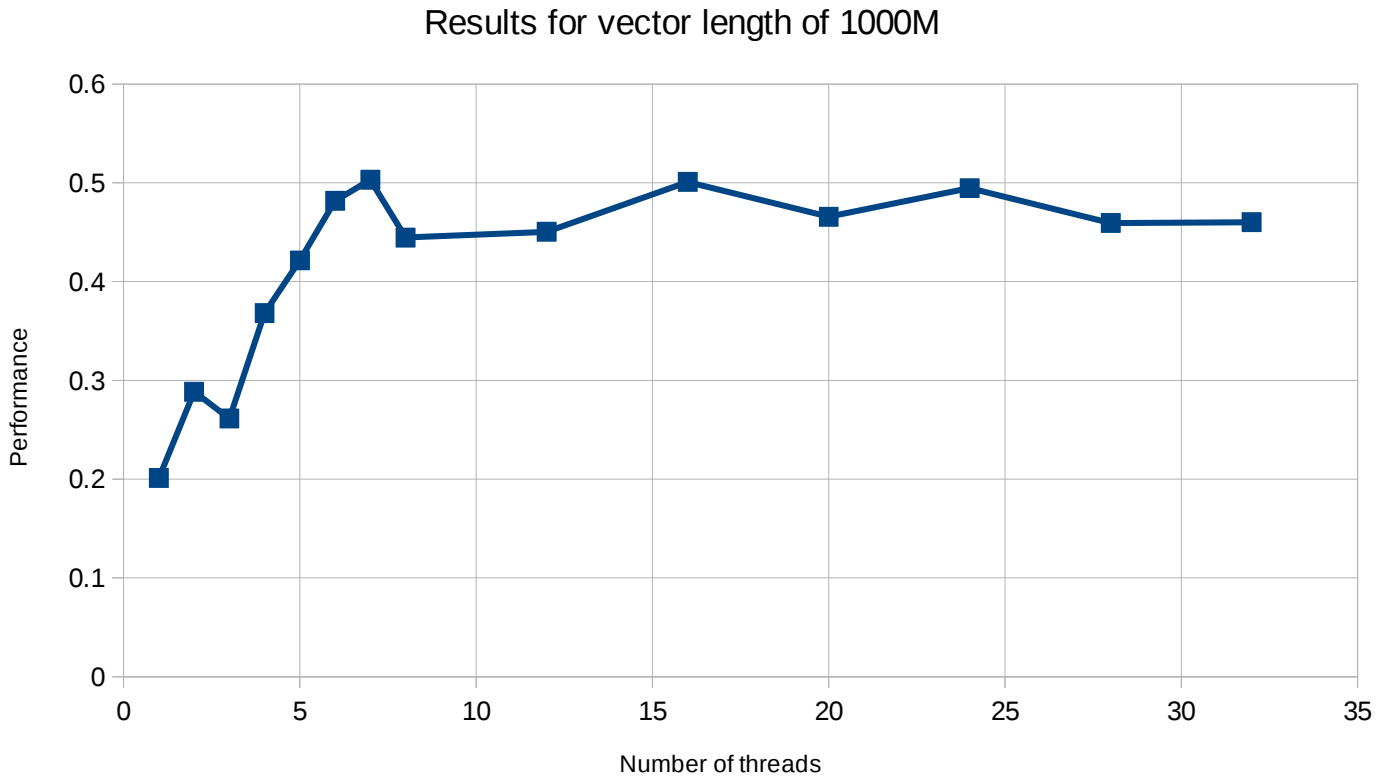
Gabor Hosszu
ID: 8070815

When we reach a vector length of 10 million, we notice the first significant increase in performance (of course, the actual precise 'sweet' point where multi-threading starts to have positive effect is probably somewhere between 1 and 10 million). An important observation to note here is that even though the program speeds up from multiple threads, there is still a limit on how many threads should be used before they become redundant again and have negative effects (although for vector sizes this big, the time creating threads is barely affecting the overall performance any more). From the graph, it can be seen that if we try to use more than 6 – 7 threads, the performance pretty much stays the same.



Same as above, threads become redundant if we try to user more than ~7 (maybe 8). The difference between this graph and the previous one is very small. This is probably due to the fact that in order for more than 7 threads to be useful, the vector size has to increase even more than just by a few powers of 10.

Gabor Hosszu
ID: 8070815



Same as for the previous graph, the turning point seems to be around 7 – 8 threads. Again, to observe an even bigger thread – performance ratio, we would need to go beyond a vector size of 1000M and that would make the execution time of the program way too long. Even for 1000M, the script takes a good 10 minutes to finish (running for 1, 2, 3, 4, 5, 6, 7, 8, 12, 16, 20, 24, 28, 32 threads at a time).

Finally, I would also like to note that a big factor influencing these results is the fact that what we define as the 'atomic' operation in this experiment is 1 addition. I suspect that the increase in performance could be seen a lot clearer and would be greater if I increased the number of additions executed per thread (e.g. for vector length of 1M, do 1M * 10000 additions, but the 10000 additions performed by the same thread every time; basically do the same addition 10000 number of times – a redundant operation logically, but helpful for observing the increase in performance caused by the increase in the number of threads used).