Clayton Walker

Programming Assignment 1

1.1 SAXPY GPU Execution Time

Below Figures 1 and 2 demonstrate the breakup of execution time for both GPU activities and API calls in the SAXPY GPU program, with the total execution times given at the top of each bar. Only portions that take up more than 1% of the execution time at 150k are shown in the graphs.

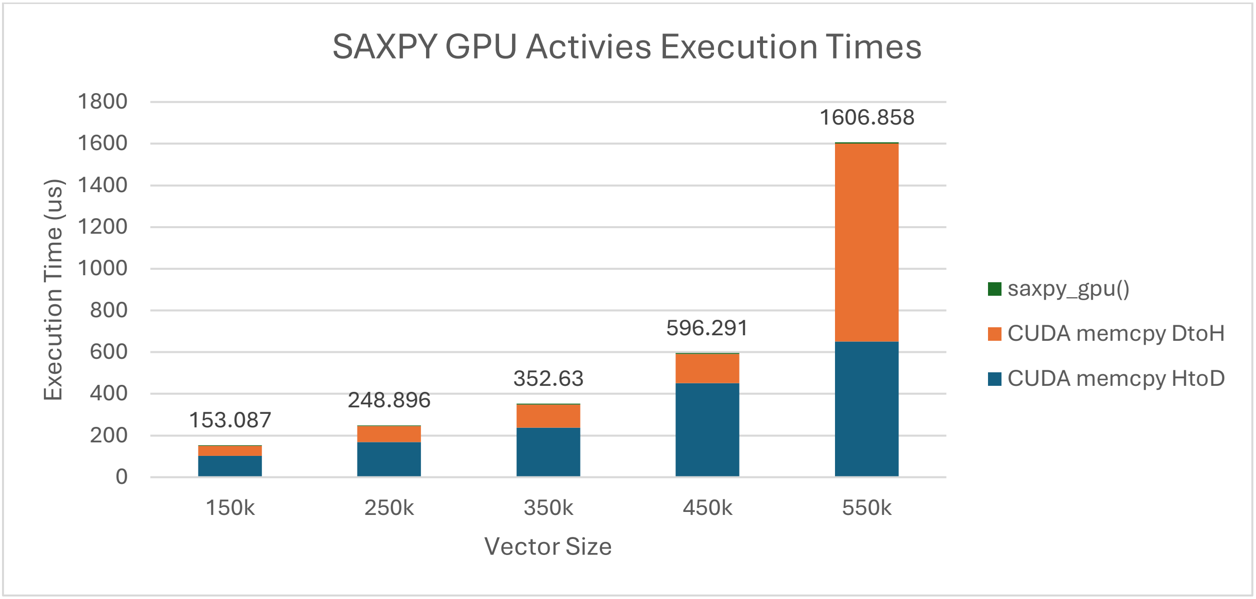

Figure 1: Execution time breakup of GPU Activities in SAXPY GPU

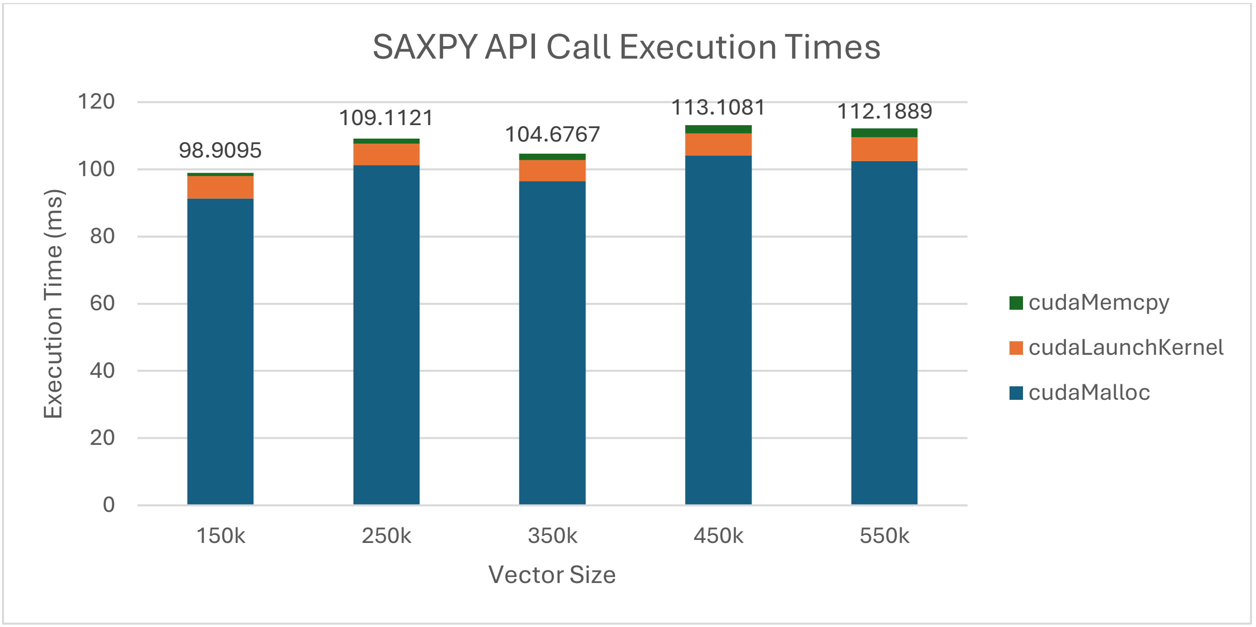

Figure 2: Execution time breakup of API Calls in SAXPY GPU

Analyzing these numbers reveals some interesting results. For one, the time required for API calls to execute is by far the largest contributor to the execution time for each program run, and this time does not seem to noticeably correlate to the vector size. This seems to show that setting up SAXPY on a GPU requires much more time than running the program on the GPU, and that the overhead for setting up the kernel is better amortized for larger vector sizes, where GPU execution time becomes a much larger factor of the total execution time.

Looking at GPU activities shows another interesting result; between 450k and 550k the device to memory copy sees a 6.5x increase in time, making it the largest contributor to execution time, despite there being two host to memory copies and only one device to memory copy. Running the profiler for even larger vector sizes seems to cause the execution split between DtoH and HtoD to even out at around 50-50. My guess for this behavior is that since calls to the Cuda API are asynchronous and memory copying takes a large majority of the time, the scheduler lets some threads continue as long as the vectors they are using are fully copied. This means that copying memory from the end of a very long output vector takes much longer, as that output has not finished computing yet, due to the memory of the input vector not being complete. For a less memory bound application that reuses more of the copied data this phenomenon would likely not occur. It is also important to note that the portion of the execution time taken by the actual kernel function saxpy_gpu() gets much smaller as the vectors get larger, confirming that this is a heavily memory bound application.

1.2 MCPi GPU Execution Time

Below are figures corresponding to the MCPi GPU execution times. For each figure, the default values are a sample size of 10M, 10k generation threads, and a reduction size of 300. Note that the number of reduction threads is always the number of generation threads divided by the reduction size, so increasing the reduction size generally decreases the number of reduction threads. Once again, portions of the execution time that account for less than 1% of the total time are not listed.

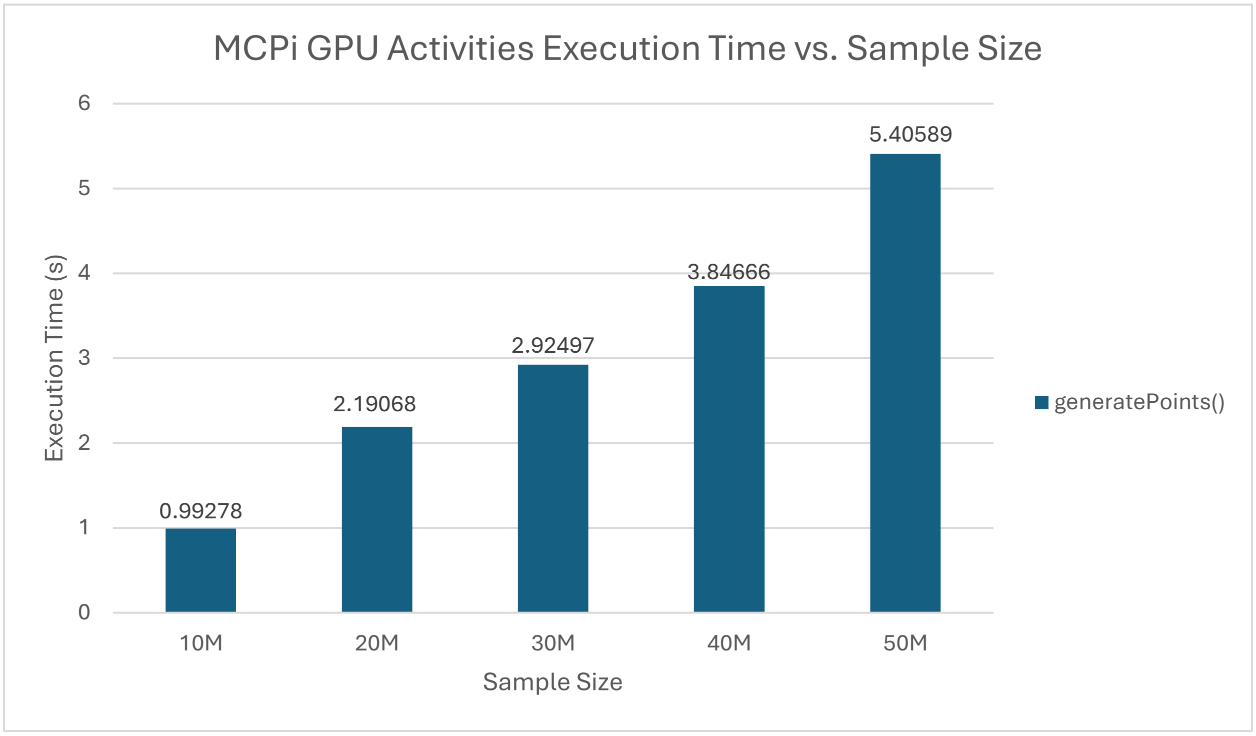

Figure 3: Execution time breakup of GPU activities in MPCi with varying sample sizes

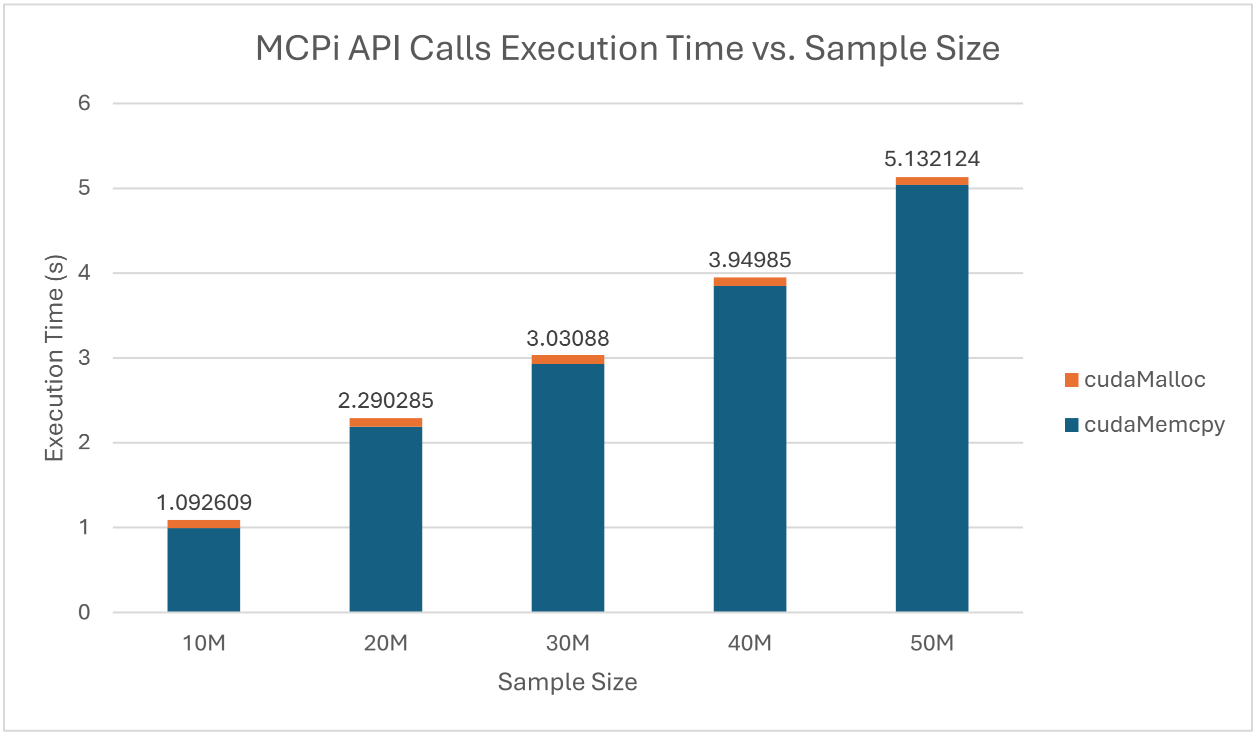

Figure 4: Execution time breakup of API calls in MPCi with varying sample sizes

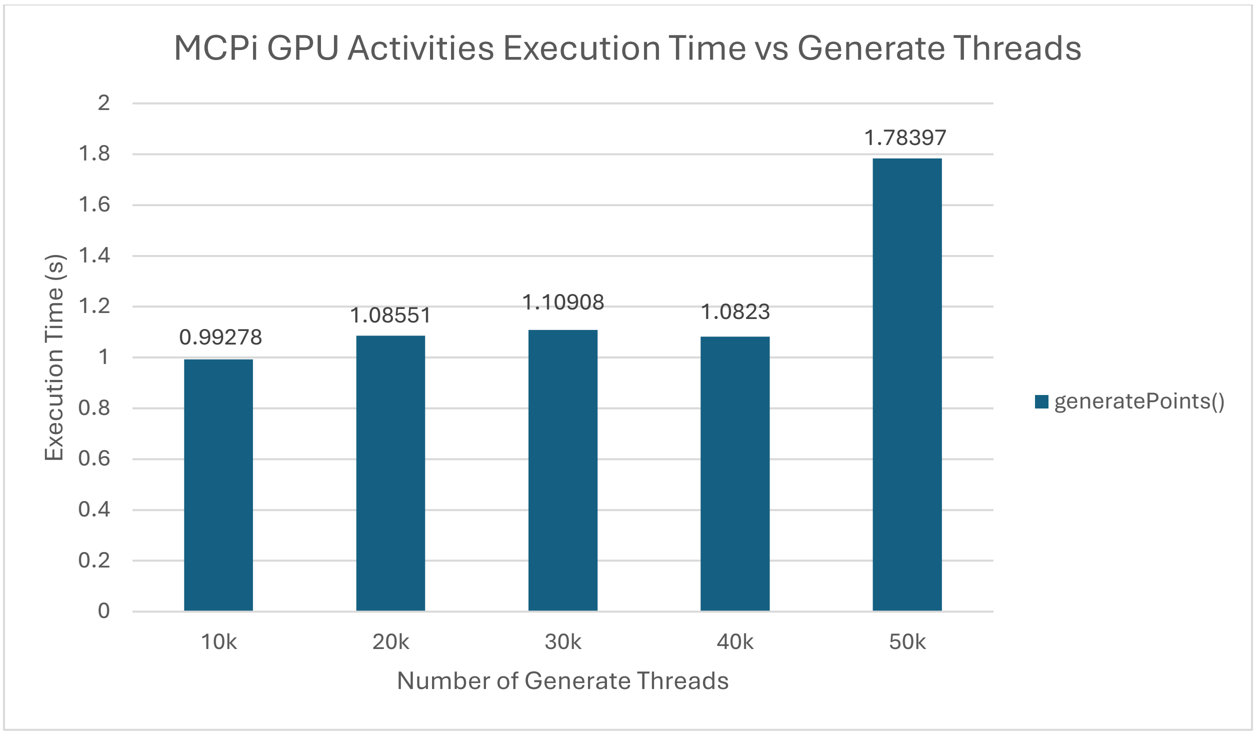

Figure 5: Execution time breakup of GPU activities in MPCi with varying numbers of generate threads

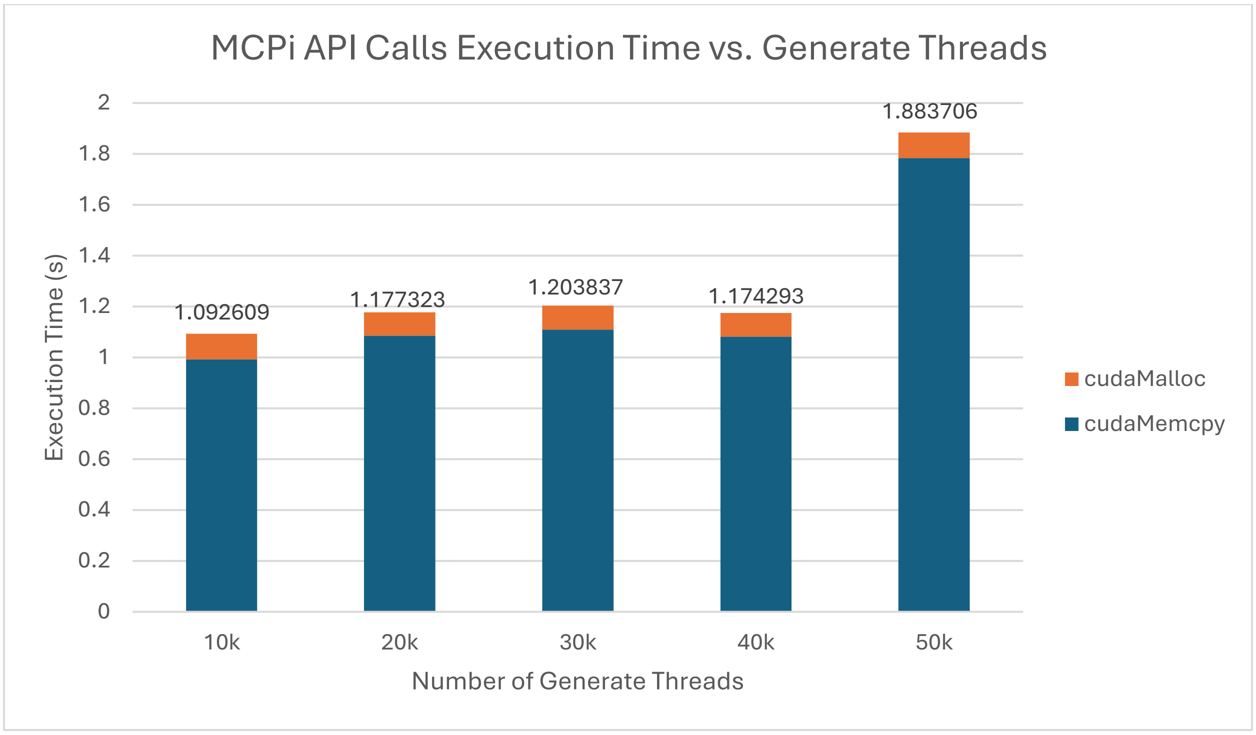

Figure 6: Execution time breakup of API calls in MPCi with varying numbers of generate threads

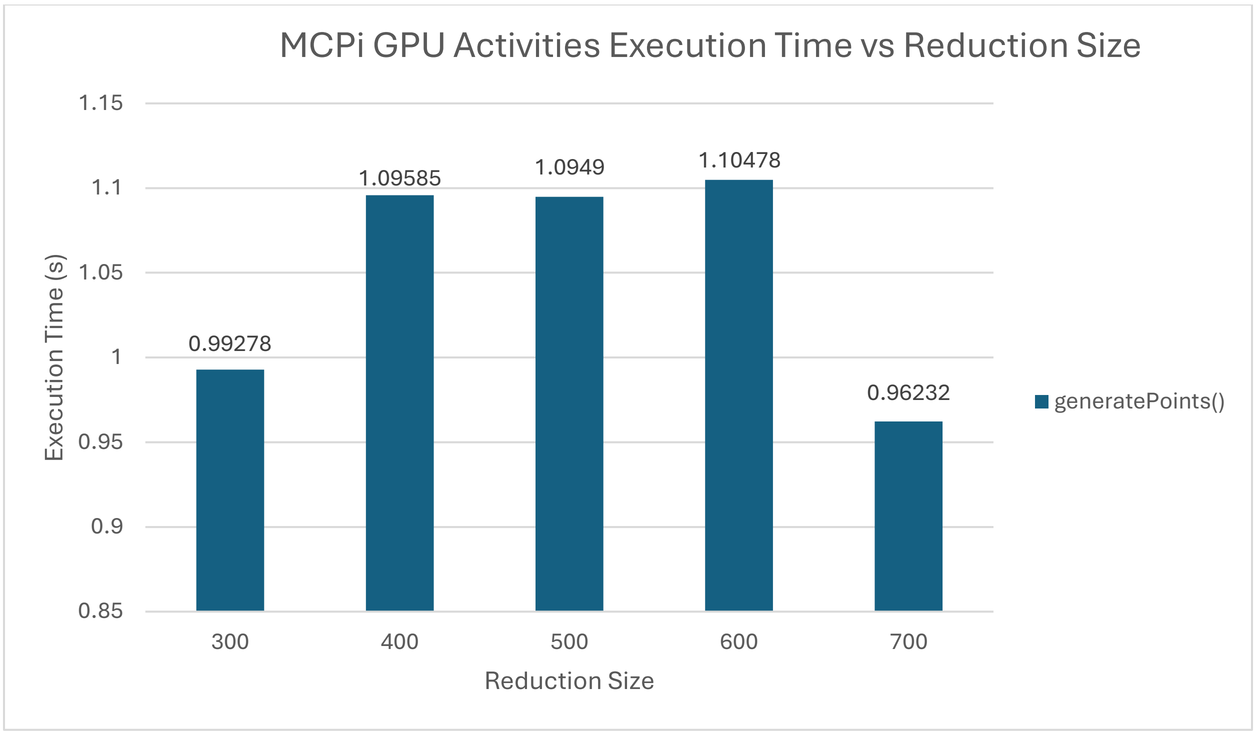

Figure 7: Execution time breakup of GPU activities in MPCi with varying reduction size

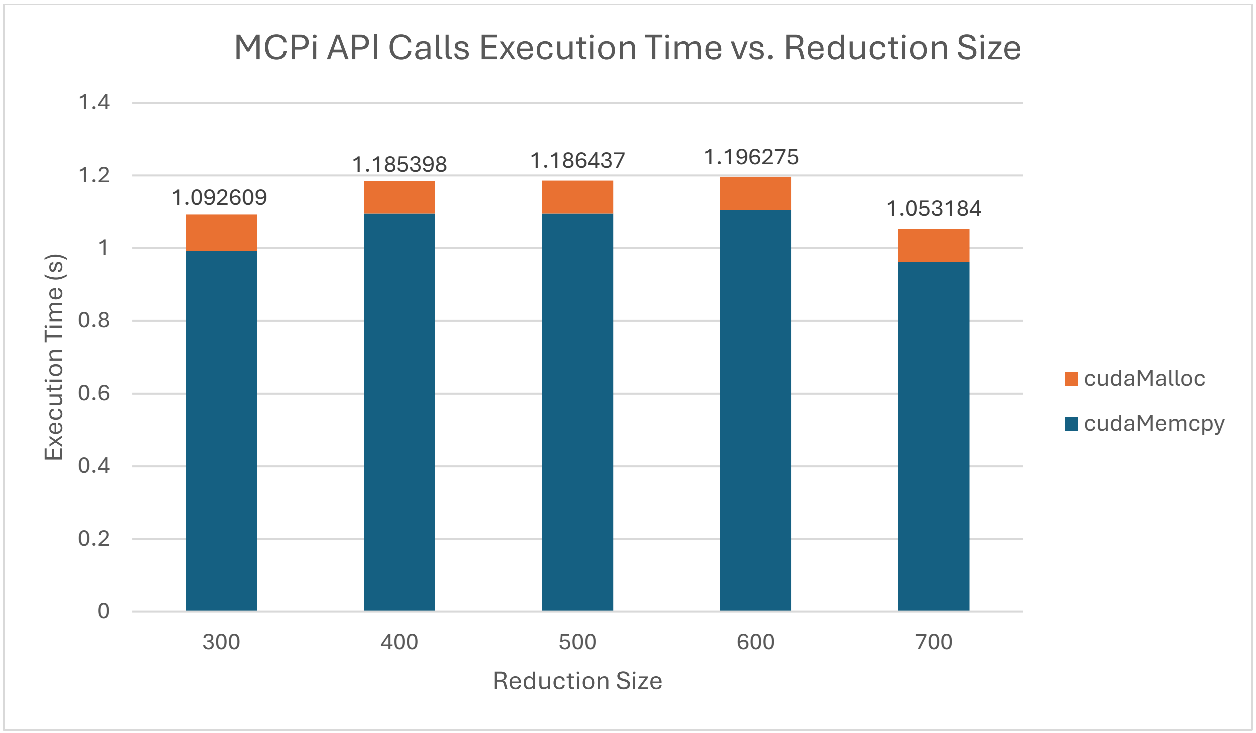

Figure 8: Execution time breakup of API calls in MPCi with varying reduction size

There is some interesting behavior to note from these execution time graphs. The first is that the only GPU activity that makes up more than 1% of the execution time is the generatePoints thread. This makes sense, as this thread does a lot of computation and there is very little memory copying. In fact, the only device memory that is copied to the host is the reduced sums array, which is much smaller than the partial sums array (by a factor of the reduction size). It then follows that increasing the sample size has a very large effect on the execution time, which seems to increase linearly with the sample size. On the API side, it seems that the execution time of cudaMemcpy is always nearly the same as the execution time of generatePoints(). This likely confirms my suspicion from section 1.1, that calls to the Cuda API are asynchronous, and that these calls adopt the execution time of other functions if they depend on that function. In this case, cudaMemcpy must wait for generatePoints to finish, and therefore its execution time is always slightly higher than generatePoints.

An anomaly occurs when increasing the number of generate threads from 40k to 50k. Before reaching this threshold, increasing the generation threads does not seem to have any effect on the execution time, which would make sense due to the parallel nature of the algorithm. However, reaching 50k threads drastically increases the execution time. My initial guess was that some sort of core or register limitation was reached at this point, but the GPU being used has the ability to handle much more than 50k threads at once, and each thread only uses 31 registers. It could just be that the memory bandwidth finally starts getting hit hard enough at this point, as this does cause the GPU to reach 694.34 MB/s according to the profiler.

One last observation is that the reduction size does not have a noticeable effect on the execution time. This partially follows from the fact that execution time is dominated by the generatePoints thread, so no matter how many points each reduction thread must reduce, it has little effect on performance. I decided to take this to an extreme by making the reduction size equal to the number of generation threads (10000) and found that the execution time still stayed at around 1.17 seconds. It is possible that with more generation threads it would reach a point where there are too many values for one reduction thread to handle, but for smaller values the execution time of generatePoints dominates everything else.