# 1 Introduction

For this project you are provided six versions of a simple N-body code – a serial CPU implementation, a multicore CPU implementation, and four slightly different GPU implementations. The codes all implement a simple direct $n^2$ N-body algorithm, where each particle (or *body*) exerts a force on all other particles that is inversely proportion to the square of their separation distance. At each timestep, the net force on each particle is computed, yielding its instantaneous acceleration, which can then be used to estimate its velocity at the next timestep (using a discrete value of time and the previous velocity). These details, though, are unimportant for the standard performance analyses required in this exercise. The key point is that N-body is much closer to a realistic computational kernel than some of the loop structures we have analyzed to this point.

The four GPU implementations carry out identical computations but include data layout and/or operation order optimizations that may have an impact on performance. Your job is to analyze these codes using concepts and tools that we have learned throughout the course, and to provide the data and response to a series of questions in the form of a short report. Think of it as an expanded version of a weekly exercise. No collaboration is permitted, and some concepts might force you to go slightly beyond topics directly discussed in lecture. Any texts or online resources are permitted. The report must address each of the following questions in order.

1. Run each version of the code and fill out the table shown below. Use $n = 100,000$ and 20 timesteps (iterations). For the CPU use `gcc -O2`. For the multicore CPU, report the fastest multicore configuration (number of threads, OMP schedule, etc.) but be sure to report times as averages over at least 5 executions for the optimal parameters. Similarly, for the GPU, report the average time of the fastest configuration (threads-per-block, etc.) that you find. As always, use a dedicated (exclusive) partition and include a description of the processor, compiler version and options, and CUDA library version.

| Metric\Code | **Serial** | **Multicore** | **GPU1** | **GPU2** | **GPU3** | **GPU4** |
|---|---|---|---|---|---|---|
| Time to solution | | | | | | |
| Grind rate (time/iteration) | | | | | | |
| GLFOP/s | | | | | | |
| % time in bodyForce() | | | | | | |
| Peak theoretical performance | | | | | | |
| % of peak obtained | | | | | | |

2. Analyze the time complexity of the main loop (big-O notation) and show a graph of execution time vs. problem size verifying your analysis.

3. Estimate the arithmetic intensity of the main bodyForce loop in FLOPs per word. In other words, how many words are read/written from fast memory per floating point operation, making simplifying assumptions about cache (base analysis just on L1). How does your paper calculation compare to CPU measurements using PAPI counters for FLOPs and memory operations?

4. Calculate the maximum GPU speedup relative to the CPU. Lookup rough pricing information and determine which system has higher performance/dollar.

5. Given the data in your table above, compute the Amdahl ceiling when parallelizing just the *bodyForce* routine.

6. Recompute the Amdahl ceiling for $n = 500,000$.

7. Estimate if the serial code is compute or memory bound. Explain your approach.

8. Re-run the CPU code using the Intel compiler and compare the performance. Speculate on any discrepancies.

9. Use opt-report with the Intel compiler to verify that CPU vector instructions are generated in the main loop. If so, what is the observed performance difference between vectorized and non-vectorized version (i.e. suppress vectorization and compare). If not, can you force vectorization? Why or why not?

10. Measure and display in a table the L1/L2/L3 hit rate for the two CPU codes.

11. For the serial CPU code, compute the fraction of branch mispredictions. Comment on how branching might affect performance in the code.

12. For the serial code, compute the TLB hit ratio. Comment on whether TLB performance is likely to have a significant impact on overall code performance.

13. For the multicore code, report on the type of hardware multithreading your code uses. Report on how much performance improvement, if any, you gain by using hardware threads.

14. For the multicore code, measure/calculate the memory bandwidth utilized as a function of thread count. At what thread count does memory bandwidth saturate?

15. For the multicore code, calculate the number of threads for which the code achieves max parallel efficiency. Speculate on why the performance flattens or decreases at higher thread counts.

16. For the multicore code, plot the parallel speedup for

    - strong scaling
    - weak scaling

17. Compare the per core L1 miss rate of the multicore to the serial code. Speculate on the source of any discrepancies.

18. For the GPU code, briefly explain each optimization in version 1 - version 3.

19. For each GPU version, compute the fraction of the total execution time taken by the host/device copy for a few problem sizes ranging from $n = 1000$ up to the $n = 500,000$

20. Compare the GPU occupancy for each version. To what degree does occupancy explain observed performance differences?