Coursework: Performance Modelling, Vectorisation and GPU Programming

Shazzad Hasan

(1.1)

Function	Floating Point Operations	Data Accesses (Loads and Stores)	Operational Intensity (flops/ byte)
$function\_a()$	$2N^2$	$4N^2 + N$	$\frac{2N^2}{8\times(4N^2+N)}$
$function\_b()$	$2N$	$3N$	$\frac{2N}{8\times(3N)}$
$function\_c()$	$1.5N$	$3N$	$\frac{1.5N}{8\times(3N)}$
$function\_d()$	$2N$	$2N+1$	$\frac{2N}{8\times(2N+1)}$

(1.2)

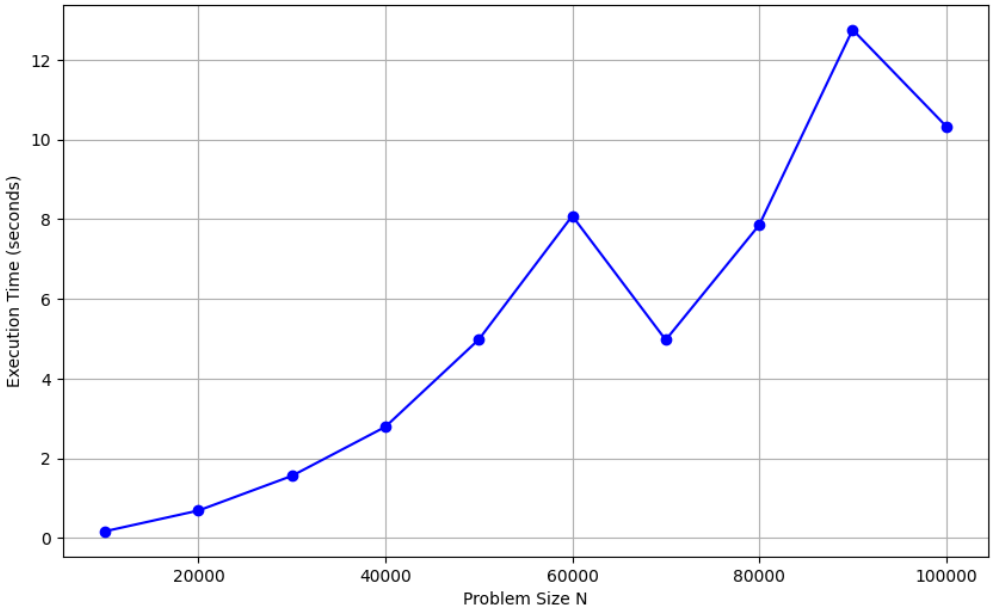


Figure 1: Execution Time vs Problem Size N.

Figure 1 depicts the overall execution times as problem size, N increases. Overall, the execution time data from profiling indicates a general trend of increasing runtime as N increases, which is the expected typical behavior for computationally intensive tasks. However, there is an unexpected decrease in execution time at N=70,000 before it rises again, which might be due to the temporary alleviation in computational or memory access patterns, or it could caused by external factor e.g., system load or resource availability during the profiling.

The characteristics of Figure 1 are closely related to the computational characteristics of the functions: function_a, function_b, function_c, and function_d, based on their data accesses, floating point operations, and operational intensity. The function_a dominates in computational demand due to its $2N^2$ floating point operations and the significant increase in $4N^2 + N$ data access as N increases, resulting in a decrease in operational intensity. This suggests that the function becomes more memory-bound due to increased data

transfer relative to the amount of computation, which has been reflected in the observed longer execution times for larger N.

The other functions, function_b, function_c, and function_d, have low operational intensities. Their impact on execution time is moderate and increases linearly with N. To summarize, function_a is the hotspot function i.e. the primary driver of execution time, especially for larger N.

(1.3) The likwid profiling of the hotspot function, function_a for both memory and floating point performance gives insights into its computational behavior on an AMD EPYC 7702 processor. The operational intensity for function_a is 0.2448 FLOPs/Byte, demonstrating that the function is moderately memory-bound. It also indicates significant memory bandwidth usage (8733.1675 MBytes/s) compared to its floating-point computation throughput (2137.9748 MFLOP/s), suggesting that the function's performance is likely limited by data transfer rates rather than computational capabilities.

Efficiency metrics, for instance, a low CPI (0.3894) indicate that the CPU efficiently processes instructions, pointing towards effective CPU utilization while highlighting a bottleneck in memory access patterns. The profiling also confirms that 200 million FLOPs were performed. The consistency between the MEM_DP and FLOPS_DP groups in terms of retired instructions and floating-point operations confirms precise measurement and highlights the necessity for optimizations to decrease memory latency and increase cache efficiency.

(1.4)

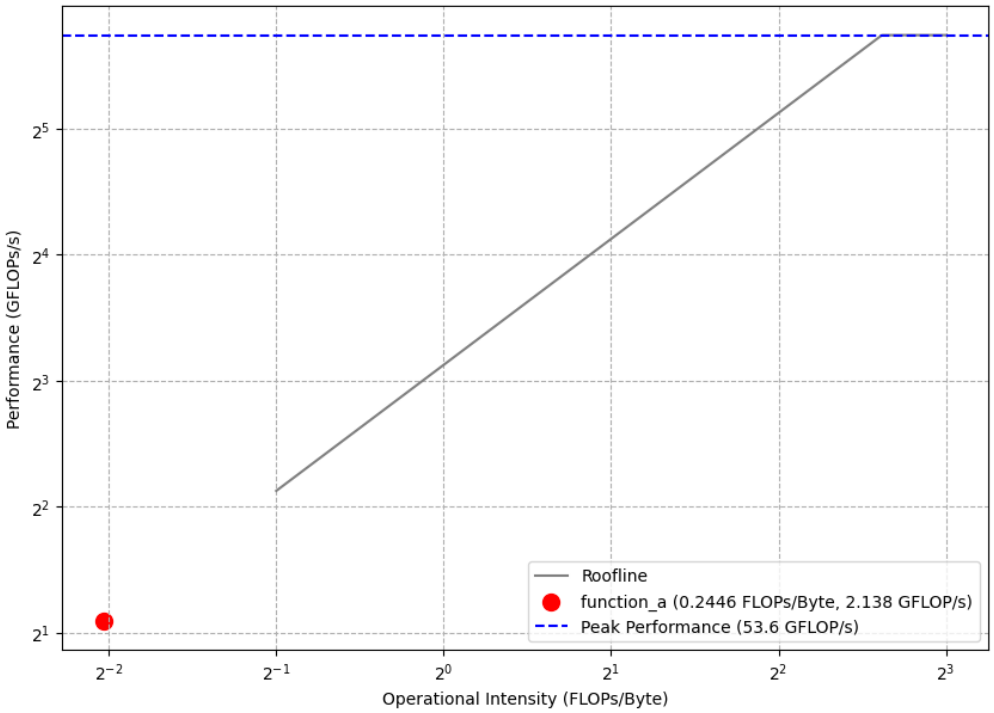
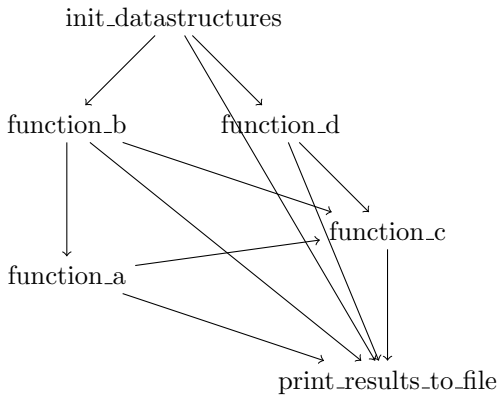


Figure 2: Roofline Model.

(2.2)



(2.4)

To implement a GPU implementation of the `number_crunching.cpp` program, I used CUDA programming module because of its suitability for computationally intensive tasks such as those present in this program. The reason behind choosing CUDA rather than OpenMP and SYCL is that CUDA is specifically optimized for NVIDIA GPUs, which makes it particularly effective for exploiting the parallel power of NVIDIA hardware. In contrast, OpenMP and SYCL don't offer the same level of control or optimization for GPU as CUDA. Moreover, CUDA provides fine-grained control over both memory management and parallel execution.

Initially, I implemented a CUDA program that exploits loop parallelism by translating computational loops from the CPU code into GPU kernels that execute across many threads, effectively utilizing the GPU for operations like matrix arithmetic present in the source code. Each kernel handles a portion of the loop, allowing for the simultaneous processing of multiple data elements, which is especially beneficial for large data sets. Then, I extended the CUDA program to make use of task parallelism by employing CUDA streams to run independent kernels concurrently. This approach not only parallelizes the loops within each kernel but also overlaps the execution of different kernels, which can significantly reduce overall runtime by optimizing GPU and memory usage.

Before computation began, I initialized data (u, v, A, x, y, z) on the host and transferred to the device. Memory for these data is allocated on the GPU using `cudaMalloc`. After computation, results are transferred back to the host using `cudaMemcpy`. These steps ensure that the GPU has immediate access to the data it needs without waiting for ongoing transfers, thus reducing idle time and increasing overall efficiency

I implemented separate kernels for each independent computational task and each kernel was optimized for GPU execution. I have chosen 256 as block size (i.e. 256 threads per block) and calculated grid size dynamically using the formula $(N+255)/256$. This ensures that there are enough blocks to cover all elements when N is not a multiple of the block size. Then I used CUDA stream to facilitate the overlapping execution of independent kernels. I created separate CUDA streams for each independent kernel and executed them in their respective streams, allowing for simultaneous data processing. To ensure, dependencies between data, particularly where output from one kernel serves as input to another, I used CUDA `cudaStreamSynchronize` function.

Figure 3 illustrates the runtime of sequential implementation and performance gains from GPU acceleration. The program was executed with different problem sizes, N. The sequential version outperforms both CUDA versions for N = 1000. This can be attributed to the overhead of GPU operations, such as memory allocation and data transfer. However, the CUDA versions significantly outperform the sequential version at larger problem sizes, i.e., N = 100000, showing substantial time savings. Notice the loop parallelism version slightly outperforms the task parallelism version, potentially due to the overhead of managing multiple streams or inefficiencies in overlapping execution, which could be improved further.

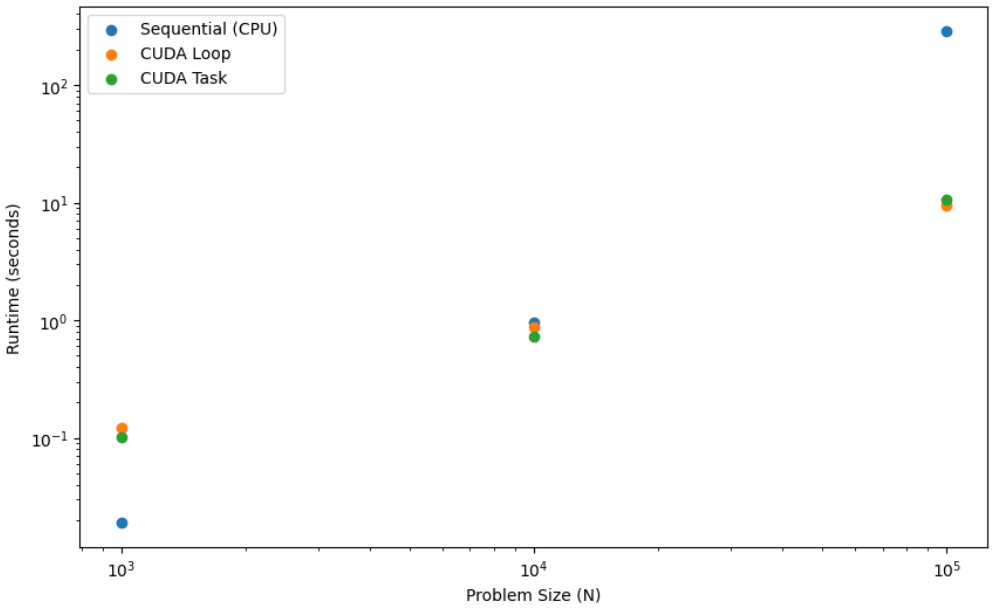


Figure 3: Runtime Comparison of Number Crunching Implementations.