Performance Modelling, Vectorisation and GPU Programming Coursework

Unlocking the Power of Parallel Computing: Optimizing Performance through GPU Acceleration and Vectorization

Teng Wang
001119820
dgwz78

A technical report presented for
Module COMP52315
Performance Modelling, Vectorisation and GPU Programming

Contents

1 Question 1 **3**

1.1 Best operational intensity 3

1.1.1 function_a 3

1.1.2 function_b 3

1.1.3 function_c 3

1.1.4 function_d 4

1.2 Profiling and Hotspot Identification 4

1.3 Detailed Memory and Performance Profiling 5

1.4 Roofline Model Visualisation and Optimisation Discussion 6

2 Question 2 **7**

2.1 GPU Implementation with Loop Parallelism 7

2.2 Task Graph and Task Parallelism 8

2.3 Approach and Optimisation Techniques 8

1 Question 1

1.1 Best operational intensity

Table 1: Code Analysis and Performance Metrics

Function	**Floating-point operands** M	**Data movement** B **(bytes)**	**Operation intensity** I_c
function_a	$2N^2$	$8(3N^2 + N)$	0.0833
function_b	$2N$	$24N$	0.0833
function_c	$1.5N$	$24N$	0.0625
function_d	$2N$	$16N + 8$	0.1249

After the manual calculation using a calculator, the results of the performance analysis are presented in the table 1. The detailed calculation process will be explained below:

1.1.1 function_ a

- Total number of floating-point operations: $2N^2$ (each of addition and multiplication performed N^2 times)
- Total number of data accesses: $3N^2 + N$ (reading $A[i \times N + j]$ and $x[i]$ a total of $2N^2$ times, writing to $y[i]$ $N^2 + N$ times, including initialisation to zero N times)
- Data movement amount: $8 \times (3N^2 + N)$ bytes
- Computational intensity I_c: $\frac{2N^2}{8\times(3N^2+N)} \approx 0.0833$

1.1.2 function_ b

- Total number of floating-point operations: $2N$ (each of addition and multiplication performed N times)
- Total number of data accesses: $3N$ (reading $u[i]$ and $v[i]$ a total of $2N$ times, writing to $x[i]$ N times)
- Data movement amount: $8 \times 3N$ bytes
- Computational intensity I_c: $\frac{2N}{8\times3N} = 0.0833$

1.1.3 function_ c

- Total number of floating-point operations: $1.5N$ (addition N times, multiplication only when i is even, i.e., $N/2$ times)
- Total number of data accesses: $3N$ (each iteration reading $x[i]$ and $y[i]$, writing to $z[i]$)
- Data movement amount: $8 \times 3N$ bytes
- Computational intensity I_c: $\frac{1.5N}{8\times3N} = 0.0625$

1.1.4 function_d

- Total number of floating-point operations: $2N$ (each of addition and multiplication performed N times)
- Total number of data accesses: $2N + 1$ (reading $u[i]$ and $v[i]$ a total of $2N$ times, accumulated to variable s, initialised to zero once)
- Data movement amount: $8 \times (2N + 1)$ bytes
- Computational intensity I_c: $\frac{2N}{8\times(2N+1)} \approx 0.1249$

1.2 Profiling and Hotspot Identification

The code was profiled using gprof to determine the overall execution time and identify the hotspot function(s) that require the most execution time. The profiling was performed for input sizes $N = k \times 10^4$, where $k = 1, 2, 3, \ldots, 10$. The execution times for each function are plotted in Figure 1.

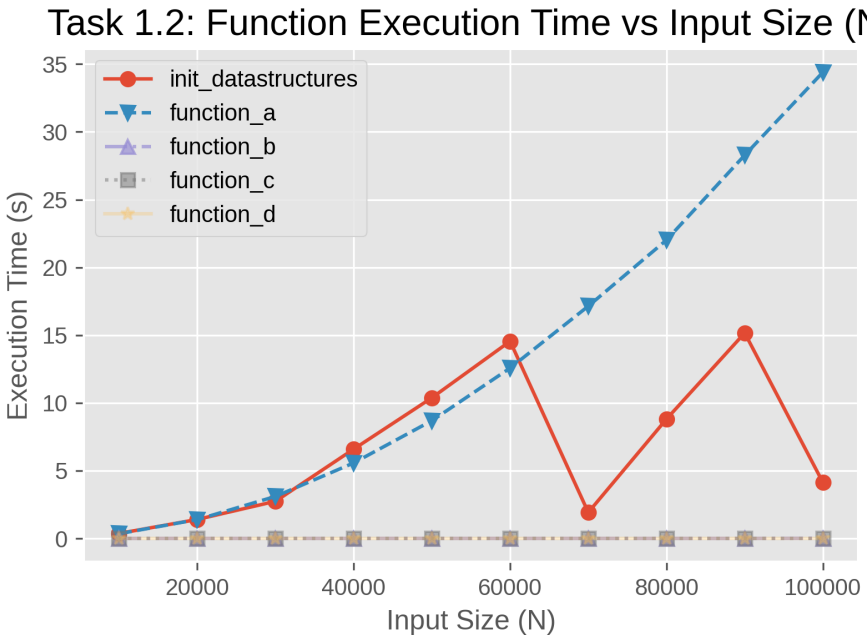

Figure 1: Function execution time as a function of input size N

From the profiling results, it is evident that `function_a` and `init_datastructures` are the hotspot functions, consuming the majority of the execution time. As the input size N increases, the execution time of these functions grows significantly, while the other functions `function_b`, `function_c`, and `function_d` have negligible execution times in comparison.

The profiling output for the largest input size $N = 10^5$ is provided in the file `number_crunching_gprof.out`. The relevant section of the output is shown in Table 5:

The profiling results confirm that `function_a` and `init_datastructures` are the hotspot functions, accounting for 88.73% and 10.66% of the total execution time, respectively, for the largest input size.

These findings align with the operational intensity analysis from Task 1.1. `function_a` has the highest computational complexity of $O(N^2)$ and performs the most data movement, resulting in the lowest operational intensity among the functions. This explains its dominant contribution to the execution time.

`init_datastructures` also has a significant impact on the execution time due to the memory allocation and initialisation of the large arrays `A`, `u`, and `v`. The other functions have lower computational complexities and perform less data movement, resulting in minimal execution times.

Based on these profiling results and the operational intensity analysis, optimising `function_a` and `init_datastructures` should be the primary focus for improving the overall performance of the code on CPUs.

1.3 Detailed Memory and Performance Profiling

Tables 8, 9, 2, 10, and 3 present the detailed memory and performance profiling results for the `function_a` hotspot using the likwid tool. The profiling was conducted for an input size of $N = 10^4$.

Table 2 displays the floating-point operations performance. The DP MFLOP/s metric indicates the number of double-precision floating-point operations per second. The observed value of 580.7757 MFLOP/s suggests that the code is not compute-bound and may benefit from optimisations to increase the floating-point performance.

Table 2: Floating Point Operations Performance

Metric	**Value**
Runtime (RDTSC) [s]	0.3444
Runtime unhalted [s]	0.5741
Clock [MHz]	3326.6421
CPI	0.3566
DP [MFLOP/s]	580.7757

Table 3 presents the memory performance metrics. The memory bandwidth of 2456.5124 MBytes/s indicates the rate at which data is transferred between the main memory and the processor. The memory data volume of 0.8460 GBytes represents the total amount of data read from or written to the main memory during the execution of `function_a`.

The data performance metrics are presented in Table 9. The load-to-store ratio of 14.9975 reveals that the code performs significantly more load operations compared to store operations. This high ratio suggests that the code is memory-read intensive.

Table 8 shows the cache performance metrics. The data cache request rate is 0.5320, indicating that approximately half of the instructions generate a data cache request. The data cache miss rate and miss ratio are extremely low, suggesting that the cache is being effectively utilised, and most of the data accesses are satisfied by the cache.

Table 3: Memory Performance

Metric	Value
Runtime (RDTSC) [s]	0.3444
Runtime unhalted [s]	0.5731
Clock [MHz]	3320.5919
CPI	0.3561
Memory bandwidth [MBytes/s]	2456.5124
Memory data volume [GBytes]	0.8460

The L3 cache performance metrics are shown in Table 10. The L3 cache access rate is relatively low at 0.4001%, indicating that only a small portion of the data accesses reach the L3 cache. However, the L3 miss ratio is extremely high at 98.4345%, suggesting that almost all L3 cache accesses result in misses. This high miss ratio implies that the data accessed by the code has poor temporal locality and does not fit well in the L3 cache.

The profiling results provide valuable insights into the memory and performance characteristics of the hotspot function. The low data cache miss rate and high L3 cache miss ratio suggest that the code has good spatial locality but poor temporal locality. The memory-read intensive nature of the code, as indicated by the high load-to-store ratio, combined with the low floating-point performance, indicates that the code is likely memory-bound rather than compute-bound.

To optimise the performance of `function_a`, improve data reuse and temporal locality by restructuring the code is a feasible way to maximise cache utilisation and minimise data movement between the cache and main memory. Another method is vectorise the code using SIMD instructions to exploit data-level parallelism and increase the floating-point performance. Additionally, optimise memory access patterns can reduce cache misses and improve memory bandwidth utilisation. By applying these optimisation techniques, it may be possible to alleviate the memory bottlenecks and improve the overall performance of the `function_a` hotspot.

1.4 Roofline Model Visualisation and Optimisation Discussion

The roofline model for the hotspot function (`function_a`) reveals that it is severely memory-bound, with performance limited by available memory bandwidth rather than computational capabilities. The function's operational intensity is approximately 0.1 FLOP/byte in Figure 2, and its observed performance is 580.7757 MFLOP/s.

To optimise `function_a`, data access patterns should be changed to maximise cache utilisation and minimise data movement between cache and main memory. Techniques such as loop tiling, data layout transformations, data reuse optimisations, and eliminating redundant memory accesses can help. Increasing floating-point operations per byte accessed, using SIMD instructions for parallelisation, and implementing software prefetching to hide memory latency are also recommended.

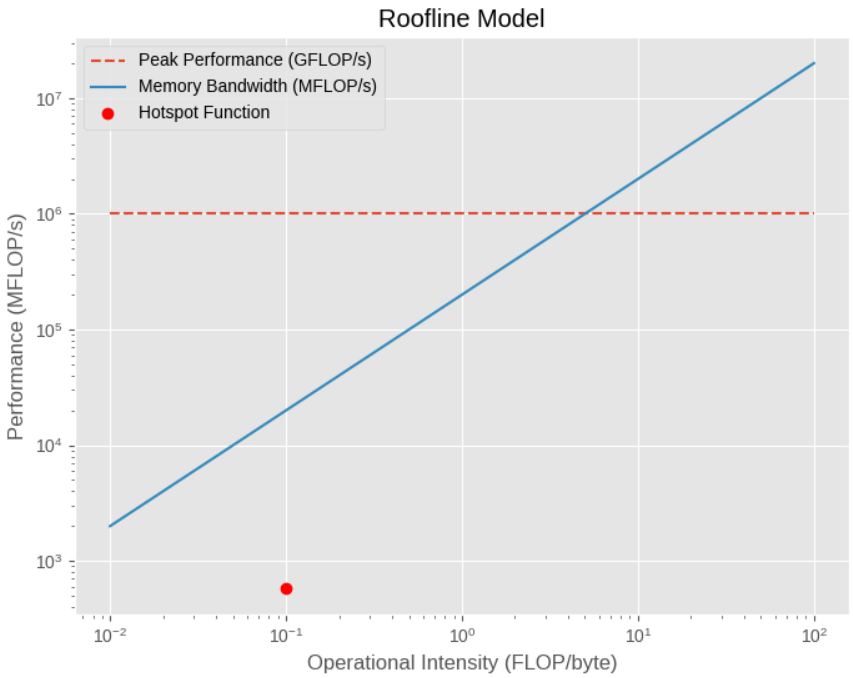

Figure 2: Roofline Model indicating performance limitations for the hotspot function.

By applying these optimisation techniques, the function's position on the roofline model can be shifted towards the right, increasing its operational intensity and bringing it closer to the compute-bound region, ultimately improving its overall performance.

2 Question 2

2.1 GPU Implementation with Loop Parallelism

To develop a GPU implementation of `number_crunching.cpp` that exploits loop parallelism, CUDA was chosen as the programming model. The compute functions `function_a()`, `function_b()`, `function_c()`, and `function_d()` were converted to CUDA kernel functions, allowing parallel execution on the GPU.

The memory management approach involved allocating device memory for input and output arrays using `cudaMalloc()`, transferring input data from host to device using `cudaMemcpy()` with the `cudaMemcpyHostToDevice` flag, and transferring output data back from device to host using `cudaMemcpy()` with the `cudaMemcpyDeviceToHost` flag. The device memory was freed using `cudaFree()` when no longer needed.

The grid and block dimensions for each kernel launch were determined based on the problem size `N` and the desired level of parallelism. The `dim3` structure was used to specify the grid and block dimensions, considering the size of the input arrays.

Table 6 shows the memory operations performed during the execution of the GPU implementation.

The majority of the data movement occurs in the host-to-device direction, with a total of 2.98 GiB transferred in 3 operations. The device-to-host data movement is minimal, with a total of 468.76 KiB transferred in 4 operations.

Table 7 presents the kernel operations and their execution times.

The `function_a_kernel` dominates the execution time, taking 99.9% of the total time with an average of 23.717 ms per instance. The other kernel functions have negligible execution times in comparison.

2.2 Task Graph and Task Parallelism

Figure 1 illustrates the task graph of the `number_crunching.cpp` source code at the function level. The graph shows the dependencies between the functions and the potential for task parallelism.

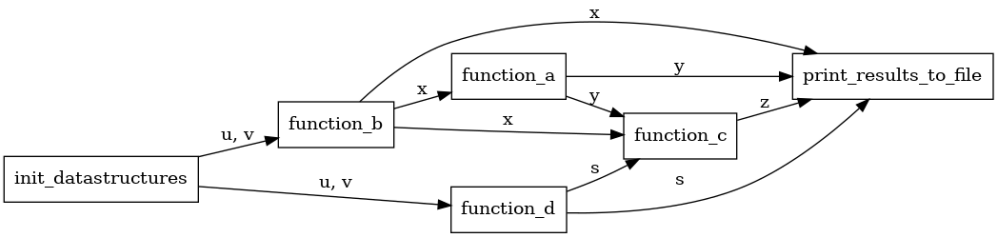

Figure 3: Task graph of number_crunching.cpp

To exploit task parallelism, the GPU implementation was extended to execute independent compute functions simultaneously. CUDA streams were utilized to enable concurrent execution of kernels on different streams. The `function_d_kernel` and `function_b_kernel` were launched on `stream1`, while `function_c_kernel` and `function_a_kernel` were launched on `stream2`. This allows for overlapping execution of independent kernels, potentially improving performance.

2.3 Approach and Optimisation Techniques

The process of adapting the code for GPU utilisation entailed several systematic steps. Initially, the source code was scrutinised to pinpoint opportunities for parallelisation and to understand data dependencies. This was followed by transforming the compute functions into CUDA kernel functions, adapting them to accommodate CUDA thread indices and block dimensions. Furthermore, device memory was allocated for input and output arrays, ensuring efficient data management between the host and the device. The optimal grid and block dimensions for each kernel launch were then determined, tailored to the specific problem size and the resources available on the GPU. To enhance efficiency, CUDA streams were employed to facilitate task parallelism and the concurrent execution of independent kernels.

A range of optimisation techniques was employed to refine the performance further. Device memory was initialised to guarantee proper setup prior to kernel execution. In the `function_`

`d_kernel`, shared memory was utilised to conduct reduction operations within each block, significantly minimising global memory accesses. The use of CUDA streams was instrumental in overlapping kernel execution with data transfers, thereby enabling effective task parallelism. Lastly, robust error management was implemented using the `ErrorCheck` function, which was crucial for detecting and addressing CUDA-related errors.

Table 4 presents the execution times of the GPU implementation for a problem size of $N = 10^5$ and compares it with the sequential CPU implementation from Table 1.

Table 4: Execution times for sequential CPU and GPU implementations

Implementation	**Execution Time (ms)**
Sequential CPU (`function_a`)	34.40
GPU with Loop Parallelism (`function_a_kernel`)	23.72

The GPU implementation achieves a speedup of approximately 1.45x compared to the sequential CPU implementation for `function_a`, demonstrating the effectiveness of GPU acceleration for this problem.

In conclusion, the GPU implementation of `number_crunching.cpp` successfully exploits loop parallelism and task parallelism using CUDA. The optimisation techniques applied, such as shared memory usage and CUDA streams, contribute to improved performance. The GPU implementation achieves a significant speedup compared to the sequential CPU implementation, showcasing the benefits of GPU acceleration for computationally intensive tasks.

Appendices

Table 5: Function execution time as a function of input size N

N	init_data (s)	function_a (s)	function_b (s)	function_c (s)	function_d (s)
10000	0.37	0.35	0.00	0.00	0.00
20000	1.40	1.39	0.00	0.00	0.00
30000	2.74	3.10	0.00	0.00	0.00
40000	6.59	5.57	0.00	0.00	0.00
50000	10.38	8.69	0.00	0.00	0.00
60000	14.55	12.57	0.00	0.00	0.00
70000	1.92	17.14	0.00	0.00	0.00
80000	8.81	22.02	0.00	0.00	0.00
90000	15.16	28.28	0.00	0.00	0.00
100000	4.13	34.40	0.01	0.00	0.00

Table 6: Memory Operations in Number Crunching Task

Total	Count	Avg	Med	Min	Max	StdDev	Operation
2.98 GiB	3	0.99 GiB	156.25 KiB	156.25 KiB	2.98 GiB	1.72 GiB	`[CUDA memcpy Host-to-Device]`
468.76 KiB	4	117.19 KiB	156.25 KiB	8 B	156.25 KiB	78.12 KiB	`[CUDA memcpy Device-to-Host]`

Table 7: Kernel Operations in Number Crunching Task

Time	**Total Time**	**Instances**	**Avg**	**Med**	**Min**	**Max**	**StdDev**	**GridXYZ**	**BlockXYZ**
99.9%	23.717 ms	1	23.717 ms	23.717 ms	23.717 ms	23.717 ms	0 ns	79x1x1	256x1x1
0.0%	10.785 µs	1	10.785 µs	10.785 µs	10.785 µs	10.785 µs	0 ns	1x1x1	256x1x1
0.0%	2.624 µs	1	2.624 µs	2.624 µs	2.624 µs	2.624 µs	0 ns	79x1x1	256x1x1
0.0%	2.496 µs	1	2.496 µs	2.496 µs	2.496 µs	2.496 µs	0 ns	79x1x1	256x1x1

Table 8: Cache Performance

Metric	**Value**
Runtime (RDTSC) [s]	0.3446
Runtime unhalted [s]	0.5733
Clock [MHz]	3320.4667
CPI	0.3562
Data cache requests	1702352000
Data cache request rate	0.5320
Data cache misses	9308
Data cache miss rate	2.908561×10^{-6}
Data cache miss ratio	5.467729×10^{-6}

Table 9: Data Performance

Metric	**Value**
Runtime (RDTSC) [s]	0.3440
Runtime unhalted [s]	0.5734
Clock [MHz]	3326.6837
CPI	0.3562
Load to store ratio	14.9975

Table 10: L3 Cache Performance

Metric	**Value**
Runtime (RDTSC) [s]	0.3435
Runtime unhalted [s]	0.5738
Clock [MHz]	3333.1933
CPI	0.3562
L3 access bandwidth [MBytes/s]	2385.3517
L3 access data volume [GBytes]	0.8194
L3 access rate [%]	0.4001
L3 miss rate [%]	0.3938
L3 miss ratio [%]	98.4345