Parallel Computing - Work Assignment Phase III

Eduardo Miguel Pacheco Silva
Departamento de Informática
Mestrado em Engenharia Informática
Guimarães, Portugal
pg53800

Inês Sousa da Silva
Departamento de Informática
Mestrado em Engenharia Informática
Guimarães, Portugal
pg55949

Luna dos Santos Figueiredo
Departamento de Informática
Mestrado em Engenharia Informática
Porto, Portugal
pg55979

***Abstract*— This report explores the optimization of a 3D fluid dynamics simulation based on Jos Stam's stable fluid algorithm. Three phases were implemented: instruction-level parallelism (ILP) for sequential optimization, OpenMP for shared-memory parallelism and CUDA for GPU acceleration. The goal is to improve execution performance, scalability and analyzing trade-offs between CPU and GPU based parallelism.**

Index terms—*Fluid Dynamics Simulation, ILP, OpenMP, CUDA, Profiling, Scalability*

I. Introduction

This report focuses on exploring different optimisation approaches on a 3D fluid dynamics simulation based on Jos Stam's stable fluids solver algorithm. The report is divided into three phases, each introducing **increasingly** advanced parallelization techniques to improve execution performance:

- **Phase 1:** focused on sequential code optimization, employing techniques such as instruction-level parallelism (ILP), loop reordering, and cache optimizations. These methods aimed to maximize computational efficiency by minimizing memory latency (cache efficiency) and improving processor utilization.
- **Phase 2:** Introduced shared-memory parallelism using OpenMP, introducing directives for efficient multithreading.
- **Phase 3:** transitioned to a CUDA-based implementation, harnessing the massive parallelism of GPUs. By offloading computationally intensive tasks to the GPU, this phase aimed to handle larger workloads and achieve significant performance gains through optimized memory access and thread management.

By analyzing performance across these implementations, the study evaluates the trade-offs of each approach. Scalability testing and performance metrics are used to validate the results and guide optimization strategies.

II. Iterative Optimization Process

All optimizations described in all three phases were implemented incrementally and tested **iteratively**. By measuring the impact of each change individually, we were able to refine the solution systematically, ensuring that no optimization inadvertently degraded performance or accuracy.

III. Test Environment

The performance tests were conducted on two machines, Machine 1 and Machine 2, with distinct hardware configurations (see appendix A) in a controlled environment to ensure consistency and reproducibility. The evaluation process was divided into two stages:

A. Intermediate Performance Evaluation

Each phase was tested on a single machine using a specific grid size to assess the impact of optimizations incrementally:

- **Phase 1 (ILP):** Small grid size (42^3) on **Machine 1**
- **Phase 2 (OpenMP):** Medium grid size (84^3) on **Machine 1**
- **Phase 3 (Cuda):** Large grid size (168^3) on **Machine 2**

B. General Performance Evaluation

Additionally, to ensure comprehensive testing and evaluate the solver's scalability and performance across different workloads, we conducted tests on grids of varying sizes: 42^3, 84^3,126^3,168^3, 210^3.

IV. Phase 1 (ILP)

This initial phase focuses on optimizing the single-threaded implementation of the solver by leveraging **Instruction-Level Parallelism** (ILP). The main goal was to analyze the initial code, identify performance bottlenecks, and apply sequential optimization techniques to reduce execution time by maximizing performance by improving computational efficiency while minimizing memory latency.

A. Code Analysis

Profiling tools such as **gprof** and **perf** were used to analyze the code's execution behaviour and identify computational hotspots.

TABLE1: *gprof* results

func name	% time	self seconds	calls
lin_solve()	85.59	20.15	600
advect()	7.48	1.76	400

Considering the results present in the table above (Table 1) we realize that `lin_solve()` function was our computational burden accounting for **85.59%** of the runtime, making it the primary optimization target.

Further analysis of the `lin_solve()` function and the rest of the source code revealed that the complexity growth rate, could not be reduced due to the inherent nature of the numerical solver. However, multiple functions exhibited a high degree of arithmetic intensity ($O(N^3)$ and $O(N^2)$) but poor memory access patterns showed poor spatial locality, increasing cache misses and contributing to its high execution time.

The macro `IX(i, j, k)` calculates the linear index for a 3D array stored in row-major order. In this layout, the i dimension represents the fastest-changing index (contiguous memory), followed by the j dimension, and finally the k dimension.

B. Optimizations and their impact

Based on the previous code analysis, we successfully implemented the following optimizations:

1) *Loop Order Rearrangement:*

Loops in `lin_solve()`, `set_bnd()`, `advect()`, `project()` were rearranged **from** i, j, k to k, j, i. This optimization seeks to improve cache efficiency decreasing cache misses by ensuring that memory is accessed in a sequential pattern.

2) *`lin_solve()` Block/Tiling:*

The grid was segmented into smaller blocks to enhance cache efficiency. Each block was processed independently, ensuring that neighboring data elements remained in the cache for subsequent computations. By employing the Block technique, we implicitly benefited from a reduction in the number of accesses to the lower levels of memory hierarchy.

3) *Division Replacement with Multiplication:*

To reduce computational overhead, divisions were replaced with equivalent multiplications wherever possible. Division operations are generally more expensive in terms of clock cycles compared to multiplication. For instance, division like:

$$a = \frac{x}{c} \tag{1}$$

were replaced with:

$$a = x \times \left(\frac{1}{c}\right) \tag{2}$$

This optimization ensures that the division value ($\frac{1}{c}$) is precomputed **before entering the loop**, so the loop operates solely with multiplications. By moving the reciprocal calculation outside the loop, we minimized the arithmetic workload within the iterative computation, further reducing execution time.

It is important to note that the compiler flag `-Ofast` also performs such optimizations by replacing divisions with multiplications as part of its aggressive floating-point transformations (see Appendix B). However, implementing this optimization manually ensured its effectiveness regardless of compiler settings, providing explicit control over computational efficiency and enabling benefits even in environments where `-Ofast` was not used.

C. Intermediate Performance Evaluation

All results presented in this section were obtained using a small grid size of $\mathbf{42^3}$ and tested on **Machine 1**.

Considering the adopted optimizations we managed to reduce significantly the number of instructions and cycles, leading to less memory accesses. As shown in Figure 1, the number of L1-Cache misses also decreased to approximately 2.49% cache misses out of all cache hits. In addition, the CPI dropped slightly from 0.531 to 0.509. The execution time was reduced from 30.85 seconds to **3.69 seconds**, demonstrating the effectiveness of the applied optimizations.

Finally we achieved an overall **speedup of 8.36** times faster compared with the original program and an output deviation of +0.3 due to `-Ofast` optimizations.

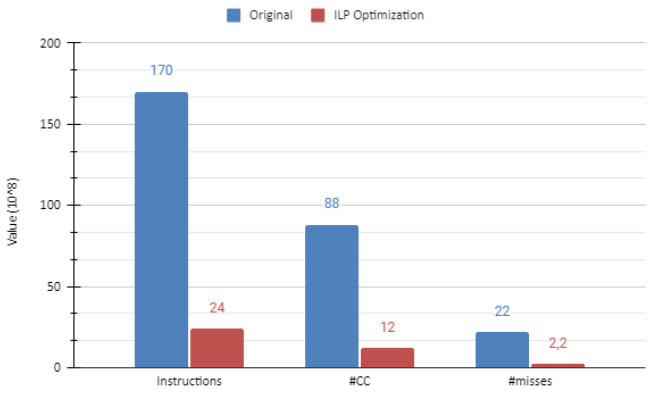

Fig. 1: Comparison of Performance Metrics Between Original and ILP-Optimized Code (Values Scaled by 10^8)

V. Phase 2 (OpenMP)

In this phase, the focus shifted to leveraging shared memory parallelism using **OpenMP** to optimize the 3D fluid solver. To improve performance and adaptability to parallel execution, the original solver was **replaced with a Red/Black Gauss-Seidel solver**. This change enabled more efficient computation by allowing structured parallelization of alternate grid points.

A. Parallelism Potential Analysis

Before implementing parallelism, we evaluated the theoretical ideal speedup achievable using **Amdahl's Law**. By analyzing the code and using tools like *gprof* and *perf*. We determined that 99% of the workload was parallelizable (see Appendix C), leaving only 1% as the serial fraction.

B. Implementation of OpenMP Parallelism

To exploit the computational capabilities of multicore CPUs, OpenMP directives were used to parallelize key computational hotspots in the fluid solver. Loops within functions such as `lin_solve()`, `project()`, `advect()`, `add_source()`, `clear_data()`, `sum_density()` were optimized using **parallel for**, with the appropriate scheduling strategy **static** to ensure balanced workload distribution. Nested loops were parallelized using the **collapse** clause, enabling multi-level parallelism for 3D grid computations. **Reduction** clauses were employed to handle shared variables, such as `max_c` in `lin_solve()`. These optimizations ensure efficient use of available cores, minimize thread contention, and improve scalability while maintaining numerical accuracy.

C. Intermediate Performance Evaluation

All results presented in this section were obtained using a medium grid size of 84^3 and tested on **Machine 1** which was equipped with 40 threads.

The analysis of the performance metrics reveals several key trends in the application's behavior across different thread counts. The application's **required bandwidth** per thread remains consistently **low** (below 1 GB/s) and is significantly less than the available bandwidth per thread across all thread counts (appendix D). This indicates that the application is **not hitting the memory wall**. Instead, performance is influenced by other factors, such as computation efficiency, thread synchronization, and resource contention.

At lower thread counts, the application is compute-bound, with high arithmetic intensity (AI) and efficient CPU utilization, as reflected by the low CPI. However, as thread counts increase, AI drops sharply due to higher LLC misses, and CPI rises, suggesting increased synchronization overhead and reduced computational efficiency (appendix E and F).

The execution time improves significantly up to 24 threads, demonstrating that the load is well-balanced across the threads (appendix G). However, after 24 threads, gains plateau, and performance degrades slightly. This stagnation, also reflected in the speedup curve (Fig. 2), indicates that synchronization overhead, cache contention, and potential non-uniform memory access effects begin to dominate, limiting scalability.

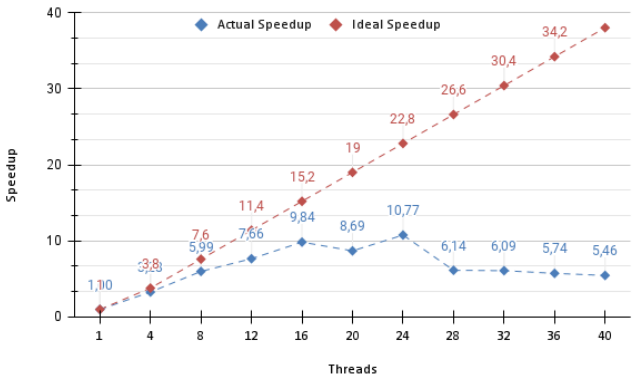

Fig. 2: Strong Scalability analysis: Actual vs. Ideal Speedup with increasing Threads

The application performs optimally with 16-24 threads, where execution time reaches a peak of **1.38s** a total **speedup of 10.77** and arithmetic intensity is well balanced. Scalability is limited beyond 24 threads due to synchronization overhead and resource contention.

VI. Phase 3 (CUDA)

Lastly, our objective in this phase was to leverage GPU acceleration through **CUDA (Compute Unified Device Architecture)** to achieve significant performance improvements. By replacing the CPU-based OpenMP implementation with highly parallelized GPU kernels, we aimed to fully exploit the computational of GPUs.

A. Profiling

Profiling was conducted using NVIDIA Nsight Systems (NSys) to analyze kernel execution times, memory bandwidth, and GPU resource utilization. The tool provided detailed insights into thread efficiency, warp execution, and memory access patterns, which guided optimizations during this phase.

B. Optimizations and their impact

1) *GPU Memory Allocation:* At the beginning of the simulation, all fluid simulation arrays (e.g. `u`, `v`, `w`, `dens`...) were allocated on the GPU using `cudaMalloc`. This avoided

repeated memory allocations and costly host-device memory transfers during runtime. By keeping all data in device memory throughout the simulation we:

- Minimized latency associated with `cudaMemcpy` calls for frequent data exchanges between host and device;
- Reduced memory management overhead by eliminating repeated allocation/deallocation;

2) *`lin_solve()`:* The previously implementation of Red/Black Gauss-Seidel method was updated where each colour updates (red and black) were performed in two separate kernel launches.

Moreover, we realized a convergency check would still be needed to achieve better performance. Thus, we implemented a **complete static unroll**. The reduction loop was fully unrolled to eliminate runtime branching and improve register-level performance. By statically unrolling the loop, each thread performed a fixed sequence of operations, leveraging GPU pipelines efficiently.

Static unrolling maximized instruction throughput and minimized thread divergence, which is critical for performance.

A key optimization in the `lin_solve()` kernel was the use of **`atomicMaxFloat`** for reductions. Each block performed a partial reduction in shared memory to calculate the maximum change within its subset of the grid. Consequently, the results were then reduced globally using `atomicMaxFloat`, which ensures thread-safe updates to the global maximum. By performing the reduction entirely on the GPU, the overhead of transferring the entire grid to the CPU for a final reduction was avoided leveraging GPU's parallel processing capabilities.

3) *Other Kernels Implementations:* All other critical functions (`add_source()`, `advect()`, `project()`) of the solver were implemented as dedicated CUDA kernels to maximize modularity and efficiency.

The `project()` step was split into two separate kernels.

A dedicated `set_bnd_kernel()` was implemented to handle boundary conditions efficiently in the x, y and z directions. Besides, special care was taken for corner cases, which were managed by a single thread.

C. Block Size Selection

The choice of block sizes was carefully tailored to balance, occupancy, memory efficiency, and computational workloads across kernels.

For all kernels we have set the blocks as it follows:

$$B_x = 32 \qquad B_y = 4 \qquad B_z = 1$$

Threads along the x-dimension (i) access contiguous memory addresses, ensuring **coalesced memory access**. This optimizes DRAM bandwidth utilization, which is critical for memory-bound operations.

D. Challenges

Despite efforts to optimize kernel design, it was not feasible to join the two kernels (red and black updates) in `lin_solve()`. The Red/Black Gauss-Seidel method inherently requires sequential dependency between red and black point updates. Joining the kernels would violate this dependency, introducing race conditions and numerical inaccuracies. Therefore a single kernel would just synchronize the warps that belong to the **same** block and not grid.

Moreover, despite the GPU-based reduction, there remained an overhead from transferring the final maximum value back to the CPU at the end of each iteration to check for early convergency.

E. Intermediate Performance Evaluation

All results presented in this section were obtained using a large grid size of $\mathbf{168^3}$ and tested on **Machine 2** achieving an execution time of **12.6 seconds**. The `lin_solve_kernel()` accounted for **89.6%** of kernel execution time, making it the primary computational bottleneck. The implementation reached a **maximum throughput of 90.16GiB/s**, which is close to the **theoretical memory bandwidth of a 112 GiB/s** for the NVIDIA GTX 1050 Ti (see Appendix A), demonstrating efficient memory utilization.

Memory operations including Device-to-Host (DtoH), transfers, Host-to-Device (HtoD) transfers, and memory initialization (`cudaMemset`), accounted for only **0.1%** of the total execution time, rendering their impact negligible.

We conducted an extensive analysis of how thread **synchronization impacts the density results**, aiming to reduce execution time. We found that removing all synchronization between processing units (PUs) **did not affect the density values** but resulted in a significant trade-off in execution time.

VII. General Performance Evaluation

This final evaluation integrates data form all phases (ILP, OpenMP, CUDA) and grid sizes to provide a comprehensive analysis of performance, scalability and efficiency across machines and implementations.

Firstly, Machine 1 demonstrated better performance with OpenMP compared to CUDA due to its high thread count capability. Equipped with **40 threads**, it leveraged the shared-memory parallelism of OpenMP more effectively, distributing the workload across a large number of threads. This resulted in lower execution times compared to CUDA for most grid sizes, as shown in Fig. 3.

OpenMP benefited from the CPU's cache hierarchy. This minimized memory latency and improved overall efficiency.

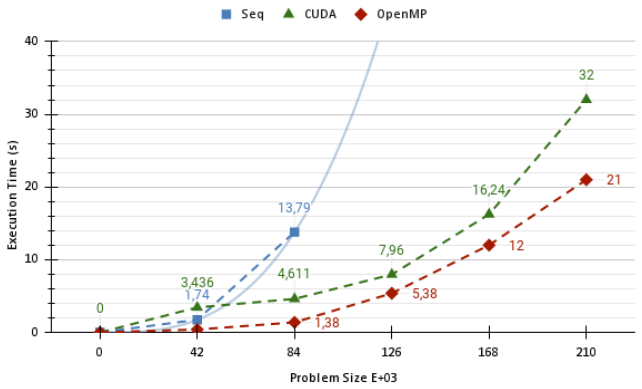

Fig. 3: Execution Time vs. Problem Size on Machine 1 (Cluster)

Machine 2, on the other hand, demonstrated superior performance for CUDA compared to OpenMP (Fig. 4). With only **8 cores** available for OpenMP, the CPU's parallel capabilities were limited, making CUDA the more efficient choice, particularly as the workload increased. As the problem size increased (e.g. 168^3 and 210^3), the CUDA implementation scaled more effectively. The GPU's massive thread parallelism and optimized memory bandwidth (achieving previously mentioned **90.16GiB/s**) to handle larger workloads efficiently.

With only 8 threads, OpenMP faced significant contention and synchronization overhead, limiting its scalability and efficiency.

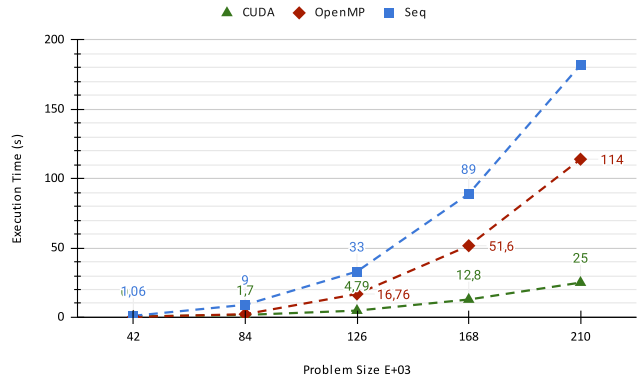

Fig. 4: Execution Time vs. Problem Size on Machine 2

Unsurprisingly, the sequential implementation performed the worst across all grid sizes and machines. Since we couldn't collect data for sizes larger than 84^3 on Machine 1, we used a power-based tendency line to estimate its performance, providing a reasonable prediction of execution time.

A. *Weak Scalability Analysis*

The **weak scalability** analysis (Fig. 5) shows that CUDA's performance does not align well with **Gustafson's Law** (see Appendix I). While Gustafson's Law predicts nearly linear speedup as the number of processing units increases, the observed scalability was suboptimal due to bottlenecks and inefficiencies, such as `lin_solve()` reduction, mentioned before (Section VI).

This issue was not caused by a memory wall, as both machines still had sufficient bandwidth (measured in GiB/s) to support higher throughput (see appendix A). Instead, the primary factors were:

- challenges mentioned in Section VI.D
- the overhead of kernel launches, particularly in `lin_solve()` which accounted for 14,232 instances
- thread synchronization delays

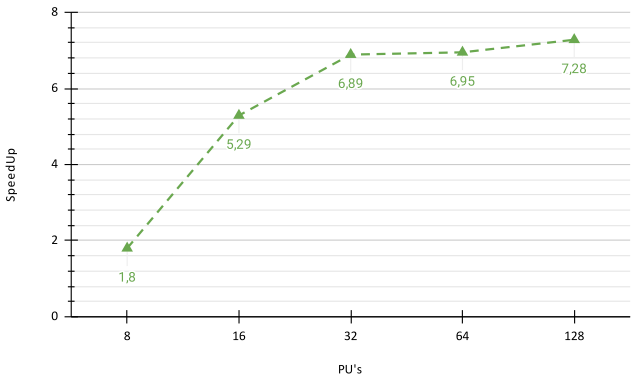

Fig. 5: Weak Scalability Analysis: CUDA Processing Units (PUs) vs. Speedup with Increasing Workload on Machine 2.

Furthermore, Smaller blocks achieved better GPU occupancy, which allowed the GPU to better utilize its Streaming Multiprocessors (SMs). Thus, with more blocks running concurrently, the GPU avoids idle SMs, achieving higher parallelism. Moreover during the reduction kernel, threads progressively become inactive as the reduction progresses. Therefore smaller blocks minimize the number of threads per block that go idle, maintaining better warp efficiency compared to larger blocks.

VIII. Conclusion

In this project, we systematically explored and implemented optimizations for a 3D fluid dynamics solver across three distinct phases: Instruction-Level Parallelism (ILP), OpenMP and CUDA. Each phase progressively introduced advanced optimization techniques, enabling us to achieve significant execution time reductions and performance improvements while maintaining numerical accuracy.

OpenMP demonstrated storng performance on high-threaded-count CPUs. However, its scalability was constrained beyond 24 threads due to synchronization, cache contention. Conversely, CUDA excelled on Machine 2, leveraging its GPU.

Weak scalability analysis revealed deviation from Gustafson's Law, primarily due to `lin_solve()` kernels bottlenecks.

This study demonstrates the importance of tailoring optimization strategies to specific hardware architectures.

References

[1] J. Hammond and J. D. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers."

[2] University of Virginia, Department of Computer Science, "STREAM Benchmark: Measuring Sustainable Memory Bandwidth."

[3] "Gustafson's law."

Appendix A

Test Environment Hardware Specifications

Performance tests were conducted on two machines with distinct hardware configurations described in the tables below (Tables 2 and 3).

TABLE2: CPU Specifications

Machine	CPU	Threads	L1 Cache
1 (Cluster)	Intel Xeon E5-2670 v2	40	32 KB
2	Intel Core i7-9700K	8	32 KB

TABLE3: GPU Specifications

Machine	GPU	CUDA Cores	Memory	Memory Bandwidth
1 (Cluster)	NVIDIA Tesla K20m	2496	5 GB	208 GB/s
2	NVIDIA GTX 1050 Ti	768	4 GB	112 GB/s

Appendix B

-Ofast Optimizations

We analyzed the effect of each flag separately with the help of assembly, as follows:

- **Ofast** : when compiling with -Ofast flag, **–fast-math** flag is typically enabled, allowing for a faster but less precise implementation of floating-point arithmetic operations. The loss of precision can result in approximate results that do not comply strictly with IEEE standards.
- Additionally, division operations were replaced with multiplications owing to the fact that division operations take more clock cycles to perform (described below).

-Ofast Disabled

```
...
addq    %rdx, %rax
divss   -88(%rbp), %xmm0
movss   %xmm0, (%rax)
...
```

-Ofast Enabled

```
...
addq    $1, %rdx
mulss   %xmm5, %xmm0
movss   %xmm0, -4(%rax)
...
```

Appendix C

Estimation of Ideal Speedup Using Amdahl's Law

Amdahl's Law:

$$S(P) = \frac{1}{f + \frac{1-f}{P}} \tag{3}$$

- $S(P)$: Speedup.
- f: serial fraction of work.
- P: number of PUs.

TABLE4: *gprof* flat profile of the ILP optimized sequential code

% time	self seconds	calls	function name
51.26	16.35	300	*lin_solve()*
32.14	10,25	100	*vel_step()*
8.31	2.65	200	*project()*
6.12	1.95	100	*dens_step()*
1.32	0.42	6794	*set_bnd()*

The functions listed on the table above represent the major sort list of time spent in each function and the percentage of the total running time consumed by each one. Each of those functions can be parallelized based on the type of operations and how the functions are implemented.

Other functions like `advect()` and `add_source()` can also be parallelized. Thus P can be calculated as:

$$\sum_{\%_{\text{parallel}}} = 51.26 + 32.14 + 8.31 + 6.12 + 1.32 + \%_{\text{advect}} + \%_{\text{add\_source}} \approx 99\%$$

$$f = 1 - 0.99 = 0.01 \tag{4}$$

$$S(N) = \frac{1}{0.01 + \frac{0.99}{N}} \tag{5}$$

Appendix D

Required Bandwith vs Available Bandwith

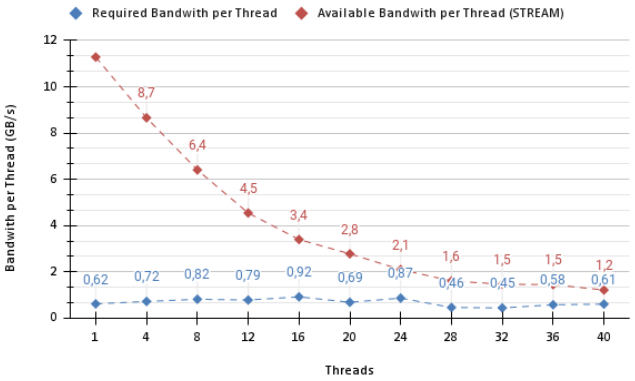

Fig. 6: Required Bandwidth per Thread vs Available Bandwidth (STREAM) [1]

The chart above compares the required bandwidth per thread (blue line) from our application's workload with the available bandwidth per thread (red line) from the STREAM benchmark [2] as the number of threads increases.

Required bandwidth was measured according to the total memory traffic generated by the application and dividing it by the execution time. For **required bandwidth per thread**, the total required bandwidth was further divided by the number of threads used. This approach provides an estimate of how much memory bandwidth each thread demands during execution.

$$\text{MemTraffic(GB/s)} = \frac{(\text{LLC-loads} + \text{LLC-stores}) \times 64}{10^9} \quad (6)$$

$$\text{ReqBandwidth(GB/s)} = \frac{\text{MemTraffic}}{T_{\text{exec}}} \quad (7)$$

$$\text{ReqBandwidth per Thread(GB/s)} = \frac{\text{ReqBandwidth(GB/s)}}{P} \quad (8)$$

Appendix E

Arithmetic Intensity

We estimated the Arithmetic Intensity based on the number of #Instructions ($\#I$) and Last-Level Cache Misses (LLC.MISS).

$$\text{LLC.MISS} = \text{LLC-load-misses} + \text{LLC-stores-misses} \quad (9)$$

$$\text{Arithmetic Intensity} = \frac{\#\text{I}}{\text{LLC.MISS}} \quad (10)$$

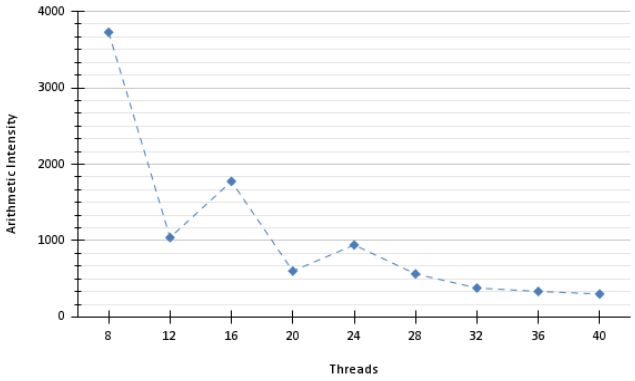

Fig. 7: Arithmetic Intensity

Appendix F

CPI Evolution

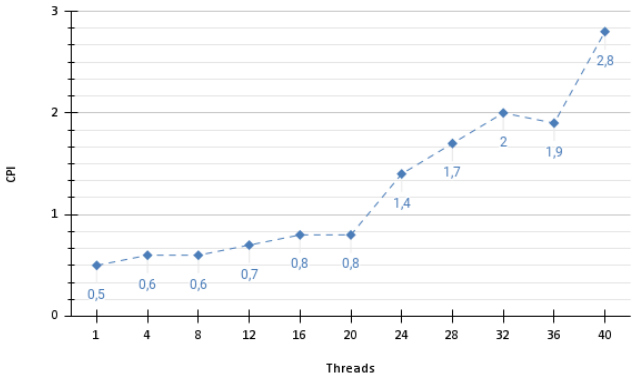

Fig. 8: CPI Evolution

Appendix G

Efficiency Analysis

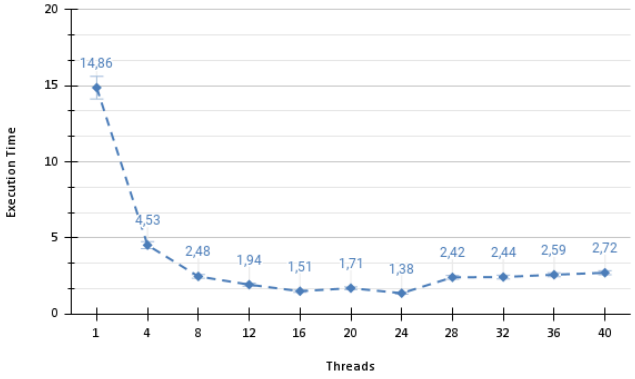

Fig. 9: Efficiency Analysis: Execution Time(s) vs. Number of Threads

APPENDIX I

GUSTAFSON'S LAW

In opposition to Amdahl's Law, which assumes a fixed workload, Gustafson's Law [3] suggests that as the number of processors increases, we can proportionally increase the size of the aprallelizable workload.

$$S = P + (1 - P) \cdot f \tag{11}$$

Where:

- S: Speedup of the parallel system
- P: Number of PUs
- f: serial fraction of work

As previously mention in Appendix C, serial fraction is $f = 0.01$ (meaning 99% of the workload is parallelization).