Performance evaluation of GPU-enhanced Jacobi algorithm implementations for optimizing CFD Poisson partial differential equation solving

Sebastian Sbîrnă
DTU Compute Department
Institute of Applied Mathematics and Computer Science
Technical University of Denmark
Kongens Lyngby, Denmark
seby.sbirna@gmail.com

Liana-Simona Sbîrnă
Department of Chemistry
Faculty of Sciences,
University of Craiova
Craiova, Romania
simona.sbirna@gmail.com

2024 28th International Conference on System Theory, Control and Computing (ICSTCC) | 979-8-3503-6429-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICSTCC62912.2024.10744755

***Abstract* — The current work presents an in-depth analysis of several optimizations using GPU parallel computing applied to the Jacobi method for solving Poisson partial differential equations in computational fluid dynamics (CFD). We expand on previous CPU-parallelized Jacobi algorithm research, exploring four GPU-optimized Jacobi method variants: single-threaded, multi-threaded, multi-GPU and a norm-based stopping criterion kernel. These implementations are benchmarked against a multi-threaded CPU baseline. Results indicate that, whereas the single-threaded GPU version is slower than the CPU baseline, multi-threaded GPU versions achieve significant speed gains, especially for larger grid sizes. The multi-GPU version doubles memory bandwidth, enhancing performance for extensive computations, despite overhead for smaller matrices. The norm-stopping criterion kernel offers early convergence for small matrices but at a high overhead cost. Profiling confirms a memory-bound bottleneck, suggesting single-precision and optimized memory access as improvements. Ultimately, multi-threaded GPU kernels substantially outperform the CPU baseline for large-scale CFD problems, establishing GPUs as efficient accelerators for the Jacobi algorithm.**

Keywords — GPU computational optimization; parallel processing; software engineering; CFD Poisson process

I. INTRODUCTION

Within the field of computational fluid dynamics (CFD), our previous research [1] looked into the possibility of finding computationally-efficient algorithmic convergence models for solving Poisson partial differential equation. The results from the previous paper have shown that a CPU-parallelized Jacobi algorithm has a good performance-per-core and scalability potential, compared with the Gauss-Seidel method.

The focus of this research is to find, model, compute and assess the possibility of optimizing the solving of CFD Poisson differential equations using GPU hardware for parallel high-performance computing.

Because of its ability for high throughput to complete tasks in a time unit, the GPU is optimally used in highly parallel, compute-intensive operations. We are interested in optimizing this problem from a throughput-oriented perspective, since, for large matrix sizes, differential equation solving through Jacobi method will become memory-bound.

In order to remind the reader of the general idea behind the Poisson problem, it is mentioned that its equation represents an elliptic partial differential equation, which is widely used in science and industry [2 - 5]. A three-dimensional Poisson equation (1) has been applied to estimate the heat distribution in a cubic room only considering the influence of a radiator. The three-dimensional problem is referring to estimating the steady-state temperature across the room, only considering the influence of a radiator:

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2}=-f(x,y,z),\quad (x,y,z)\in\Omega \qquad (1)$$

Due to the fact that it is often impossible to analytically determine the unknown function u of (1) [2 - 5], the problem is first discretized on a cubic $(N+2)\times(N+2)\times(N+2)$ grid, where N represents the number of interior points in each dimension, and then solved using a numerical method.

The only method being discussed in this next part of the assignment is the Jacobi method, which is capable of numerically approximating the solution to the differential equation [6].

The assumptions for the modeling of a CFD Poisson equation are the same as for our previous research [1], with: $\Omega=[-1,1]\times[-1,1]\times[-1,1]$, six interior walls (as per a cubic room), with five of them having a temperature of 20°C, while the sixth having the temperature of 0°C. The radiator, which outputs 200°C/m^2 radiation, is set close to the 0°C wall. The heat dissipation will again be measured through equidistant points within the 3D room and computing their values through differential equations [2 - 5].

II. HARDWARE AND COMPUTATIONAL TOOLS

For reproducible and reliable performance comparison results, all algorithm implementations mentioned in this paper have been executed in a computing cluster environment [6]. This cluster gives access to peer-linked powerful NVIDIA GPUs, which are essential for testing the CUDA performance of our kernels. In this cluster, all machines rely on the same hardware, which is described in Table I:

TABLE I. HARDWARE AND SOFTWARE SPECIFICATIONS

CPU-related properties	CPU Model	Intel XEON® Gold 6126 @2.60GHz
	Architecture	64-bit
	Sockets	2
	CPUs per socket	12
	Total number of CPUs	24
GPU-related properties	GPU Model	NVIDIA Tesla V100-PCIE-16GB @1.38GHz
	Total Memory	16 GB
	CUDA Cores	5120
	Multiprocessors	80
	CUDA Cores /multiprocessor	64
	Warp size	32
	Max threads per block	1024
	Max threads per multiprocessor	2048

From these specifications, it is relevant to know the number of lock-step threads running on the same block, which is (the standard) 32. Also, by taking into account other properties, such as the number of maximum threads per block, it will allow better optimization of the written code when storing matrices and computing within GPU memory.

The program environment in which the library of CUDA kernels and C functions is compiled is as follows:

- Compiler: $nvcc$ for CUDA-related files and gcc v8.3.0 for CPU-related files
- Optimization flags: all code was compiled using a general level 3 optimization: "-O3" parameter for gcc
- Matrix implementation: Single-pointer vectorized array allocation

III. Implementation of Optimized Jacobi Algorithm

As per our previous research [1], for the Jacobi algorithm introduced, it is of interest to determine the number of grid points inside the cubic room, which is represented in our case as the maximum size of the variable N.

For this research, we will start from the CPU-parallelized version of the Jacobi algorithm, and re-implement it for solving differential equations using GPU-optimized kernels. There will be four different versions of such functions, which will have the following naming conventions:

- jac_gpu1: Single-threaded implementation with 1 single thread running.
- jac_gpu2: Multi-threaded implementation with 1 thread per internal grid point.
- jac_gpu3: Multi-threaded multi-GPU implementation where the problem is solved through execution of 2 parallel kernels, each on a different GPU.
- jac_gpu4: Multi-threaded implementation, with 1 thread per internal grid point, and norm-based stopping criterion. The measuring tool for convergence will be the Frobenius norm $\|\cdot\|_F$ [7];

These algorithms will all be compared against an OpenMP-based multithreaded CPU "*baseline*" of the Jacobi method. When performing comparison and analysis between different kernel versions in the following subsections, these experiments have used different grid sizes, with the chosen values of N depending on whether the routines executed were the sequential ones (much slower, therefore N must be kept low), or the parallelized ones, and the number of maximum iterations before stopping the kernel has been set to $max\_iter = 1000$.

Operations relating to memory read/write for transferring initial values of the u, u_old and f to the GPU and copying the final results back to the host memory will be included in the total computing time, so as for our comparison to be fair between CPU and GPU versions. This choice has been made because the memory transfer is a necessary drawback of using the GPU that can be avoided by staying on the CPU. The fairest comparison is thus, when this additional overhead is taken into account.

In order to properly describe the optimization extent of the various kernels, we have chosen to visualize the data for these comparisons with graphs (shown in each subsection below, respectively). These graphs will plot the memory footprint necessary for working with our three cubic vectors against the number of gigaflops in computation.

The formula for computing the memory footprint necessary for 3 cubic matrices of size $(N + 2)$ and the size of a double being 8 bits is (2):

$$memory = 3 \cdot (N + 2)^3 \cdot 8 \tag{2}$$

The size of the cubic grid points N selected for experiments will depend highly on the size of the requested GPU memory and on the method on which the kernel is computing. The default amount of GPU memory requested in our batch scripts is 5GB. A cubic grid with $N \cong 600$ is close to requiring all the requested 5GB available.

Since some of the kernels perform quickly high-performance computations even upon such large cubic matrix sizes, and since the maximum GPU memory available on our Tesla V100 models is 16GB, it is reasonable to increase the request of our GPU memory to 12GB, in order to see the performance of kernel execution when $N = 768$ (which requires about 10.7GB).

We have selected the size for experiments of the cubic grid in the following way: for the sequential version jac_gpu1, it has been decided that any cubic matrix size above $N = 128$ takes too long to compute the output, therefore that will be the upper bound. Therefore, for the 1-core baseline CPU and for jac_gpu1, the sizes of N have been selected as: [16, 24, 32, 48, 64, 96, 128], in total having 7 grid sizes to compare against.

For the 12-cores CPU baseline and the parallelized GPU kernels, it can be anticipated that their performance will be much faster than the sequential code, so it is also reasonable to select higher grid matrix sizes to compute against, however not larger than our total requested GPU memory. This allows our setup to include the following values of N: [16, 24, 32, 48, 64, 96, 128, 192, 256, 384, 512, 600, 768], in total having 13 grid sizes for our plots.

In order to compute the speed-up of the kernels, since most of the time the GPU kernel is much faster than the CPU baseline, a speedup above 1.00 means that the GPU kernel has outperformed the CPU baseline in *flop/s* operations. The speedup formula will be as per eq. (3):

$$speedup = \frac{CPU[s]}{GPU[s]} \tag{3}$$

For the calculation of the number of *flop/s*, eq. (4) is used, where N is the number of inner grid points (in each dimension), I is the number of iterations performed, and T is the total wall-clock time spend performing the algorithm. The first part of the equation represents the number of lattice updates per second, which then needs to be multiplied with F, the number of floating point operations in the innermost loop of an algorithm.

$$flop/s = \frac{N^3 \cdot I \cdot F}{T} \tag{4}$$

In this next section, the best version of the parallelized OpenMP Jacobi algorithm has been chosen for reference as "*baseline*", and modified so that the stopping criterion is the maximum number of iterations allowing the program to run.

Against this baseline, four different GPU kernels have been created, which aim to consistently optimize the efficient computing of the updated values of matrix u. Only for the sequential GPU kernel (i.e. jac_gpu1), the CPU version of Jacobi method will also be compared against the usage of one core (instead of the optimal number of 12 cores), in order to make a fair 1:1 comparison between them. All of the other kernels will have as baseline the 12-core parallelized OpenMP Jacobi CPU implementation.

IV. Performance Analysis of GPU-Optimized Kernels

A. *Sequential kernel (jac_gpu1)*

The sequential kernel implementation of the Jacobi method is the most inefficient of all four, since the individual cores on the GPU are much slower than their CPU counterparts. The primary motivation for using this version is to verify that memory management is done correctly.

When timing this kernel against the CPU baseline, it is important to select the number of cores running on the CPU to be 1, in order to compare in a fair manner with the sequential version of the GPU implementation.

The time comparison between this version and the CPU version can be seen in Table II, along with a plot comparing the memory footprint of the cubic matrix sizes against the performance of the algorithms (in GFlops), which can be seen in Figure 1. It must be mentioned that both axes of the plot are in logarithmic scale, so as to better see the order of magnitude in the performance differences between kernel and baseline.

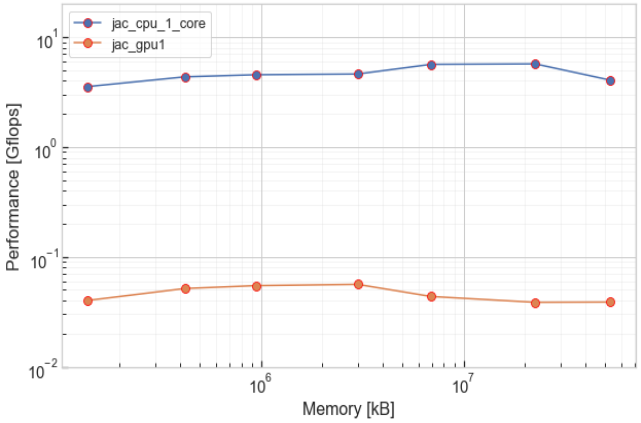

Fig. 1. GFlops performance comparison of the jac_gpu1 kernel against the 1-core CPU baseline

TABLE II. Performance comparison between jac_gpu1 and the 1-core CPU baseline

Algorithm	Time (s) for $N = 64$	Speed-up (compared with baseline)
CPU Baseline (1-core)	0.370	1.0000x (baseline)
jac_gpu1	48.104	0.0076x

From Figure 1, it can immediately be seen that the sequential GPU version of the Jacobi algorithm is two orders of magnitude slower than the sequential CPU version. This is a very reasonable result, since the GPU cores has much slower clock frequency than the CPU cores, as per Table I. Additionally, the CPU is potentially able to use instruction-level parallelism by transforming iterations of the loop into vector instructions. Running one thread per block on a GPU is highly inefficient, and the above table and graph demonstrate that.

It is worth noting that this kernel version is defeated by even the slowest possible run of our CPU, when no OpenMP parallelization is happening.

However, since all of the parallelized kernel versions will be much faster than even the quickest CPU version (parallelized on 12 threads), this 1-core baseline will not be used further in the assessment of the other kernels' performances.

Just for comparison reasons, let us have a data graph plotting both 1-core, 12-core baselines and also this sequential kernel version. By requesting more CPU cores from the cluster computing systems and setting the flag "*OMP_NUM_THREADS=12*" in our batch scripts, we are able to obtain a new, "best-version" baseline for the CPU, which can be seen in Figure 2 and Table 3 below.

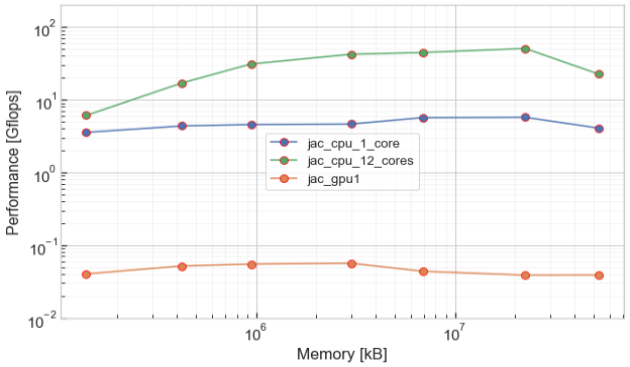

Fig. 2. GFlops performance comparison of the jac_gpu1 kernel against the 1-core and 12-core CPU baseline

TABLE III. Performance comparison between jac_gpu1 and the 1-core CPU baseline

Algorithm	Time (s) for $N = 64$	Speed-up (compared with baseline)
CPU Baseline (12-cores)	0.047	1.0000x (baseline)
jac_gpu1	48.104	0.0009x

B. *Naïve multi-threaded kernel*

In order to efficiently utilize the high computational throughput of the GPU, it is necessary to write a kernel which fits the SIMT model. This next version of our code will use one GPU thread per grid point update, thereby updating the whole cubic grid in parallel.

TABLE IV. Speed-up comparison between jac_gpu2 and the 12-core CPU baseline

Algorithm	Time (s) for $N = 768$	Speed-up (compared with baseline)
CPU Baseline (12-cores)	242.402	1.0000x (baseline)
jac_gpu2	23.914	10.136x

When timing this kernel against the CPU baseline, it is important to note that the size of the cubic grid matrix used as reference has been changed from $N = 64$ (used in the previous comparisons) to $N = 768$, since this is the largest value of N for which data has been collected on both the baseline and the faster GPU kernels.

When analyzing the speed-up improvements collected in Table IV, it can be noticed that running a parallelized version of the CUDA kernel allows for a speed-up of more than 10x compared to a parallelized CPU version.

Against this, the jac_gpu1 version is extremely slow, and it has been shown there for comparison (note the different speeds compared to Table II, since the size of N has increased).

Lastly, a plot showing the comparison between the parallelized CPU baseline, the sequential kernel and jac_gpu2 is shown in Figure 3. The axes are once again representing a logarithmic scale. The data shows that the baseline version is more efficient for small grid sizes (until N becomes higher than 96), most likely because the problem size is simply too small to fully utilize the number of cores in GPU. For small problem sizes, the superior clock speed of the CPU cores and the lack of overhead from data transfer implies that the CPU will perform better. For larger problem sizes however, the number of execution units becomes the determining factor. For the largest matrix size selected ($N = 768$), the kernel's performance reaches a peak of approximately 150 GFlops.

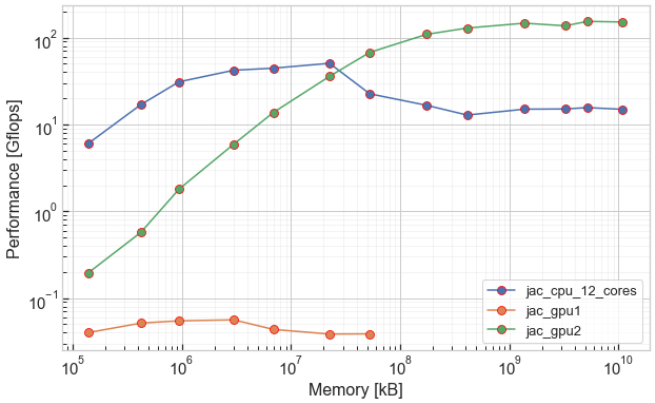

Fig. 3. GFlops performance comparison of the jac_gpu2 kernel against the 12-core CPU baseline and jac_gpu1

In order to understand where the performance bottlenecks are occurring, NVIDIAs visual profiler *nvvp* can be used to analyze performance. From the *nvvp* performance analysis in Figure 4, we can see that the computational operations, namely the arithmetic and control-flow operations, are being utilized at only approximately 20% of their capacity, while the device memory operations are close to being fully utilized (at approx. 85% of the maximum). This indicates that the computational problem is memory-bound.

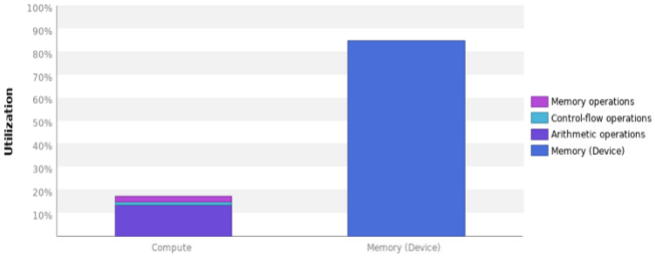

Fig. 4. Performance analysis of the jac_gpu2 kernel using the *nvvp* visual profiler. There is a clear bottleneck at the GPU memory level.

This behavior is an expected one, since the memory is globally fetched and only very few numerical operations are performed between two consecutive memory manipulation operations. Investigating further into the *kernel memory* analysis of the profiler (with results seen in Figure 5), it is easy to both confirm the memory-bound behavior of our kernel and also to identify which type of memory is being the bottleneck: the device memory, i.e. global memory.

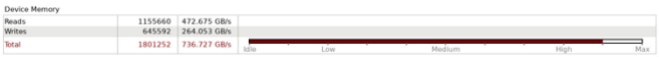

Fig. 5. Kernel memory analysis of the jac_gpu2 kernel using the *nvvp* visual profiler. Global memory is the source of bottleneck.

Since the code is limited by the memory bandwidth, there are multiple ways of improving this kernel. A quick way to improve the kernel's performance is to increase the available bandwidth. The simplest way to do that with the available hardware is to use more than one GPU. With twice the GPUs, the available bandwidth is doubled. This option is investigated in the next subsection (*subsection C*).

Another useful approach is to reduce the needed bandwidth. One way to do that is to restructure the way the work is distributed among the threads. It might be possible to distribute the work in such a way that the most of the data which are far from the grid point can be shared in the "*shared memory*" of the blocks. Therefore, if those points which are far away in memory can be cached in shared memory, the L2 cache could be utilized more effectively.

Yet another way of reducing the bandwidth is to reduce the overall memory footprint. As it is written, the code uses double precision for all the grid values. Depending on the application, single precision floating point might be sufficient. Using "single"-precision instead of "double" would effectively halve the memory footprint and more importantly the bandwidth. With the same bandwidth, it possible to transfer twice as many "single"-precision numbers at the same time. Using "single"-precision also has the added benefit that the GPU cores have more units for processing "single"-precision numbers. The downside is of course that some situations might depend on having full "double" precision, so this tactic is not universally applicable.

C. *MultiGPU kernels*

To achieve even better parallelization opportunities, there is no need for limiting our execution to one single GPU. This next algorithm deals with running the Jacobi differential equation solver method on two parallel GPUs at the same time. The algorithm is memory-bound, but by using two GPUs simultaneously, it is possible to effectively double the memory bandwidth available. Of course, achieving higher throughput through more GPUs does have its algorithmic drawbacks as well, since the problem must be split equally among the two GPUs and it is essential to understand how to connect the matrix update computations around the split.

The performance of the dual-GPU kernel can be seen in Figure 6. Please note that the Y-axis is now linear (unlike in the previous plots).

For small problem sizes, the added overhead of communication between GPUs reduces the performance of the dual-GPU kernel, compared to the single-GPU. It is only when the problem reaches a certain size that the benefit of having twice the resources truly begins to show.

In our tests seen from Figure 6, for grid sizes below $N = 256$, the kernel has indeed shown reduced performance compared to the naive multi-threaded version, due partly to the overhead necessary to communicate between GPUs, however it can be seen that this difference is small (generally remaining within 15 GFlops difference), whereas for large matrix sizes, the multi-GPU version definitely shows its benefits, with usual improvements in performance of 25-35 GFlops.

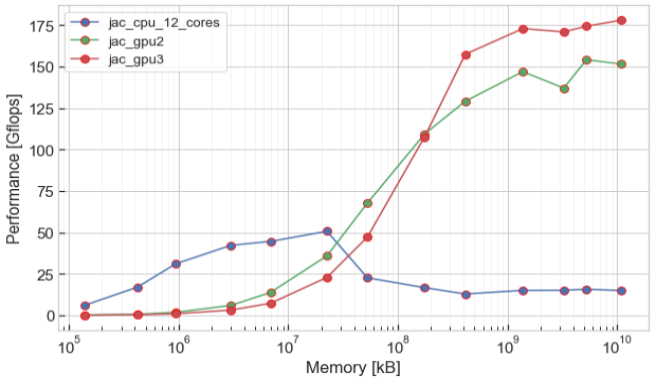

Fig. 6. GFlops performance comparison of the jac_gpu3 kernel against the 12-core CPU baseline and jac_gpu2

TABLE V. SPEED-UP COMPARISON BETWEEN jac_gpu3, THE 12-CORE CPU BASELINE AND jac_gpu2

Algorithm	Time (s) for $N = 768$	Speed-up (compared with baseline)
CPU Baseline (12-cores)	242.402	1.0000x (baseline)
jac_gpu3	20.351	11.911x
jac_gpu2	23.914	10.136x

It can be argued that increase in throughput speed would be even larger when utilizing even higher matrix sizes. The same phenomenon happens with the time measurements between the two kernel versions: for larger matrix sizes, jac_gpu3 is much faster time-wise than its counterpart, jac_gpu2. Table V confirms such a speedup increase.

D. *Naïve multi-threaded kernel with termination criterion*

Finally, the kernel version of the Jacobi algorithm implemented here will be similar to the naive multi-threaded kernel jac_gpu2, with the addition of a stopping parameter representing the norm of the difference between the current and previous value at a grid point.

When analyzing the performance of this kernel, it is essential to discuss how large should the number of maximum iterations max_iter be.

More specifically, since it is interesting to select a max_iter large enough, so that some of the smaller matrix problems converge before reaching the full number of iterations, this whole section has been dealing with speed-ups and performance analysis on new data from all GPU kernels, executed for 5000 iterations. This is done in order to have the fairest comparison possible. The 5000 iterations have also been taken for the CPU 12-core baseline.

For this reason, take notice that the speed-up values in Table VI and Figure 7 are different than those from previous tables, due to this change. N has been selected with respect to a relatively small matrix size, more specifically Table VI shows speed-up comparison for $N = 24$.

From Figure 7, we can notice that this norm-stopping kernel version performs significantly worse in terms of performance than all the other multi-threaded GPU kernels, and, on some portions, worse than the baseline as well. This is however completely according to expectations, since the norm calculation introduces significant overhead.

Therefore, it can be argued that, for this kernel version, performance was never a metric which this kernel was expected to excel at.

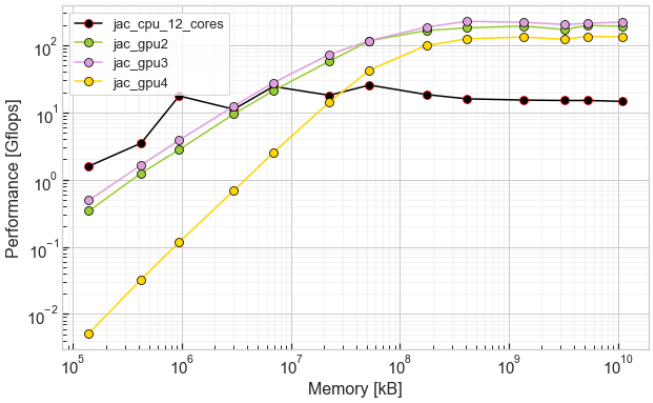

Fig. 7. GFlops performance comparison of the jac_gpu4 kernel against the 12-core CPU baseline, jac_gpu2 and jac_gpu3

TABLE VI. SPEED-UP COMPARISON BETWEEN jac_gpu4, THE 12-CORE CPU BASELINE AND MOST OTHER KERNEL VERSIONS

Algorithm	Time (s) for $N = 24$ and $max\_iter = 5000$	Speed-up (compared with baseline)
CPU Baseline (12-cores)	0.158	1.0000x (baseline)
jac_gpu4	0.431	0.366x
jac_gpu3	0.336	0.470x
jac_gpu2	0.444	0.355x

What may count, however, is the total execution time of this kernel compared to its non-modified counterpart. The true purpose of this kernel is to allow the differential equation to converge as soon as possible, hopefully before the maximum number of iterations is reached.

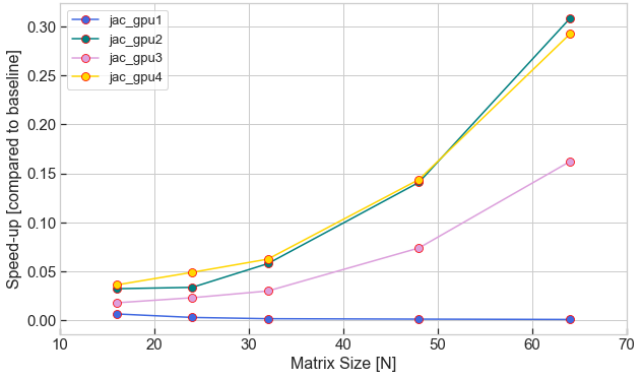

Fig. 8. Speedup plot on small matrix sizes ($N \leq 64$) of the four kernel versions of Jacobi, compared against the 12-core CPU baseline

```
==2983== NVPROF is profiling process 2983, command: ./poisson_gpu 128 5000 1 20. 4
128 1763 0.999926 0.684885 43187.137291
==2983== Profiling application: ./poisson_gpu 128 5000 1 20. 4
==2983== Profiling result:
Type            Time(%)      Time     Calls       Avg       Min       Max   Name
GPU activities:  55.36%   127.36ms      1763  72.242us  70.975us  79.936us  grid_update2(...
                 37.02%   85.183ms      1763  48.317us  46.335us  50.591us  norm_reduction...
                                ...
API calls:       43.29%   211.73ms         4  52.931ms  130.79us  211.30ms  cudaMalloc
                 27.34%   133.72ms      1763  75.847us  73.592us  91.384us  cudaDeviceSynch..
                 25.51%   124.74ms      3530  35.336us  3.5640us  4.1639ms  cudaMemcpy
                                ...
```

Fig. 9. NVIDIA *nvprof* profiler output for jac_gpu4 kernel using its norm reduction

Table VI shows one example where, for a small matrix size and a sufficiently large number of iterations selected, jac_gpu4 ends its execution faster than jac_gpu2. However, this effect will not be seen on large-enough matrix sizes, and on small matrix sizes, even if the jac_gpu4 slightly

outperforms the speed of jac_gpu2, the execution times are much slower than compared to the 12-core CPU baseline.

Therefore, there is little reason for this kernel function to be used instead of the CPU parallel function for small matrices, or the jac_gpu3 kernel for larger matrices. Zooming into the speedup-plot shown in Figure 8, it can be seen that, for a matrix size becoming "large-enough" as early as for $N = 64$, the jac_gpu2 counterpart will outperform jac_gpu4 in execution speed.

Analyzing the overhead management through NVIDIAs $nvprof$ command-line profiling tool, from Figure 9, the GPU and host activity can be seen for the jac_gpu4 kernel, when executed upon the grid size of $N = 128$, a maximum of $max\_iter = 5000$ iterations and a tolerance of 1.00. As it can be seen from the results, there is a very large overhead of calling the $norm\_reduction()$ function, and this norm-difference algorithm accounts for a GPU kernel overhead of 37% of the total resource utilization within the GPU.

Contrasting this to Figure 10, which shows the execution profile for the naive multi-threaded kernel, it can be clearly seen that not having a stop-parameter (i.e. *the norm*) allows the GPU to focus 96.6% of its total working time on the actual grid point updates, instead of 55.36% as seen for the norm-implementing version.

```
==5804== NVPROF is profiling process 5804, command: ./poisson_gpu 128 5000 1 20. 2
128 5000 1 0.837684 100140.496437
==5804== Profiling application: ./poisson_gpu 128 5000 1 20. 2
==5804== Profiling result:
Type            Time(%)      Time     Calls       Avg       Min       Max   Name
GPU activities: 96.60%   376.78ms      5000  75.356us  74.239us  79.583us  grid_update2(...
                              ...
API calls:      61.19%   394.69ms      5000  78.937us  4.3930us  94.613us  cudaDeviceSynch..
                32.79%   211.50ms         4  52.876ms  113.69us  211.12ms  cudaMemcpy
                 2.18%   14.093ms         4  3.5233ms  1.8244ms  4.0958ms  cudaMalloc
                              ...
```

Fig. 10. NVIDIA $nvprof$ profiler output for the naïve multithreaded kernel, jac_gpu2

V. Conclusion

Solving the Poisson equation is memory-bound in both theory and real-world results. Using the NVIDIA "*nvvp"* profiler has revealed that the performance was limited by the memory bandwidth. It is suggested that this can be mitigated by switching from double to single precision or by improving access patterns.

From our performance analysis, it can be seen that the single-threaded kernel implementation has lower computational throughput ability than any of the other kernels and also than the CPU baseline. The naive multithreaded kernel jac_gpu2 performs well in terms of GFlops for small matrix sizes, being faster than the multi-GPU version, until N reaches a large-enough size of $N = 256$.

A different implementation was made to use more than one GPU. This way, the available memory bandwidth is effectively doubled.

§The multi-GPU version jac_gpu3 can be used for additional performance on large computations, yet suffers from "GPU–GPU" communication overhead, which is noticeable for matrix sizes with $N \leq 256$. This has improved overall throughput, although the increased overhead made the performance less than 2x the performance of a single GPU.

The modified norm-stopping kernel jac_gpu4 has much lower performance output than all multi-threaded kernel counterparts, due to the additional computations and device-host memory copying.

A stop criterion was added to the single GPU implementation, but the added overhead of computing the reduction for the norm was outweighing the potential reduced runtime. Only for very small matrix sizes or a high maximum number of iterations would the number of iterations skipped outweigh the added overhead.

On the overall, however, all the three multithreaded kernels perform significantly better for this Jacobi problem than their 12-core CPU OpenMP baseline. This consolidates the concept of the GPU's role as throughput accelerator.

References

[1] S. Sbirna and L. S. Sbirna, "Implementation and performance analysis of sequential versus parallelized algorithms for solving multivariate equations of CFD Poisson processes", 2023 27th International Conference on System Theory, Control and Computing (ICSTCC), Timisoara, Romania, pp. 94-99, 2023.

[2] J. Tu, G. H. Yeoh, C. Liu, and Y. Tao, "Computational Fluid Dynamics: A Practical Approach", 4th Edition, Butterworth-Heinemann, 2023.

[3] F. Ghioldi and F. Piscaglia, "A hybrid CPU-GPU paradigm to accelerate reactive CFD simulations", Int J Numer Meth Fluids, pp. 1-28, 2024.

[4] D. Molinero-Hernández, S. R. Galván-González, N. D. Herrera-Sandoval, P. Guzman-Avalos, J. J. Pacheco-Ibarra, and F. J. Domínguez-Mota, "Turbomachinery GPU Accelerated CFD: An Insight into Performance", Computation, vol. 12, issue 3, art. 57, 2024.

[5] S.A. Qazi, A. Saeed, S. Nasir, and H. Omer, "Singular Value Decomposition Using Jacobi Algorithm in pMRI and CS", Appl Magn Reson, vol. 48, pp. 461–47, 2017.

[6] S. Elmisaoui, I. Kissami, and J. M. Ghidaglia, "High-Performance Computing to Accelerate Large-Scale Computational Fluid Dynamics Simulations: A Comprehensive Study", International Conference on Advanced Intelligent Systems for Sustainable Development (AI2SD'2023), Springer, vol. 930, 2024.

[7] S. Josias and W. Brink, "Jacobian Norm Regularisation and Conditioning in Neural ODEs", Artificial Intelligence Research (SACAIR 2022), Communications in Computer and Information Science, Springer, vol. 1734, pp. 31-45, 2022.