Conference on Modelling Fluid Flow (CMFF'25)
The 19th International Conference on Fluid Flow Technologies
Budapest, Hungary, August 26 - August 29, 2025

USING JACOBI METHOD TO SOLVE THE TWO-EQUATION TURBULENCE MODEL FOR PARALLELIZATION ON GPU COMPUTING SYSTEM

Ivan TOMANOVIĆ[1], Srdjan BELOŠEVIĆ[2], Nenad CRNOMARKOVIĆ[3], Aleksandar MILIĆEVIĆ[4]

[1] Corresponding Author. Department of Thermal Engineering and Energy, "VINČA" Institute of Nuclear Sciences – National Institute of the Republic of Serbia, University of Belgrade, Mike Petrovića Alasa 12–14, 11351 Vinča, PO Box 522, Belgrade, 11001 Serbia. Tel.: +381 11 3408551, E-mail: ivan.tomanovic@vin.bg.ac.rs
[2] Department of Thermal Engineering and Energy, "VINČA" Institute of Nuclear Sciences – National Institute of the Republic of Serbia, University of Belgrade, Mike Petrovića Alasa 12–14, 11351 Vinča, PO Box 522, Belgrade, 11001 Serbia. E-mail: v1belose@vin.bg.ac.rs
[3] Department of Thermal Engineering and Energy, "VINČA" Institute of Nuclear Sciences – National Institute of the Republic of Serbia, University of Belgrade, Mike Petrovića Alasa 12–14, 11351 Vinča, PO Box 522, Belgrade, 11001 Serbia. E-mail: ncrni@vin.bg.ac.rs
[4] Department of Thermal Engineering and Energy, "VINČA" Institute of Nuclear Sciences – National Institute of the Republic of Serbia, University of Belgrade, Mike Petrovića Alasa 12–14, 11351 Vinča, PO Box 522, Belgrade, 11001 Serbia. E-mail: amilicevic@vin.bg.ac.rs

ABSTRACT

The advancement in computer development in recent years saw introduction of powerful graphics processing units (GPUs) intended as general-purpose computation units, beside their core purpose. To better accommodate the existing computational fluid dynamics (CFD) code with a k-ε turbulence model to be able to run on an upcoming generation of GPUs, the existing computer codes must undergo modifications due to very different nature of GPU architecture, compared to that of a central processing unit (CPU). Due to this, a Jacobi method is used instead of Gauss-Seidel based solvers. Application of Jacobi on a CPU predictably leads to a slower execution of code, due to its slower convergence rate. However, its highly parallel nature makes it very suitable for execution on modern GPU. The subroutines both for kinetic energy and its dissipation rate originally use Gauss-seidel based tridiagonal matrix algorithm (TDMA) solver, highly optimized to run on a single computational thread. While it can be modified to run on multiple computational threads, it scales poorly with their number increase. On the other hand, Jacobi offers full parallelism, and simple code implementation, regardless of the problem scale, avoiding conflicting memory operations at the same time, but at the cost of double memory consumption. The code itself is ported to GPU using OpenACC directives in compute unified device architecture (CUDA) FORTRAN. The 2D monophase turbulence flow test-case is solved on an orthogonal structured grid, and the parallel nature of Jacobi gives us the ability to solve equations in each control volume individually between the iteration steps, without dependencies on neighbouring cells, leading to a significant number of operations that can be executed in parallel. Due to initial memory operations overhead cost in time, the model must solve sufficiently large problem to be able to outperform fast CPU.

Keywords: CFD, GPU, Jacobi method, OpenACC, parallelization, turbulence

NOMENCLATURE

CLK	[*Hz*]	single processing unit clock speed
E	[-]	per core efficiency
n	[-]	input size of problem
p	[-]	number of processing units
S	[-]	speedup of the parallel algorithm
T(n)	[-]	total operations count
t	[*s*]	execution time

Subscripts and Superscripts

b, base	base (reference) case/value
overhead	overhead time or operations
parallel	executed in parallel
rw	real-world conditions
serial	executed in serial

1. INTRODUCTION

Application of turbulence models in CFD research demands a significant amount of high-performance computational resources. Past century brought us vast number of models specifically created and well optimized to run on single core processing units. Together with those codes less frequently were developed models intended to run on proprietary super computers, often closed to public. Development of computer graphics lead to creation of powerful GPU devices that could readily perform

operations beyond their intended purpose, and found their place in scientific computing. Understanding the potential of GPUs as very new and promising accelerators that can be used in computational research is essential, as they are being developed at high pace, and thus it is of great importance to create highly efficient computer codes that would follow.

The historical development of CFD codes and more importantly, the equation solvers in them followed the development of both personal and HPC computers, starting with the single core codes implementing the tridiagonal matrix algorithm (TDMA) e.g. [1] and Stone's SIP algorithm [2], as well as the parallel implementations of those on multicore CPUs: [3, 4]. Other solvers that include domain decompositions [5, 6], Cyclic Reduction (CR) and Parallel Cyclic Reduction (PCR) [7], Multi-Grid (MG), Colour Checkerboarding (CC) [8], Alternating-Direction Implicit (ADI) [9], and others followed the increase in core count on personal computers, and the early versions of GPUs as well, providing mixed results and success, but often fast solvers that have their own limitations.

The languages initially developed for GPUs, such as OpenCL and CUDA C, provided users with access to their capabilities, but at a cost of a very long learning curve in order to provide good working and well optimized codes. An easier tool that would move burden of sometimes difficult code writing and memory management from the user to the machine was needed, and it came in the form of OpenACC, providing wider audience with access to GPU device performance, while maintaining simplicity in code writing. A good example of difficulties that had to be overcome when writing codes from scratch can be found in work of Cohen and Molemaker [10] who created a massively parallel GPU implementation of fast CFD code for simulation of Rayleigh-Bérnard convection. Creation of such a code in native GPU language, such as CUDA C, is labour intensive task. On the other hand, OpenACC implementations of code nowadays can provide similar performance gains without requiring the user to tackle difficult tasks, including the load distribution and memory management. One of such implementations is by Xia et al. [11] who discussed the conversion of legacy CFD solvers to new hardware, and also presented a model of compressible CFD code implemented on unstructured grid using OpenACC.

Our own implementation is based off TEACH-T CFD code [12], converting and modifying it to allow massive parallelisation of all parts of the code. The turbulence in code is modelled using a well-known k-ε two equation model, and the implementation is done so that the turbulent kinetic energy equation is solved first, due to internal dependencies in the model itself. The entire CFD solver is ported to the GPU to avoid massive overhead times caused by data movements through the PCI-E lanes.

Amongst other changes to the code structure, one of the most important changes in this code is the inclusion of Jacobi based solver for equations, allowing future massive parallelization of code as the GPU computational cores number grows, and at the same time providing noticeable gains at current architecture. While the Jacobi implementation does have its own disadvantages on real-world parallel machines, most important the double memory consumption, and potentially slower convergence on machines with finite number of cores, it can still be the best choice for the newer machines that often have core count proportional to the size of problem solved, as these machines are moving closer to ideal parallel machines. This is also supported by findings of Tsitsiklis [13] who proves that parallel Jacobi iterations are no slower than any parallel Gauss-Seidel variant of that iteration, confirming the almost certain superiority of Jacobi algorithm over Gauss-Seidel [14]. On GPUs with lower computational core counts and low amount of available memory a good compromise could be the use of Red-Black checkerboarding in one or more directions, which is a version of Colour Checkerboarding algorithm, but is not considered as part of this work.

In this work we put our focus on determining the number of computational operations in subroutine for turbulent kinetic energy, setting aside other subroutines, and we use this information to determine potential for speedup of parallel code based on theoretical background set by Amdahl [15] and Gustafson [16]. Tasks in this work are separated to determination of total number of operations in serial code, grouping them in sections that can be parallelized, determining potential sections of code that can be asynchronously executed, upon which we can determine total number of steps on ideal parallel machine. Once these tasks are complete, we turn towards determination of the computational core number-based operation count, which is informative, but insufficient to provide real-world performance benchmarks. To gain insights into the code behaviour on real hardware, we set a base case that should be used as a ground-line benchmark for performance estimation over several grid sizes. We use running time as a reference and convert the operation count criteria into a running time criteria to provide a fair comparison between different platforms. This approach takes into account real-world overhead times, which, as shown, are significant for GPU-based CFD simulation platforms.

2. CODE IMPLEMENTATION

In order to accommodate CFD code with a k-ε turbulence model to run on GPU, an existing code is modified due to different nature of GPU architecture compared to CPU. A 2D monophase turbulence flow test case is solved on an orthogonal structured grid. Instead of Gauss-Seidel the Jacobi method is used,

enabling solving the equations in each control volume with no influence of neighbouring cells, thus offers parallelization potential. The code is ported to GPU by OpenACC directives in CUDA FORTRAN.

Parallelisation of this 2D code opens direct pathway towards creation of parallel 3D codes. Major change to a code would be an introduction of subroutine that would solve the third component of the velocity. Also, given that there are three dimensions it would lead to addition of k loop in most cases, and increased number of operations where new terms and variables are introduced in equations.

A code structure extracted from subroutine used for calculation of turbulent kinetic energy is given in Table 1, accompanied by the number of computational and memory access operations at every group of lines. The number of operations here is determined based on total arithmetic operations at the line/segment of code, as well as the memory read and write operations that took place. It can be noticed that many operations are enclosed in loops, thus increasing the number of operations depending on the loop size. The total number of operations, as can be noticed, depends on the problem size, but it can be estimated roughly for known size of computational domain. For example, the first loop over interior grid has in single pass total of 113 computational operations, that needed 109 memory reads, and wrote to the memory total of 29 times, but this is repeated (nj-2)·(ni-2) times in total. Sample number of operations for different grid sizes is given in Table 2. In all cases number of swipes (nswp), which tells us how many times Jacobi solver loop is repeated in each iteration step, is assumed 30.

Table 1. Number of computational and memory operations in subroutine for turbulent energy

Pseudo code	calculation	memory	
		read	write
loop over interior grid			
for j = 2 **to** nj – 1			
for i = 2 **to** ni – 1			
calculate areas	3	6	3
calculate volume	2	3	1
calculate conv. coef.	16	16	4
calculate diff. coef.	20	20	4
calculate source terms	61	59	11
update source terms	11	15	6
end			
end			
in total: **(nj – 2)·(ni – 2)·**	**113**	**109**	**29**
wall conditions: top wall			
initial ops	**1**	**3**	**2**
for i = 2 **to** ni – 1			
calculate y^+	7	9	3
update source terms	31	34	3
end			
in total: **(ni – 2)·**	**38**	**43**	**6**
: side wall			
for j = jstp **to** nj – 1			
calculate y^+	7	9	3
update source terms	31	34	3
end			
in total: **(nj – jstp)·**	**38**	**43**	**6**
: symmetry axis			
for i = 2 **to** ni – 1			
calculate source terms	15	22	9
update source terms	2	3	2
end			
in total: **(ni – 2)·**	**17**	**25**	**11**
res. and under-relaxation			
for j = 2 **to** nj – 1			
for i = 2 **to** ni – 1			
calculate $a_{p,te}$	4	5	1
calculate residual	10	11	1
calculate under-relax.	4	6	1
end			
end			
in total: **(nj – 2)·(ni – 2)·**	**18**	**22**	**3**
iterative solver loop			
for swipe = 1 **to** nswp			
for j = 2 **to** nj – 1			
for i = 2 **to** ni – 1			
calculate te_{new}	13	13	1
update residual	3	4	2
end			
end			
for j = 2 **to** nj – 1			
for i = 2 **to** ni – 1			
calculate te_{new}	0	1	1
end			
end			
in total: **n· (nj – 2)·(ni – 2)·**	**16**	**18**	**4**

Table 2. Estimated number of serial operations $T_{serial}(n)$ for different grid sizes

Grid	calculation	Memory access	
		read	write
32×16	$2.6 \cdot 10^5$	$2.8 \cdot 10^5$	$6.4 \cdot 10^5$
64×32	$1.1 \cdot 10^6$	$1.3 \cdot 10^6$	$2.8 \cdot 10^5$
128×64	$4.8 \cdot 10^6$	$5.3 \cdot 10^6$	$1.2 \cdot 10^6$
256×128	$2.0 \cdot 10^7$	$2.1 \cdot 10^7$	$4.9 \cdot 10^6$
512×256	$7.9 \cdot 10^7$	$8.7 \cdot 10^7$	$2.0 \cdot 10^7$
1024×512	$3.2 \cdot 10^8$	$3.5 \cdot 10^8$	$7.9 \cdot 10^7$
2048×1024	$1.3 \cdot 10^9$	$1.4 \cdot 10^9$	$3.2 \cdot 10^8$
4096×2048	$5.1 \cdot 10^9$	$5.6 \cdot 10^9$	$1.3 \cdot 10^9$

From this table we can see that the number of calculation operations grows from $2.6 \cdot 10^5$ all the way to $5.1 \cdot 10^9$ as the grid size grows from 32×16=512 elements to 4096×2048=$8.4 \cdot 10^6$ elements. Most of these operations are contained in loops that can be readily parallelised, and as such contribute to significant reduction in number of operations when the parallelisation is introduced. The rough estimation of number of parallel operations on an ideal parallel machine is $T_{parallel}(n)$ = 106 (due to

some parts of loops, or entire loops being able to execute asynchronously (i.e. parallel one to another), taking into account 16 successive parallel operations that include massive arrays inside the loops that can be executed in parallel, 3 operations on arrays that execute in series, 30 times each, as well as internal dependencies that enforce us to wait for computation of some variables before we are able to compute others.

3. CODE SPEEDUP AND EFFICIENCY

In theory, a proper way to estimate algorithm speedup and efficiency is to define the number of operations $T(n)$ executed to solve the problem of given size n. From here we can estimate the number of parallel operations necessary to reach the same solution under ideal conditions, with finite number of computational units, based on considerations brought by Amdahl [15]:

$$T_{parallel}(n) = \frac{T_{serial}(n)}{p} \tag{1}$$

This is true for each and every loop found in the problem we solve if it is readily parallelizable, and if we consider execution on an ideal parallel machine.

As an example, we will apply Eq. (1) to the calculation operations from Table 2, and compare data between the parallel CPU (Intel Xeon E5-1620 v3) with 8 threads against the GPU (NVIDIA Quadro RTX 4000) with 2304 cores. The $T_{parallel}(n)$ for both is given in Table 3, but it must be noted that this value reflects purely theoretical performance and it ignores overhead times necessary for load distribution, memory transfer times when simulations are done on GPU, other operational overheads, as well as the individual core performance (core speed, amount of available cache, bandwidth, etc.).

Table 3. Estimated number of computational operations $T_{parallel}(n)$ for different accelerators

Grid size	p = 8 (CPU)	p = 2304 (GPU)
32×16	$3.2 \cdot 10^4$	$1.1 \cdot 10^2$
64×32	$1.4 \cdot 10^5$	$5.0 \cdot 10^2$
128×64	$6.0 \cdot 10^5$	$2.1 \cdot 10^3$
256×128	$2.4 \cdot 10^6$	$8.5 \cdot 10^3$
512×256	$9.9 \cdot 10^6$	$3.4 \cdot 10^4$
1024×512	$4.0 \cdot 10^7$	$1.4 \cdot 10^5$
2048×1024	$1.6 \cdot 10^8$	$5.5 \cdot 10^5$
4096×2048	$6.4 \cdot 10^8$	$2.2 \cdot 10^6$

Given that both parallel CPUs and GPUs are not the ideal machines, they need additional resources to communicate, distribute and balance the load over their respective computational cores. This leads to the introduction of additional overhead operations, caused by additional communication and memory management, especially if we take into account memory transfers that are unavoidable in CPU-GPU communication in real-world runs, as an expansion of Amdahl's law by Gustafson [16], Eq. (2):

$$T_{parallel,rw}(n) = \frac{T_{serial}(n)}{p} + T_{overhead} \tag{2}$$

As the complexity of code structure increases it becomes more difficult to accurately estimate the number of operations. Also, in Eqs. (1) and (2) it is noticeable that if the $T(n)$ is less or equal to the number of processing units we can get the number of parallel operations smaller than 1, which is impossible, and such cases rather tend to have smallest, but finite number of operations that should be determined by estimating the number of parallel computation steps. While the total number of operations remain the same between serial and parallel codes, the number of execution steps is usually reduced as the parallelism is introduced, virtually converting a large number of steps found in parts of serial code to a single step on massively parallel machine. However, we still have to follow some execution thread, given that prior to executing some operations others must complete, creating a new kind of overhead that adds to previous, and a new issue in operation counting to overcome these issues, it would be easier to measure execution time, instead of operation counts in real life codes.

Furthermore, the physical differences in performance of different accelerators include additional challenges when estimating the performance originating from individual processing unit and memory speed. Those could be partially addressed by expressing the $T_{parallel,rw}(n)$ as a function of those variables, by expressing performance per normalised processing unit speed:

$$T_{parallel,rw}(n) = \frac{T_{serial}(n)}{p} \cdot \frac{CLK_{base}}{CLK} + T_{overhead} \tag{3}$$

On the other hand, the execution time as a measurable output, takes into account all overheads, without focusing on them individually, as well as the hardware constraints that include clock and memory speeds, limitations in bandwidth, and all other influences that occur, thus making it an indicator that is easier to measure. We can in a similar manner estimate the execution time that should be expected from an ideal parallel machine, based on its number of processors:

$$t_{parallel}(n) = \frac{t_{serial}(n)}{p} \tag{4}$$

Also, in the similar manner we can include overhead time to this equation:

$$t_{parallel,rw}(n) = \frac{t_{serial}(n)}{p} + t_{overhead} \tag{5}$$

It is important to note that, the parallel execution time expressed in this manner is the time per processing unit found in accelerator (parallel CPU or GPU), and as such has similar limitations as the number of operations from Eq. (3).

The overhead time, as a difference between real-world and ideal runs, can be determined by

comparing the actual measured execution time in real world runs to that obtained from Eq. (4), giving us the following:

$$t_{overhead} = t_{parallel,rw}(n) - t_{parallel}(n) \quad (6)$$

To effectively estimate both the speedup and efficiency of parallel execution on parallel CPU and GPU we need a solid baseline case that will serve as a reference. To this end we will utilize a code compiled for a single threaded run with same Jacobi algorithm, with all compiler optimizations applied. This will give us an insight into an actual execution time which we will denote as $t_{serial,CPU}(n)$ in analysis. From this $t(n)$ we will determine $t_{parallel,CPU}(n)$ and $t_{parallel,GPU}(n)$ using the Eq. (4). The number of processing units/threads p for CPU and GPU used in simulations is 8 and 2304, respectively.

The speedup of the parallel algorithm, both theoretical and real-world, can be determined by comparing it to its serial counterpart, per Eqs. (7) and (8):

$$S(p) = \frac{t_{serial,b}(n)}{t_{parallel}(n)} \quad (7)$$

$$S_{rw}(p) = \frac{t_{serial,b}(n)}{t_{parallel,rw}(n)} \quad (8)$$

Expressing the speedup over the available number of processing units provides us with an insight into the efficiency of parallelization, Eqs. (9) and (10), where values $E \approx 1$ indicate highly efficient parallelization, while the values of $E \ll 1$ indicate significant slowdowns due to overhead or insufficient utilization of individual processing units, this is in relation to the findings of Hill and Marty [17] who shown that locally inefficient core design can be globally efficient at large scale.

$$E(p) = \frac{S(p)}{p} \quad (9)$$

$$E_{rw}(p) = \frac{S_{rw}(p)}{p} \quad (10)$$

Results for simulations of turbulent flow model in 2D channel on parallel CPU and GPU, using Jacobi model to solve equations are given in Tables (4) and (5). Test cases are named after the accelerator used and number of cells in simulation grid for that case. All test case times are given for 20 000 iterations with double precision floating point numbers on numerical grids consisting of 8192 (8k), 23768 (32k), 131072 (131k), 524288 (524k), 2097152 (2M), and 8388608 (8M) elements. From the Table 4 it is noticeable that there is significant growth in overhead execution time for the parallel CPU execution. The growth in the overhead time with the grid size for the GPU is significantly lower, but it makes nearly entire difference between real-world and ideal parallel execution.

From these results, we notice that expected parallel execution times, using above Eqs. are significantly lower than the real-world performance. This is something that can't be avoided, but can be reduced by proper hardware selection and different solver choices depending on the scale of the problem.

Table 4. Simulation times and overhead times

CASE	$t_{serial,b}$	$t_{parallel}$	$t_{parallel,rw}$	$t_{overhead}$
Eq.		(4)		(6)
CPU8k	25.8	3.22	11.41	8.19
CPU32k	152.4	19.05	57.37	38.32
CPU131k	722.5	90.32	301.45	211.13
CPU524k	5860.5	732.56	4197.20	3464.63
CPU2M	25327.4	3165.93	23675.66	20509.73
CPU8M	120473.7	15059.21	82879.52	67820.31
GPU8k	25.8	0.011	92.50	92.49
GPU32k	152.4	0.066	125.73	125.66
GPU131k	722.5	0.313	158.14	157.82
GPU524k	5860.5	2.544	413.59	411.05
GPU2M	25327.4	10.993	1410.14	1399.15
GPU8M	120473.7	52.289	5366.13	5313.84

Following the results from the Table 4 speedup and computational efficiency of each test case is calculated. Results provided in Table 5 indicate that at larger grid sizes GPU can provide significant speedup in real-world conditions, providing real world benefits to the user. However, efficiency of the parallelisation from the same table indicates a very high overhead on GPU.

Table 5. Speedup and per core efficiency

CASE	$S(p)$	$S_{rw}(p)$	$E(p)$	$E_{rw}(p)$
Eq.	(7)	(8)	(9)	(10)
CPU8k	8	2.26	1	0.28
CPU32k	"	2.66	"	0.33
CPU131k	"	2.40	"	0.30
CPU524k	"	1.40	"	0.17
CPU2M	"	1.07	"	0.13
CPU8M	"	1.45	"	0.18
GPU8k	2304	0.28	"	0.000121
GPU32k	"	1.21	"	0.000526
GPU131k	"	4.57	"	0.001983
GPU524k	"	14.17	"	0.006150
GPU2M	"	17.96	"	0.007796
GPU8M	"	22.45	"	0,009744

It can be seen that the increase of grid size significantly reduces computational capabilities of parallel CPU, leading to a lower speedup performance over the base case after certain grid size, as well as to a drop in per core efficiency at the same time. However, on the GPU, the speedup is generally increasing with the grid size, and it should follow that trend until the physical limitations of the hardware are met. At the same time, we can notice significant rise in per core efficiency, even though its value remains very low, indicating the underutilization of computational resources and the existence of a significant overhead.

4. SUMMARY

A CFD code with a two-equation turbulence model was modified to run on GPU architecture. Monophase 2D turbulence flow test case was solved on an orthogonal structured grid, while the Jacobi solver is used for efficient parallelization. The code speedup and computational efficiency were analysed over the grid size and compared between parallel and serial execution of the code, both on CPU and GPU.

A significant disparity between results that can be expected in theory, and the ones obtained by solving the real-world problems that include turbulence modelling on parallel machines was observable. While the majority of the work executed can be readily parallelized, and the number of operations can in theory drop to a value close to 106 (for ideal parallel machine), in reality different kinds of overhead become an issue for performance, and as such significantly overload accelerators leading to slower than expected execution of programs.

Even though the existing accelerators do suffer from issues caused by different overheads, real-world performance improvements are noticeable, and they outperform each other at different problem sizes.

The use of Jacobi solver has potential on both current and upcoming GPU and similarly structured accelerators/platforms, due to its low number of operations, even though it is the slowest to converge to a solution, and takes double memory space. An alternative to it would be the implementation of red-black Gauss-Seidel algorithm, using less space, but operating only on one half of the problem per dimension of array, to avoid recursive dependencies that would prevent the parallelization.

Considering the implementation from the aspect of speedup and per core computational efficiency, it is noticeable that both parallel CPU and GPU outperform single core CPU by a significant amount. However, the speedup factor for parallel CPU slowly drops as the problem size increases. On the other hand, the speedup factor of GPU grows with the problem size at larger scales, and should keep growing until it meets hardware limitations. Efficiency-wise this parallel CPU shown highest value in cases CPU32k and CPU131k with a value over 0.30. Above those cases efficiency slowly drops. On the other hand, GPU efficiency per core is very low, in range of 10^{-3}, but it constantly grows, and we must keep in mind that these computational nodes (CPU and GPU cores) use significantly different amounts of power.

An average CPU core consumption is around 5-15W, or even high as 20-30W per core depending on its purpose and generation, while the GPU cores stay around 10-30mW power per core, leading to values for our devices that would fit range of 80-120W for CPU (total power 140W) and 45-70W for GPU for all CUDA cores (total power 160W). From this a power consumption per run in case CPU524k would be around 0.163 kWh, and similar case GPU524k will be in the range of 0.018 kWh on a GPU based accelerator.

The real-world performance of GPU depends on its characteristics. While the available number of computational cores is significant factor, major problems and performance losses come from the architecture of Streaming Multiprocessors and different Compute Capability versions, with significant differences in ability to manage memory, schedule tasks, locally store data, or utilize some other special functions. Even though improper coding for targeted family of GPUs leads to performance penalties, noticeable issues come from the outside parameters – PCIe lane size, causing the very slow data movements between CPU and GPU.

As said, current CPU-GPU architecture is very limited considering the data movements between CPU and GPU, creating inevitable overhead delays. These problems should be overcome as the new architectures and improvements to the existing ones are introduced, as well as their availability and affordability grows on the market. Some of promising current/future platforms/architectures for development of highly parallel CFD applications such as NVIDIA's Grace Hopper and Grace Blackwell, or the recently announced AMD CPUs with NPU Ryzen AI Max+ 395 should reach wider market in the following years. OpenACC expansion to those accelerators should follow, providing smooth and practical transition of existing codes to these new platforms with higher memory bandwidth, capacity and raw computational power.

ACKNOWLEDGEMENTS

This work has been supported by the Ministry of Science, Technological Development and Innovation of the Republic of Serbia (Contract Annex: 451-03-136/2025-03/ 200017).

REFERENCES

[1] Ferziger, J. H., Perić, M., and Street, R. L., 2020, *Computational Methods for Fluid Dynamics* (4th ed.), Springer International Publishing, Cham.

[2] Stone, H. L., 1968, "Iterative Solution of Implicit Approximations of Multidimensional Partial Differential Equations", *SIAM J Numer Anal*, Vol. 5, pp. 530–558.

[3] Reeve, J. S., Scurr, A. D., and Merlin, J. H., 2001, "Parallel versions of Stone's strongly implicit algorithm", *Concurr Comput*, Vol. 13, pp. 1049–1062.

[4] Chathalingath, A., and Manoharan, A., 2019, "Performance Optimization of Tridiagonal Matrix Algorithm [TDMA] on Multicore Architectures", *International Journal of Grid*

and High Performance Computing, Vol. 11, pp. 1–12.

[5] Ahmadi, A., Manganiello, F., Khademi, A., and Smith, M. C., 2021, "A Parallel Jacobi-Embedded Gauss-Seidel Method", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 32, pp. 1452–1464.

[6] Amritkar, A., Tafti, D., Liu, R., Kufrin, R., and Chapman, B., 2012, "OpenMP parallelism for fluid and fluid-particulate systems", *Parallel Comput*, Vol. 38, pp. 501–517.

[7] Souri, M., Akbarzadeh, P., and Mahmoodi Darian, H., 2020, "Parallel Thomas approach development for solving tridiagonal systems in GPU programming − steady and unsteady flow simulation", *Mechanics & Industry*, Vol. 21, p. 303.

[8] Parker, J. T., Hill, P. A., Dickinson, D., and Dudson, B. D., 2022, "Parallel tridiagonal matrix inversion with a hybrid multigrid-Thomas algorithm method", *J Comput Appl Math*, Vol. 399, p. 113706.

[9] Wei, Z., Jang, B., Zhang, Y., and Jia, Y., 2013, "Parallelizing Alternating Direction Implicit Solver on GPUs", *Procedia Comput Sci*, Vol. 18, pp. 389–398.

[10] Cohen, J. M., and Molemaker, J., 2009, "A Fast Double Precision CFD Code using CUDA", *J Physical Soc Japan*, Vol. 1, pp. 237–341.

[11] Xia, Y., Lou, J., Luo, H., Edwards, J., and Mueller, F., 2015, "OpenACC acceleration of an unstructured CFD solver based on a reconstructed discontinuous Galerkin method for compressible flows", *Int J Numer Methods Fluids*, Vol. 78, pp. 123–139.

[12] Tomanović, I., Belošević, S., Milićević, A., Crnomarković, N., Stojanović, A., Deng, L., and Che, D., 2024, "CFD Code Parallelization on GPU and the Code Portability", *Adv Theory Simul*, p. 2400629.

[13] Tsitsiklis, J. N., 1989, "A Comparison of Jacobi and Gauss-Seidel Parallel Iterations", *Appl Math Left*, Vol. 2, pp. 167–170.

[14] Smart, D., and White, J., 1988, "*Reducing the Parallel Solution Time of Sparse Circuit Matrices Using Reordered Gaussian Elimination and Relaxation*", VLSI Memo No. 88-440, Massachusetts Institute of Technology.

[15] Gene, D. R., and Amdahl, M., n.d."Validity of the single processor approach to achieving large scale computing capabilities", *Spring Joint Computer Conference*, Atlantic City, New Jersey., 483–485.

[16] Gustafson, J. L., n.d."REEVALUATING AMDAHL'S LAW", *Commun ACM*, Vol. 31, pp. 532–533.

[17] Hill, M. D., and Marty, M. R., 2007, "Amdahl's Law in the Multicore Era", Technical Report #1593, University of Wisconsin-Madison.