2023 IEEE 18th International Conference on Computer Science and Information Technologies (CSIT) | 979-8-3503-6046-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/CSIT61576.2023.10324243

Performance of the modern parallel programming approaches: a case study

Volodymyr Fedynyak*, Oleksa Hryniv*, Bohdan Vey*, Oleg Farenyuk*†
* *Faculty of Applied Sciences of Ukrainian Catholic University, L'viv, Ukraine*
† *Institute for Condensed Matter Physics of the National Academy of Sciences of Ukraine, L'viv, Ukraine*
e-mail: indrekis@icmp.lviv.ua

***Abstract*—This research is devoted to a quantitative comparison of the performance of several parallel programming approaches and compares their computational performance. Comparison is performed for the Computational Dynamics Problem solved by the MacCormack scheme. Parallel computation properties of this sample problem task are well-understood. The parallel programming techniques were chosen considering the recent trends in high-performance computing. Both high-level framework-based implementations (using OneAPI DPC++ and ArrayFire) and low-level implementations (based on CUDA C++) are reviewed and their performance is compared. Additionally, single SMP systems with multiple CUDA-capable GPUs were studied using GPUDirect and Unified Memory technologies. Walltime was used as a performance metric. The comparison was performed using Student's t-test for the Gauss-distributed experimental results and the non-parametric Wilcoxon signed-rank test – for other distributions. The results show that CUDA-based solutions outperform other approaches though development time considerations often can favor more high-level approaches.**

***Index Terms*—CFD, parallel programming, multithreading, GPU, CUDA, MPI, benchmarking, statistical hypothesis testing**

I. Introduction

There is a great demand to solve computationally complex problems. Single-CPU solutions, being now close to the hard physical limits of the current technologies [1] are not adequate for solving them, so there is a large demand for the parallelized approaches, designed for both CPUs and dedicated processing units. This demand leads to cause to appear a large variety of high-performance computing (HPC) techniques spanning multiple levels of abstraction, each focusing primarily on one or a few key factors such as performance, scalability, portability, and length of development cycles.

This research aims to compare some of the most established and state-of-the-art relevant frameworks and methods to clarify their advantages in terms of performance, thus establishing a basis for choosing between them. We have selected the computational fluid dynamics (CFD) problem solved with the finite difference method, implemented using simple explicit MacCormack scheme [2] as a gauge for our measurements, as it is well-suited for parallelization.

Implementations of the problem's solution using GPGPU (general-purpose graphical processing unit) tensor libraries (PyTorch, ArrayFire), parallel programming APIs (OneAPI DPC++, OpenMP, OpenMPI), as well as pure NVIDIA CUDA C++ implementations, are examined. The simplicity of usage of the mentioned approaches is reflected in and compared.

II. Background

CFD problems are parallelized rather well and, to some extent, can be regarded as ideal real-world parallel programming problems. A detailed description of the model system and used numerical approaches are presented in our previous work [3]. In this section, we overview approaches to implement them that we believe are most relevant and discuss their underlying ideas and principles.

A. Parallel programming APIs

There are several popular parallel programming models that encapsulate implementation details and present a flexible interface, high stability, and considerable portability.

1) OpenMPI: Message passing interface (MPI) is a standard for High-Performance Computing on Beowulf clusters and other supercomputers. Its model is based on message passing between isolated processes. Recent implementations are CUDA-aware [4], [5]. For this research, OpenMPI implementation was used.

2) OpenMP: Open Multi-Processing is a cross-platform application programming interface for shared-memory parallel programming in C, C++, and Fortran [6]. OpenMP is often used in hybrid models, for example, in combination with MPI running on a cluster or in a multi-GPU CUDA setup.

3) OneAPI DPC++: Data Parallel C++ is a developing cross-architecture programming language based on C, C++, and SYCL [7]. It provides a wide array of explicit parallel primitives. OneAPI [8] is a standard unified programming model, with DPC++ as its core, for solving computationally demanding tasks across various accelerator architectures [9]. Because of its recent introduction, this language and its performance implications are yet to be fully understood.

B. Nvidia CUDA

Nvidia CUDA C/C++ is a C/C++ language extension that provides a simple path for users familiar with C/C++ to quickly write programs for execution by the CUDA-capable GPU. It provides a low-level API for direct management of a GPU's memory model, computational blocks, and inter-device communication. CUDA GPU implies a scalable set of multithreaded streaming multiprocessors (SM) that employ the SIMT (single instruction – multiple threads) approach. CUDA GPU architecture suggests a multi-level hierarchical memory model. Each thread owns a private local memory, while each thread block has a shared memory visible to all its threads.

Since the memories of the CPU and GPUs are physically separated, memory transactions through PCIe are required. It is rather slow in comparison with GPU inner buses. Moreover, transactions between different GPUs are accomplished through system RAM. Over the last few years, several optimization methods have been developed:

- NVIDIA GPUDirect RDMA – technology for direct access between GPUs on PCIe bus using DMA;
- NVIDIA GPUDirect v2 P2P – employs peer-to-peer communication between GPUs via sharing a single bus [10];
- NVIDIA NVLink – additional proprietary bus with bandwidth up to 12 GB/sec [11].

Additional technologies impacting the performance are:

- Unified Virtual Addressing (UVA) – common address space for all devices, which simplifies development but hides the non-uniform nature of memory types which can have drastic performance implications.
- Unified Memory – late technology, complementing the common address space by the idea of the managed memory, which automatically migrates to an allocated place, thus upholding the locality principle [12].

Optimization potential of these technologies for the model problem is studied in this research.

C. GPGPU Tensor libraries

Based on low-level APIs, several lightweight libraries for parallel tensor processing with support for general-purpose GPU computing have been developed.

1) ArrayFire: ArrayFire [13] is a modern high-level C++ parallel programming framework for boosting the computations with GPUs.

It supports computation on the CPU, OpenCL, and CUDA GPUs, but does not directly support thread-level SMP CPU parallelism.

2) PyTorch: PyTorch is a high-level Python 3 Machine Learning framework [14]. It uses a JIT compiler to increase efficiency and can use both CPU and CUDA backends. For the Python implementation, we use the PyTorch tensor processing library.

III. Experimental setup

A. Hardware setup

Experiments were conducted using the following hardware:

- Machine 1:
  - CPU: Intel Core i7-7820X, 3.6GHz, 8 cores;
  - GPUs: 2 x Nvidia GeForce GTX Titan X 11 Gb;
  - RAM: 16 Gb DDR4 SDRAM, 2400Ghz, 2 channels;
  - CUDA: Nvidia CUDA 11.3, GPUDirect P2P.
- Machine 2:
  - CPU: AMD Threadripper 3970X, 3.6GHz, 32 cores;
  - GPUs: 2 x Nvidia GeForce RTX 3090 22 Gb;
  - RAM: 64 Gb DDR4 SDRAM, 2400Ghz, 4 channels;
  - CUDA: Nvidia CUDA 11.4, no GPUDirect P2P.
- Machine 3:
  - CPU: AMD Threadripper 1920X, 3.5GHz, 12 cores;
  - GPUs: 3 x Nvidia GeForce RTX 2080 Ti 11 Gb;
  - RAM: 64 Gb DDR4 SDRAM, 2400Ghz, 4 channels;
  - CUDA: Nvidia CUDA 10.2, no GPUDirect P2P.

B. Solution validation

Each implementation was validated to avoid bias and outliers in performance benchmarking. The interfaces were normalized, and the CFD simulation with fixed parameters was conducted. Then, the calculation results were compared to the results of the reference simulation.

C. Experiments

Several pre-selected benchmarks were conducted using separate subsets of implementations in order to counteract the problem of multiple comparisons. To this end, the Bonferroni correction is used [15]. For each benchmark, some scenario with fixed parameters was chosen. The performance benchmarking results were passed on for further analysis. Wall time of the execution was used as a performance comparison metric.

D. Experimental data analysis

Wall time measurement has a serious drawback: each separate run of the same simulation produces different results. This issue is caused by not easily predictable processes inside the hardware and the operating system: task switches, interrupts servicing, change of the CPU or the GPU frequency, CPU/GPU microarchitecture resources conflicts, etc. So to produce an adequate comparison of the performance for the different approaches, statistical hypothesis testing should be used. For wall time measurement, the distribution of the resulting time intervals is often far from normal, requiring to use of the Kolmogorov-Smirnov test for normality before applying Student's t-test [16]. If the distribution is non-normal (non-Gaussian), we choose to use a non-parametric test, independent of the exact distribution – Wilcoxon test [17]. Obtained non-normal distributions require the usage of the minimal time interval as the final metric [16].

IV. Experimental results and their analysis

A. General comparison

Figure 1 shows a comparison of the performance of the: high-level implementations (ArrayFire, PyTorch, OneAPI DPC++), low-level implementations (SMP C++, CUDA C++), multi-GPU implementation (MPI with ArrayFire CUDA). For each implementation, minimal execution time was obtained. The iteration throughput for each approach is presented on the chart. The performance of CUDA C++ low-level implementation is significantly higher than other ones. Generally, CUDA-based approaches outperform CPU-based, except for OneAPI – it implies experimental support for CUDA. Since ArrayFire requires a sequential solution while running on CPU due to lack of multithreading support, it shows the worst performance results. Since CUDA-based approaches tend to outperform others and are widely used in the machine learning domain, we focus on them in the following experiments.

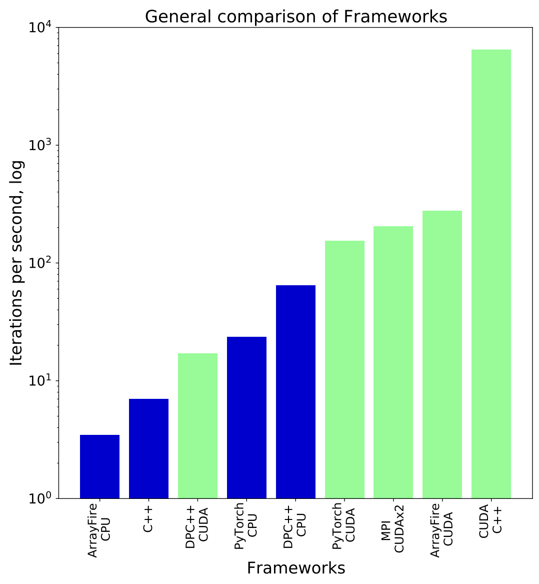


Fig. 1. Throughput of the different approaches for the CFD problem solving. GPU-based approaches are marked with green, CPU-based are marked with blue. Logarithm scale is used. Details in the text.

B. CUDA-based solutions comparison

The performance of the following CUDA-based implementations was compared, using hypothesis testing: PyTorch CUDA, ArrayFire CUDA, CUDA C++, MPI+ArrayFire CUDA x2 (on two GPUs). Distributions are normal (with p-value < 0.01) only for the PyTorch and ArrayFire-based approaches. No two distributions are supposed to be equal with p-value less than 10^{-10}.

Relative pairwise accelerations based on minimal time are presented in Table I. CUDA C++ implementation outperforms all high-level solutions more than four times. A rather unexpected result that Multi-GPU MPI solution using ArrayFire is slower than a single-GPU one can be explained by the small problem size – the speedup from using multi-GPU setups becomes crucial while using huge problem sizes.

TABLE I
THE RELATIVE PERFORMANCE OF THE CUDA-BASED IMPLEMENTATIONS: TIME OF THE METHOD IN THE ROW TO THE TIME OF THE METHOD IN THE COLUMN. AF – ARRAYFIRE.

	AF CUDA	MPI AF CUDA x2	CUDA C++	PyTorch CUDA
ArrayFire CUDA	1.00	0.88	4.26	0.61
MPI AF CUDA x2	1.13	1.00	4.82	0.69
CUDA C++	0.23	0.21	1.00	0.14
PyTorch CUDA	1.64	1.45	6.97	1.00

C. C++ CUDA optimization

1) Memory management: Device-only, Unified (Managed), and Host-pinned memory management scenarios were applied to CUDA C++ implementation. Pinned-memory approach was more than 100 times slower than others, so its results are not included in the final comparison. Each memory management approach was additionally implemented using the device's shared memory. The normality hypothesis was only rejected for the Device-only Shared memory scenario. No two distributions appear to be equal with a p-value less than 10^{-15}, except for the Machine 3 Device only vs Unified case where $p = 9 \cdot 10^{-3}$.

Relative performance of those optimizations are presented in the Table II. The benefits of using device-only memory are highly dependent on the CUDA GPU type, ranging between 2% and 7% speedup compared to using Unified memory for the tested configurations. The same goes for the shared memory optimization, with range from 13% up to 48%.

TABLE II
THE RELATIVE PERFORMANCE OF THE DIFFERENT CUDA MEMORY MANAGEMENT APPROACHES: TIME OF THE METHOD IN THE ROW TO THE TIME OF THE METHOD IN THE COLUMN. A) MACHINE 1, B) MACHINE 2, C) MACHINE 3.

(a)

	Dev. only	Dev. only, Shared	Unified	Unified, Shared
Dev. only	1.00	1.37	0.96	1.37
Dev. only, Shared	0.73	1.00	0.70	1.00
Unified	1.04	1.43	1.00	1.43
Unified, Shared	0.73	1.00	0.70	1.00

(b)

	Dev. only	Dev. only, Shared	Unified	Unified, Shared
Dev. only	1.00	1.10	0.94	1.06
Dev. only, Shared	0.91	1.00	0.85	0.96
Unified	1.07	1.17	1.00	1.13
Unified, Shared	0.94	1.04	0.88	1.00

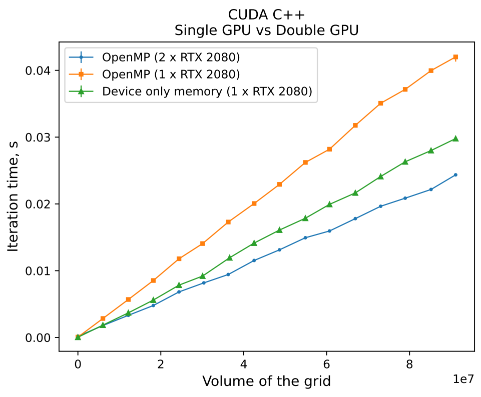
(c)

	Dev. only	Dev. only, Shared	Unified	Unified, Shared
Dev. only	1.00	1.46	0.98	1.45
Dev. only, Shared	0.69	1.00	0.67	0.99
Unified	1.02	1.48	1.00	1.47
Unified, Shared	0.69	1.01	0.68	1.00

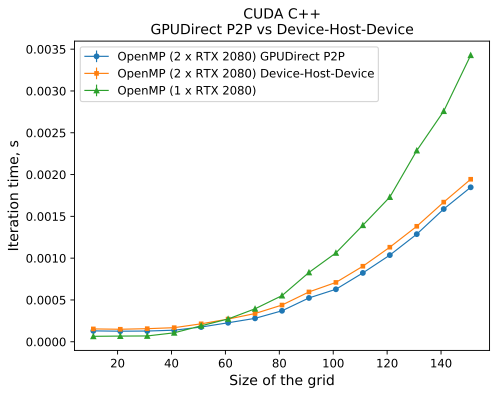
2) Multi-GPU communication: Nvidia CUDA C++ implementation was augmented using OpenMP to achieve multi-GPU support. Both GPUDirect P2P and device-host-device communication techniques were implemented. Experiments were conducted for a range of grid sizes.

Double-GPU test performance increases with problem size almost two times faster than Single-GPU performance (Figure 2). However, the shared memory-based approach outperforms the single-GPU version of CUDA C++ OpenMP, thus double-GPU setup implies less than two times the speedup compared to the most efficient single-GPU implementation.

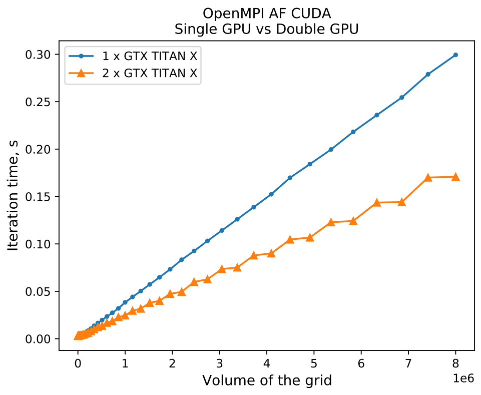
The comparison between GPUDirect and Device-Host-Device communication shows that a multi-GPU setup outperforms a single-GPU one starting from a grid size of 50 for GPUDirect communication and 60 for Device-Host-Device



(a)



(b)



(c)

Fig. 2. Iteration time as a function of the system size. All error bars would be smaller than the line width. (a) Single GPU vs double GPU, using GPUDirect, Machine 3. (b) GPUDirect vs Device-Host-Device, Machine 1. (c) ArrayFire performance for the one GPU vs two-GPU, Machine 1.

communication. In general GPUDirect technology gives a slight speedup, vanishing with the problem size growth.

MPI-driven ArrayFire CUDA implementation is most simple from the development point of view of using a multi-GPU setup, but it is significantly slower than pure C++ CUDA implementations. The reason is that the GPUDirect feature isn't supported by MPI without the use of CUDA C++, so the communications are done through the CPU.

V. Conclusion

In this research, we investigated a wide array of parallel programming techniques, spanning multiple levels of abstraction, available for the CPU and CUDA GPUs. Wall time measurements were used. Statistical hypothesis testing was used to compensate inherent noise of such measurements.

As expected, the lowest-level pure CUDA C++ implementation is the most efficient and flexible. The high-level single-GPU ArrayFire CUDA implementation provides lower development time but is more than 4 times slower. It also can be easily generalized for executing on Beowulf clusters using the MPI, though, for the CPU-dominated clusters, oneAPI DPC++ is a better choice. As a drawback, currently, oneAPI's support for CUDA has significantly worse performance than even CPU-only code using DPC++. PyTorch could be recommended for prototyping and validating high-level solutions and can also be used quite well when short development cycles cannot allow for a more optimized approach.

The code used for this research is available at GitHub [18].

References

[1] "CPU DB clock frequency history," 2021. [Online]. Available: http://cpudb.stanford.edu/visualize/clock_frequency
[2] P. Bochev, M. Gunzburger, and R. Lehoucq, "On stabilized finite element methods for the stokes problem in the small time step limit," *International Journal for Numerical Methods in Fluids*, vol. 53, pp. 573 – 597, 02 2007.
[3] V. Fedynyak, O. Hryniv, O. Sobkovych, B. Vey, and O. Farenyuk, "Productivity comparison of the popular parallel programming approaches on computational fluid dynamics problem," in *2021 IEEE 16th International Conference on Computer Sciences and Information Technologies (CSIT)*, vol. 1, 2021, pp. 1–4.
[4] "MPI solutions for Nvidia GPUs," 2023. [Online]. Available: https://developer.nvidia.com/mpi-solutions-gpus
[5] "Running CUDA-aware OpenMPI," 2019. [Online]. Available: https://www.open-mpi.org/faq/?category=runcuda
[6] "OpenMP," 2022. [Online]. Available: https://www.openmp.org/
[7] *SYCL*, 2020. [Online]. Available: https://www.khronos.org/sycl/
[8] *oneAPI*, 2023. [Online]. Available: http://bit.ly/UCUoneAPI
[9] A. Gorobets, F. X. Trias, and A. Oliva, "An OpenCL-based parallel CFD code for simulations on hybrid systems with massively-parallel accelerators," *Procedia Engineering*, vol. 61, pp. 81–86, 2013.
[10] "GPUDirect," 2022. [Online]. Available: https://developer.nvidia.com/gpudirect
[11] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. Tallent, and K. Barker, "Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect," 2019.
[12] M. Knap and P. Czarnul, "Performance evaluation of unified memory with prefetching and oversubscription for selected parallel cuda applications on NVIDIA Pascal and Volta GPUs," *Journal of supercomputing*, vol. 75, pp. 7625–7645, 2019.
[13] *ArrayFire*, 2021. [Online]. Available: https://arrayfire.com/
[14] *PyTorch*, 2017. [Online]. Available: https://pytorch.org/
[15] J. P. Shaffer, "Multiple hypothesis testing," *Annual Review of Psychology*, vol. 46, no. 1, pp. 561–584, 1995.
[16] A. Sultanov, M. Protsyk, M. Kuzyshyn, D. Omelkina, V. Shevchuk, and O. Farenyuk, "A statistics-based performance testing methodology: a case study for the I/O bound tasks," in *2022 IEEE 17th International Conference on Computer Sciences and Information Technologies (CSIT)*, 2022, pp. 486–489.
[17] D. Rey and M. Neuhäuser, *Wilcoxon-Signed-Rank Test*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1658–1659.
[18] "Code for this research," 2023. [Online]. Available: https://github.com/aks-research-team/HPC-research