EUMaster4HPC Challenge
Parallelizing the Conjugate Gradient Method

Daniele Ferrario
Politecnico di Milano
daniele6.ferrario@mail.polimi.it

Luca Guffanti
Politecnico di Milano
luca2.guffanti@mail.polimi.it

Ekkehard Steinmacher
Università della Svizzera italiana
ekkehard.steinmacher@usi.ch

Abstract—**This report describes various parallelization approaches employed to accelerate the execution of the conjugate gradient algorithm. CPU-, GPU- and FPGA-based techniques are explored for different matrix sizes and benchmarked on the *MeluXina* supercomputer. The serial code takes $20.673\,\mathrm{s}$ to solve a linear system with a 10000×10000 matrix. The OpenMP version shows better performance by solving the same problem in $8.48\,\mathrm{s}$ using 8 threads on 1 process and thus achieving a $2.44\times$ speedup. The OpenMP+OpenBLAS version takes $7.306\,\mathrm{s}$ using 6 threads on 1 process. The MPI version solves the same task in $1.35\,\mathrm{s}$ by using 256 processes, hence achieving a factor 15 speedup compared with the serial code. A linear system with a 70000×70000 matrix, which takes $483\,\mathrm{s}$ to solve sequentially with a tolerance of a part per thousand, is solved in $1.620\,\mathrm{s}$ using 16 GPUs that, reaching a speed of $2.96\,\mathrm{TFLOPs}$, yield an approximately $300\times$ speedup. The MPI parallel version is compute bound, while the GPU implementations are memory bound. It is also shown that I/O operations can be parallelized and thus are up to $2\times$ and $7\times$ faster than the serial I/O operations.**

The code is available in the github repository of the project: https://github.com/Ferraah/ParallelCG.

I. Introduction

The conjugate gradient method is an iterative solver for linear systems of equations. Being able to solve large-scale linear systems is a necessity to tackle a multitude of problems in science, technology, finance and many other fields. Ad-hoc implementations of high performance solvers are vital, as they drive important advancements in those fields. Furthermore, the employed techniques are not problem-specific and the *forma mentis* can be applied to the parallelization of many other computational problems.

The parallelization techniques are benchmarked on the *MeluXina* supercomputer. Its cluster module with CPU nodes, as well as both GPU or FPGA accelerator nodes, are used. The CPU cluster is composed of 573 nodes, each node featuring 2 AMD Rome CPUs, with 64 cores each at a clock rate of $2.6\,\mathrm{GHz}$, for a total of 128 cores (256 HT cores) and 512 GB of RAM [1].

There are 200 GPU nodes in *MeluXina's* architecture. Each GPU node has 2 AMD Rome CPUs and 4 NVIDIA A100-43 GPUs. Each CPU has 32 cores at a clock rate of $2.35\,\mathrm{GHz}$, which corresponds to 128 HT cores in total. Each GPU node has 512 GB of RAM and 1.92 TB of local SSD [1].

There are 20 FPGA nodes. Each FPGA node has 2 AMD Rome and 2 Intel Stratix 10MX 16 GB FPGA. CPUs have 32 cores each at a clock rate of $2.35\,\mathrm{GHz}$, which corresponds to a total of 128 HT cores. Again, each node has 512 GB of RAM and 1.92 GB of local SSD [1].

Parallelization techniques for each hardware type are explored, benchmarked and compared in the following sections.

II. Mathematical Foundation

The conjugate gradient method is an iterative solver for a system of linear equations. Consider the system

$$\mathbf{A}x = b\,, \tag{1}$$

where $\mathbf{A} \in \mathbb{R}^{n\times n}, n \in \mathbb{N}$ is a symmetric positive definite matrix, and $x \in \mathbb{R}^n$ and $b \in \mathbb{R}^n$ are vectors. $\mathbf{A}$ and b are known, and the system is solved for x.

First an initial guess x_0 is chosen. With this guess the residual $r_0 := b - \mathbf{A}x_0$ is calculated. The initial direction is $d_0 := r_0$. Each step k involves multiple sub-steps. These are repeated until convergence, for which a residual-based criterion has been chosen: $||r_{k+1}||_2 < \epsilon \in \mathbb{R}$.

- Compute step size: $\alpha_k := \frac{r_k^T r_k}{d_k^T \mathbf{A} d_k}$
- Update guess: $x_{k+1} := x_k + \alpha_k d_k$
- Update residual: $r_{k+1} := r_k - \alpha_k \mathbf{A} d_k$
- Compute scalar: $\beta_k := \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
- Update search direction: $d_{k+1} := r_{k+1} + \beta_k d_k$
- Update step number: $k := k + 1$

x_{k+1} is a good approximation of the true solution $\tilde{x}$ [2].

III. Overview of the used APIs

A. MPI

MPI, or Message Passing Interface, is a standardized and portable message-passing system designed for parallel computing. It allows multiple processes or tasks to communicate with each other in a **distributed memory environment**. The memory management is explicit, so every process in an MPI program has its own memory. Hence, communication between processes is required to share data.

B. OpenMP

OpenMP, or Open Multi-Processing, is a widely used application programming interface for parallel programming in **shared memory environments**. It allows developers to parallelize the code by adding directives to indicate parallel regions, making it suitable for multicore processors and symmetric multiprocessing (SMP) systems.

C. *CUDA*

CUDA, or **C**ompute **U**nified **D**evice **A**rchitecture, is a parallel computing platform and programming model developed by NVIDIA. It allows developers to employ NVIDIA GPUs to efficiently accelerate large data-parallel workloads, thanks to the throughput oriented underlying architecture. CUDA can be coupled with *cuBLAS* (**C**UDA **B**asics **L**inear **A**lgebra **S**ubroutine): an industry-level GPU-accelerated library designed by NVIDIA and oriented towards HPC and AI applications. Backing CUDA and cuBLAS by MPI or NCCL (**N**VIDIA **C**ollective **C**ommunication **L**ibrary) for intra- and inter-node multi-GPU orchestration of tasks can lead to high performance for extremely large datasets.

D. *OpenCL*

OpenCL, or Open Computing Language, is an open standard framework for heterogeneous computing developed by the Khronos Group. It provides a programming interface for software developers to write applications that can be executed on a variety of hardware: CPUs, GPUs, and FPGAs.

IV. Optimization Techniques & Results

The runtimes presented in this chapter exclusively reflect the durations taken by the programs to execute the CG algorithm, excluding IO. The memory traffic times were only considered in subsection IV-D.

A. *CPU based techniques*

The conjugate gradient algorithm, as described in section II can be translated into pseudo code. This pseudo code can then be expanded by the emphasized parallelizing functionalities.

The most computationally intensive part of the algorithm is the matrix-vector multiplication that, for all parallel implementations, is either sped up by introducing the OpenMP clause `omp parallel for` or by splitting the computation among MPI ranks. In the MPI version, `Allgatherv` is used to distribute the matrix-vector multiplication result to all the processes.

The code can be further improved by using the OpenBLAS library. `cblas_ddot` is used for dot products; the functions `cblas_dscal` and `cblas_daxpy` scale a vector and perform a scaled vector plus a vector addition; `cblas_dgemv` performs matrix-vector multiplications.

1) OpenMP: Figure 1 shows a the strong scaling behavior of the OpenMP version of the conjugate gradient algorithm. The problem size is $n = 10000$ and $\epsilon = 10^{-9}$.

With OpenMP, strong scaling is apparent up to 4 threads; then, the scaling flattens off, as also evidenced by the sharp decline in efficiency. A lower bound is hit, which might be due to false sharing within the shared memory architecture or mishandling the cc-NUMA architecture [3]. In addition, OpenMP's rising thread overhead might also decrease the performance significantly for higher number of threads.

Nonetheless, the OpenMP version at its best takes 8.48 s to solve a linear system with a 10000×10000 matrix. Hence, it is $2.44\times$ faster than the serial execution.

Algorithm 1 Parallelized Conjugate Gradient Algorithm

```
1:  procedure PARALLEL-CG($\mathbf{A}, b, x_0, \epsilon$, max_iters)
2:      $\mathbf{A}$ is divided among MPI ranks
3:      Each rank has $\mathbf{A_p}$ and $b$
4:      Initialize $x_0$
5:      Set residual: $r_0 \leftarrow b$
6:      Set residual product: $rr_0 \leftarrow b^T b$ // Dot products use omp parallel for reduction
7:      Set the initial search direction: $d_0 \leftarrow b$
8:      $k \leftarrow 0$
9:      while $\|r_k\| > \epsilon$ and $k <$ max_iters do
10:         Compute mat-vec product: $A_p d_k \leftarrow \mathbf{A_p} d_k$ // Allgatherv of all $A_p d_k$ to full $Ad_k$ // Can also combine with omp parallel for
11:         Compute step size: $\alpha_k \leftarrow \frac{rr_k}{d_k^T A d_k}$
12:         Update guess: $x_{k+1} \leftarrow x_k + \alpha_k d_k$ // Use omp parallel for for vector additions
13:         Update residual: $r_{k+1} \leftarrow r_k - \alpha_k A d_k$
14:         Compute residual product: $rr_{k+1} = r_{k+1}^T r_{k+1}$
15:         Compute scalar: $\beta_k \leftarrow \frac{rr_{k+1}}{rr_k}$
16:         Update search direction: $d_{k+1} \leftarrow r_{k+1} + \beta_k d_k$
17:         $k \leftarrow k + 1$
18:     end while
19:     return $x_k$
20: end procedure
```

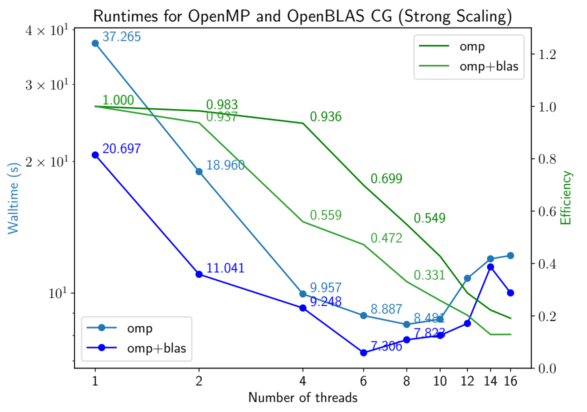

Fig. 1. Strong scaling behavior of the OpenMP and the OpenMP+OpenBLAS parallel versions of the conjugate gradient algorithm with $n = 10000$.

Efficiency is calculated by setting the runtime achieved by 1 thread as reference, and multiplying the ratio between the reference and other runtimes by the number of active threads.

2) MPI: Figure 2 shows the the execution time of the MPI parallel program with 256 processes on 1 node as well as the sequential program depending on the problem size n. The data points where fitted by the parabola $f(n) = a \cdot n^2$, to see the scaling behavior. The theoretical scaling behavior of the conjugate gradient algorithm is given by $f(n)$, since the complexity of the algorithm is $\mathcal{O}(n^2)$. Both fit functions verify this behavior, with inaccuracies for small problem sizes due to overhead. The smaller scaling coefficient a of the parallel algorithm, when compared to scaling coefficient of the sequential program, emphasizes the gained speedup. The average speedup $\bar{s}$ is also reflected in their ratio, which is $\bar{s} = \frac{a_{\text{seq}}}{a_{\text{par}}} \approx 19.64$. Finally, the MPI parallel program is on average $19.64\times$ faster than the sequential program independently of the matrix size.

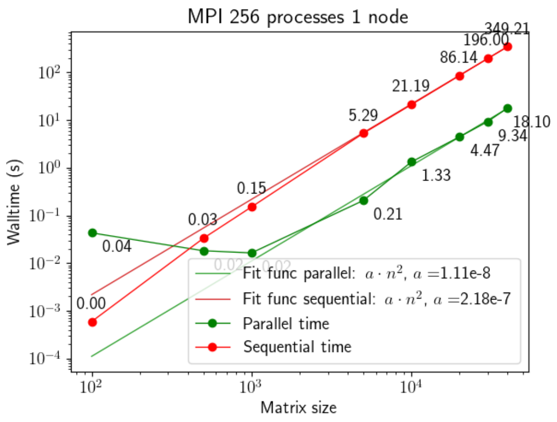

Fig. 2. Execution time as a function of the problem size of the MPI (256 processes) and sequential codes on one MeluXina CPU node.

Figure 3 shows the strong scaling behavior of the MPI version of the conjugate gradient algorithm. The problem size is $n = 10000$.

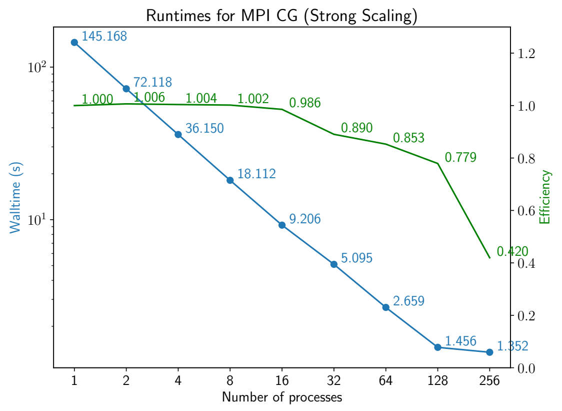

Fig. 3. Strong scaling behavior of the MPI parallel version of the conjugate gradient algorithm with $n = 10000$.

Strong scaling can be observed up to 128 processes. Whenever the number of processes doubles the execution time roughly halves. This matches the theoretical maximal achievable scaling behavior, as also underlined by the efficiency. For 256 processes the program does not scale that well, thus the decrease in efficiency.

Figure 4 shows the weak scaling behavior of the MPI version of the conjugate gradient algorithm.

Theoretically the execution time should be constant, since the workload per process is constant. For an increasing number of processes a slight increase in time is observed, which might be due to the growing communication workload. Nonetheless, the overall weak scaling behavior of the MPI version is good and can also be observed by the only slightly declining efficiency.

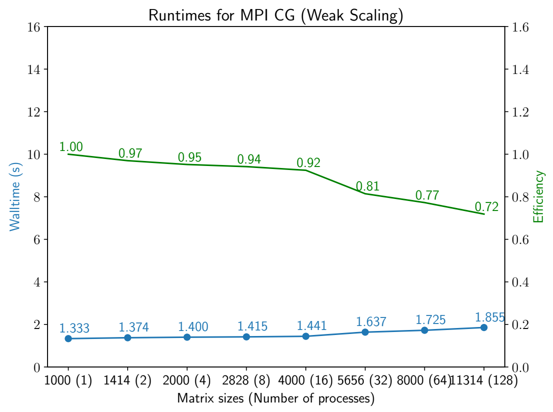

Fig. 4. Weak scaling behavior of the MPI parallel version of the conjugate gradient algorithm.

The MPI program can solve a linear system with $n = 10000$ in $1.33\,\text{s}$ on 256 processes, which is roughly 15 times faster than the serial execution. For an arbitrary problem size the MPI parallel program is $s = 19.64$ times faster than the sequential counterpart.

The MPI parallel program is compute bound, as determined by the roofline model. The conjugate gradient algorithm takes roughly $i \approx 360$ iterations to converge when the matrix size is $n = 10000$ and each element is a double (=2 floats). For $p = 128$ processes, here also equal to the number of cores, the MPI parallel program converges in roughly $t \approx 1.5\,\text{s}$. Hence, the work W, denoting the number of floating point operations performed within a second for this program is

$$W = \frac{i \cdot n^2}{t} = \frac{360 \cdot 20000^2 \text{FLOPs}}{1.5\,\text{s}} = 96\,\text{GFLOPs/s}\,.$$

The computational complexity of a single iteration of the conjugate gradient is $\mathcal{O}(n^2)$, since the matrix-vector multiplication dominates. Hence the n^2 term in the formula.

During a single iteration of the program each process transfers a vector of size n containing doubles. So, the number of bytes of data transfer, denoted by Q, during the full execution is

$$Q = \frac{p \cdot n \cdot i}{t} = \frac{128 \cdot 20\,000\,\text{floats} \cdot 4\,\text{bytes/float} \cdot 360}{1.5\,\text{s}} \approx 2.46\,\text{Gbyte/s}\,.$$

Thus the operational intensity I of the MPI parallel program is

$$I = \frac{W}{Q} \approx 39.0\,\text{FLOPs/byte}\,.$$

The best theoretical achievable performance $\tilde{W}$ with 128 cores for the *MeluXina* CPU cluster (AMD Rome CPUs @ $f = 2.6\,\text{GHz}$) with computing only one FLOP per cycle is

$$\tilde{W} = p \cdot f \cdot 1\,\text{FLOP} = 128 \cdot 2.6\,\text{G/s} \cdot 1\,\text{FLOP} = 332.8\,\text{GFLOPs/s}\,.$$

All processes are connected via HDR200 SP interconnects which have a bandwidth B of $B = 25\,\mathrm{Gbyte/s}$. Therefore the ridge point P of the roofline model for this machine is

$$P = \frac{\tilde{W}}{B} = \frac{332.8\,\mathrm{GFLOPs/s}}{25\,\mathrm{Gbyte/s}} \approx 13.3\,\mathrm{FLOPs/byte}\,.$$

The operational intensity of this program is greater than the ridge point, meaning $I > P$, therefore the program is compute bound.

3) OpenBLAS: Including the optimized mathematical operations of the OpenBLAS library on top of the OpenMP parallel program reduces the runtime. This is also showcased in Figure 1. The runtime of the OpenMP+OpenBlas parallel program is less than of just the OpenMP parallel program. However the efficiency of the OpenMP+OpenBlas parallel program is worse. Probably this is due to the fact, that for one thread the included BLAS functions have the most leverage, because for one thread they execute larger data sizes compared to multi-thread parallel executions.

The best achieved performance with this program is a runtime of $7.306\,\mathrm{s}$ with 6 threads for $n = 10000$. The thread overhead dominates with a rising number of threads and therefore faster runtimes were not achieved.

4) OpenCL: The OpenCL implementation of the algorithm is the same for both GPU and CPU, thanks to the higher level of abstraction the API provides. Parallelism is achieved through threading, and work-item corresponds to a separate thread that can be executed in parallel on the available CPU cores. The degree of parallelism and the mapping of work-items to CPU cores depends on the specific OpenCL runtime implementation and the hardware characteristics. The kernels file *kernels.cl* is compiled into binary for the CPU and saved as an external file by the host code for next usages.

Figure 5 shows the the execution time of the OpenCL kernels program as well as the sequential program run on a single CPU depending on the problem size n. The average speedup achieved is $\bar{s} \approx 7.17$.

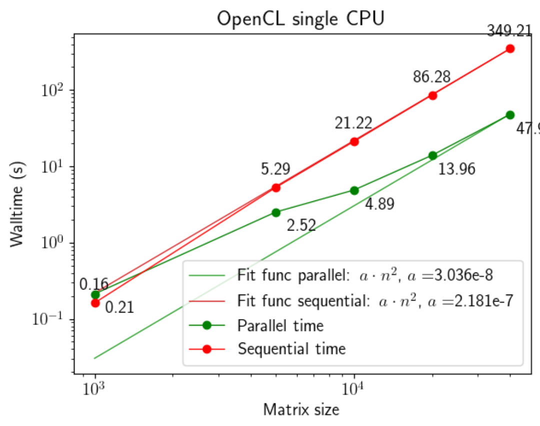

Fig. 5. Execution time as a function of the problem size of the OpenCL kernels and the sequential code on a single CPU.

B. GPU based techniques

The GPU implementations of the conjugate gradient method aim to offload as much work as possible to the available GPUs. *MeluXina* features a high number of GPU nodes, thus two implementations are explored: an initial single GPU implementation and a distributed, multi-GPU implementation. Independent of the implementation, GPUs are capable of yielding notable speedups even though performances are heavily limited by cumbersome data-transfers to and from the device. These transfers could be avoided by directly generating matrices and vectors in the device. This optimization, whilst possible, does not pertain to the scope of the project and is unfeasible considering the provided inputs. Nonetheless, keeping the context of the algorithm stored in the device memory and avoiding device oversubscription are adopted optimizations that contribute to high performances.

1) Single-GPU implementation: In the single-GPU case, all data structures needed to execute the algorithm are stored in the GPU. The main processor, after copying data to the device, commands the GPU execution of compute-intensive tasks via the invocation of cuBLAS functions. Aside from the first transfer of data to the GPU which loads the matrix and vectors, the most memory-intensive operation is the final copy of the computed solution in the host memory. Beneficially however, this is executed only once.

The single-GPU acceleration is about two orders of magnitude faster than the serial implementation as shown in Figure 6.

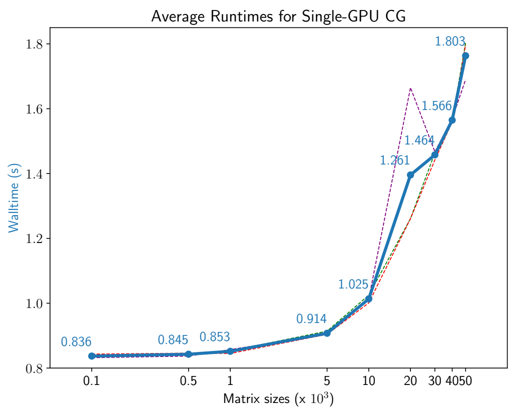

Fig. 6. Average runtimes for single-GPU CG with CUDA.

Similarly to the CUDA implementation, OpenCL kernels are run by the host (CPU) on the device (GPU), letting the OpenCL API automatically choose the best local work-group size according to the target architecture. The matrix and known vector are loaded once into the GPU memory, and results of the dot-product and matrix-vector multiplication are not transferred to host memory until the end. The kernels file *kernels.cl* is compiled into assembly code for the GPU and saved as an external file by the host code for next usages.

Figure 7 shows the the execution time of the OpenCL kernels program on a single GPU as well as the sequential program run on a CPU depending on the problem size n. The average speedup achieved is $\bar{s} \approx 44.89$.

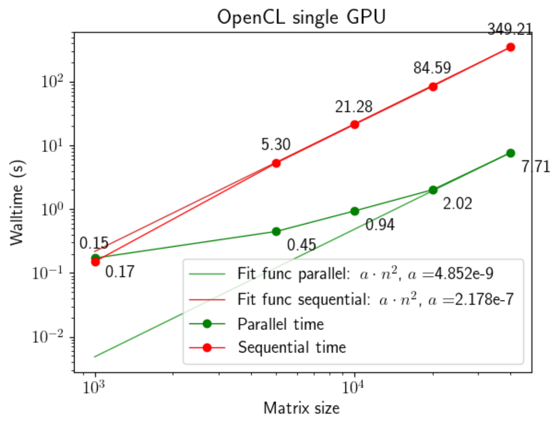

Fig. 7. Execution time as a function of the problem size of the OpenCL kernels on a single GPU and the sequential code on a CPU.

2) Multi-GPU implementation: The workload is distributed between GPUs in the accelerator nodes and implemented with CUDA, cuBLAS and MPI, enabling the access to larger problems than the single-GPU approach. MPI processes, in number equal to the number of available GPUs, are spawned and each process is mapped to a single GPU. Each device is associated to a set of rows which is determined at runtime and the matrix is read from file with MPI parallel I/O functions. Vectors (b is read in parallel) are completely stored in each GPU as they are not as large as the matrix. The matrix is distributed as evenly as possible.

All the processes execute the same algorithm: after commanding the allocation of the necessary resources they act as dispatchers of MPI operations on the host and of cuBLAS functions on the device. Vector sums and dot products, which span entire vectors, are simply invoked while the matrix multiplication, which is distributed, proceeds as follows: each process calls the matrix-vector multiplication on a subset of the matrix which updates only a subset of the elements of the vector, without overlaps. Then, after the partial result has been copied to the host memory, the `MPI_AllGatherv` collective operation is invoked to distribute the updated elements to all other process and memory is again moved back to the GPU. Last but not least, MPI synchronization constructs are put in place throughout the code (for instance at the end to each iteration) to guarantee a uniform global advancement of the algorithm and a good orchestration of the the ranks.

Results indicate that the parallelization is effective. This approach is capable of delivering high speeds for relatively big workloads, as shown in Figure 8. The slope indicates that scaling is approximately linear, which conforms to the theory. The scaling is not perfect due to various overheads introduced by the use of MPI and various memory transfers from and to the devices. Additionally, the parallelization shows a good weak scaling. This result is especially useful as it indicates that increasing the size of the problem uniformly with the available processing power does not imply a notable increase in execution time, with fluctuations addressable to memory management and synchronization overheads.

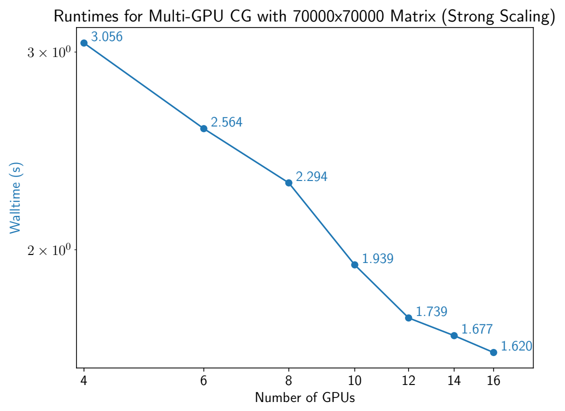

Fig. 8. Strong scaling behavior of the Multi-GPU parallel version of the conjugate gradient algorithm with $n = 60000$.

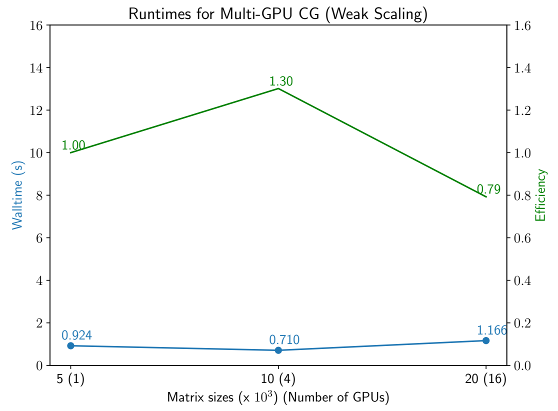

Fig. 9. Weak scaling behavior of the Multi-GPU parallel version of the conjugate gradient algorithm.

C. FPGA's

FPGAs (Field-Programmable Gate Arrays) are commonly used as accelerators in various computing applications, since they are inherently parallel devices if exploited correctly through the use of **pipelining**. They act as a solution between GPUs, which are still general purpouse devices, and the far more expensive application-specific integrated circuits. They consist of an array of programmable logic blocks and interconnects that can be configured through Hardware Description Languages (e.g. VHDL & Verilog) or higher level APIs such as OpenCL.

To implement the CG algorithm on MeluXina Intel FPGAs, OpenCL has been considered to be the best choice given its focus on heterogeneous computing. Using OpenCL, it's possible to run code on CPUs, GPUs and FPGAs with little to no changes to the kernels.

Kernels that compute the **dot product** and the **matrix-vector product**. Local/private memory exploitation, along with a careful and explicit management of the device memory, have substantially limited communications overhead.

Unfortunately, the FPGA implementation has not been tested on the real FPGA due to the long compilation times of the kernels (several hours!) and problems related to the space occupied by the synthesized components on the real device circuit.

D. IO-optimization

IO operations, especially in the case of large data-sets (i.e. reading of input matrices) significantly slow down the execution. To eliminate this bottleneck, `MPI_IO` primitives were used to operate reads from files in parallel without additional coordination of the active processes. Dynamic distribution of the workload, based on the available parallelism, proved to be essential when reading sizeable amounts of data.

Benchmarks performed on an increasing number of MPI processes, show that the use of `MPI_IO` functions effectively reduces the time of IO operations. This is displayed in Figure 10. They also indicate how synchronization overheads, especially for a high number of active processes, still slow down the reading. A high number of ranks proves to not be the best choice for IO workloads when the problem in not considerably large.

Increasing the number of MPI processes enhances solver performance. However, concurrently, the growth in MPI processes leads to escalated IO overhead, significantly impacting the overall execution time as shown in Figure 11.

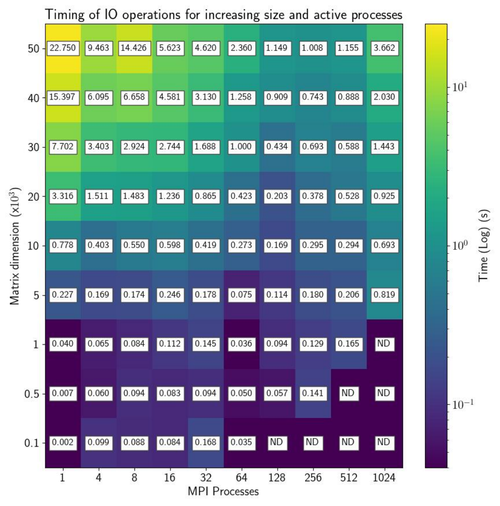

Fig. 10. Benchmark of the IO operations with increasing problem size and processes.

V. Summary and Final Comparison

This report describes various parallelization approaches employed to accelerate the execution of the conjugate gradient algorithm. CPU-, GPU- and (FPGA)-based techniques

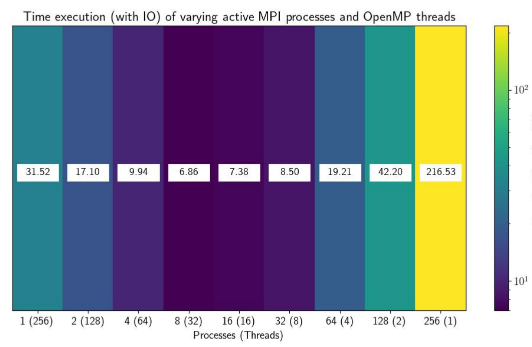

Fig. 11. Benchmark of the MPI+OpenMP implementation with a 20000×20000 matrix, res $= 10^{-6}$, on one largemem node.

TABLE I
Runtime of the CG-algorithm and speedups for the explored parallelization techniques with $n = 20000$ and $\epsilon = 10^{-6}$.

Solver strategy	Solving time w.o. IO	Speedup
Sequential	86.14	1×
OMP	20.4503	4.21×
OpenCL CPU	13.95	6.17×
MPI-OMP 8P x 32 Th	2.83	30.48×
OpenCL Single GPU	2.02	42.65×
CUDA Single GPU	1.66	51.85×
MPI 256P	1.29	66.65×
CUDA-MPI 16 GPUs	1.228	70.14×
MPI Distributed IO 256P	0.95	90.67×

were explored for different matrix sizes and benchmarked on the *MeluXina* supercomputing cluster. Solving a linear system with a 10000×10000 matrix took $20.673\,\text{s}$ to solve sequentially, $8.48\,\text{s}$ using 8 threads on 1 process with the OpenMP version, $7.306\,\text{s}$ using 6 threads on 1 process with the OpenMP+OpenBLAS version and $1.35\,\text{s}$ by using 256 processes with the MPI parallel program. Hence the OpenMP program achieved a speedup of factor 2.44, while the OpenMP+OpenBLAS and MPI programs were 2.83 and 15.3 times faster. The average speedups $\bar{s}$, independent of the matrix size, of the programs were 19.64 for MPI, 7.17 for OpenCL with a single CPU and 44.89 for OpenCL with a single GPU. Hereby they were compared to the sequential program. Table I lists the runtimes of the techniques for $n = 20000$ and $\epsilon = 10^{-6}$. The MPI and Multi-GPU programs showed nice strong and weak scaling. The MPI program is compute bound, while the GPU programs are memory bound. The best overall speedup achieved is by the Multi-GPU program. It can solve a linear system with a 70000×70000 matrix, which takes $483\,\text{s}$ to solve sequentially, in $1.620\,\text{s}$ using 16 GPUs, thus resulting in about $300\times$ speedup. Hence it is also the best performing technique is with $2.96\,\text{TFLOPs/s}$. It was also shown that I/O operations can be parallelized and thus are up to $2\times$ and $7\times$ faster than the serial I/O operations.

Acknowledgments

The authors would like to thank Dr. Ezhilmathi Krishnasamy for his expert guidance and the *MeluXina* supercomputer team not only for their excellent support but also for the providence of the computing resources.

REFERENCES

[1] (2024, Mar.) Meluxina: System overview. [Online]. Available: https://docs.lxp.lu/system/overview/
[2] (2024, Mar.) Wikipedia: Conjugate gradient method. [Online]. Available: https://en.wikipedia.org/wiki/Conjugate_gradient_method
[3] (2024, Mar.) Openmp does not scale. [Online]. Available: https://www.youtube.com/watch?v=5-ZepxpwmUU