Toward a Multi-GPU Implementation of a GMRES Solver in CHAMPS

Anthony Bouchard*
Matthieu Parenteau*
Éric Laurendeau†
anthony.bouchard@polymtl.ca
matthieu.parenteau@polymtl.ca
eric.laurendeau@polymtl.ca
Polytechnique Montréal
Montréal, Québec, Canada

ABSTRACT

The Computational Fluid Dynamics (CFD) community has successfully leveraged GPUs for their solvers. In the industry, low-order solvers are often used because only engineering levels of accuracy are needed. Unlike high-order methods, these solvers don't have high ratios of floating point operations per memory fetches but can still make good use of GPUs because of the high number of elements computed and higher memory bandwidth of those types of hardware. These solvers often use solvers that were designed to be optimal for CPUs with sequential parts like the Symmetric Gauss Seidel (SGS) solver. In an attempt to adapt to the hardware architecture and to better utilize the computational power of the GPU, a Jacobian-free Newton-Krylov (JFNK) type of solver is envisioned. The JFNK solver makes use of the fact that only the effect of the Jacobian on a vector is needed, hence removing the need to store and inverse the Jacobian matrix. Instead, a finite-difference approximation is computed. This paper discusses the early implementation of such a solver by showing the performance on the GPU of a GMRES solver (with Jacobian) developed in CHAMPS, a 3D unstructured RANS solver written in Chapel. The performance is evaluated by presenting speedups and a strong scaling analysis of the method.

CCS CONCEPTS

• **Computing methodologies → Massively parallel algorithms**; *Distributed algorithms*; Parallel programming languages; • **Applied computing** → *Computer-aided design.*

KEYWORDS

GMRES, GPU, RANS, CHAPEL

*Ph.D. Candidate, Department of Mechanical Engineering
†Professor, Department of Mechanical Engineering

ACM Reference Format:
Anthony Bouchard, Matthieu Parenteau, and Éric Laurendeau. 2021. Toward a Multi-GPU Implementation of a GMRES Solver in CHAMPS. In *Chapel Implementers and Users Workshop 2021, June 04, 2021, Virtual format.* 8 pages.

1 INTRODUCTION

Many researchers have studied and successfully leveraged GPUs within CFD codes. These types of hardware are particularly attractive for high-order methods, which are well suited for GPUs because of their high ratios of floating point operations to memory fetches [2, 19–22]. However, low-order methods are often used in the industry, where only engineering level of accuracy is needed. Fortunately, GPUs can also be of interest for these methods due to the high number of grid points needed, allowing for massive parallelism. Moreover, the higher memory bandwidth of GPUs compared to CPUs can benefit these low-order solvers, that are often limited by the memory subsystem. In fact, many research projects were able to leverage GPUs with low-order CFD codes [4, 5, 7, 10, 12, 13]. However, production codes mostly make use of the most efficient solvers, which often consist of sequential algorithms such as Symmetric Gauss-Seidel (SGS), which are not well suited for GPUs, forcing the use of less efficient solvers on GPUs. In an attempt to find a parallel version of the SGS solver, Nguyen, Castonguay and Laurendeau [13] used a red-black pattern with Jacobi iterations and were able to reach speedups of 2.4x when comparing a 1 GPU (Tesla K20 with 208 Gb/s memory bandwidth) configuration with a 2 CPU (Xeon E5-2670 with 102.4 Gb/s combined memory bandwidth) configuration using their respective best solver, which is close to the memory bandwidth ratio of the hardware.

In an effort to leverage the computational power of GPUs even more, one can make use of a Jacobian-Free Newton-Krylov (JFNK) solver [3, 8] as the linear system solver where the Krylov solver is the Globalized Minimal Residual (GMRES) method [17]. GMRES is well known for its fast convergence when given a suitable initial solution. Among others, unsteady simulations can benefit from this kind of solver because such an initial solution is provided by the previous time step. Usually, the Jacobian matrix is approximated using a first-order approximation, which negatively impacts the convergence of the solver. However, the Jacobian-Free version of the GMRES solver exploits the fact that only the effect of the Jacobian on a vector is needed, thus preventing the need of explicitly assembling the true Jacobian. Instead, a Jacobian-vector product is calculated using a finite difference approximation, which requires another evaluation of the residuals that can be efficiently computed on GPUs. This product accurately sees the true Jacobian, enhancing the convergence properties compared to a regular GMRES solver. The use of a JFNK solver solved on GPU could potentially allow for unsteady RANS simulations, which are costly compared to RANS simulations, to become a standard in the industry.

Although the work is not fully completed for the JFNK solver, and hence the work in this paper does not provide much novelty to the community, the authors make a first step toward having a JFNK solver fully on GPUs by presenting early implementation of such solver. A regular GMRES solver (with Jacobian) is developed inside CHAMPS [15], a 3D unstructured RANS/URANS solver written in Chapel, and then completely ported over GPU using NVIDIA's parallel programming CUDA. The work in this paper will focus on assessing the performance of the GMRES solver on multi GPUs by comparing it to the CPU version.

2 CHAMPS

CHAMPS is a multi-physics CFD code developed by the research team of Professor Éric Laurendeau and is written in Chapel. Application of CFD codes to external aerodynamic problems, such as aircraft configurations, requires significant computing resources since the size of the computational grid is often very large and the equations highly non-linear. Consequently, high-performance computing over large distributed systems is critical. However, development of efficient algorithm over distributed memory is not particularly simple. The Chapel programming language offers an interesting alternative compared to the more traditional languages used in CFD with MPI/OpenMP (C, C++ and Fortran). With Chapel, native parallelism for both shared and distributed memory is provided using the Partitioned Global Address Space (PGAS) parallel model, which facilitates the implementation of parallel algorithms over distributed memory. In CHAMPS, the initial problem is partitioned into several zones and the coarse-grain parallelism is implemented in a way that each grid zone is assigned to a task using the *coforall* statement. Large problems are therefore easily distributed over several compute nodes, such as the one presented in Figure 1.

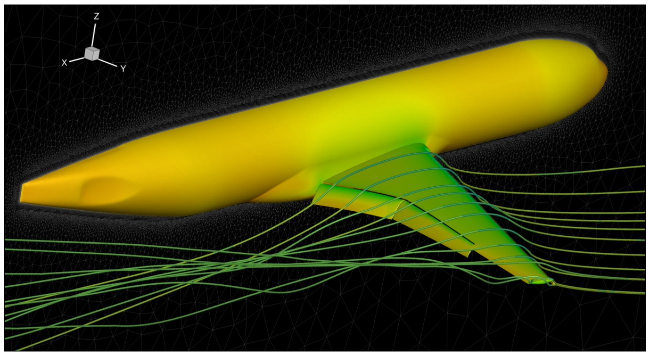


Figure 1: Computed flow solution with CHAMPS around a high-lift configuration aircraft.

Among other things, CHAMPS solves the RANS/URANS equations using an unstructured mesh-based second-order cell centered finite volume method. The governing equation can be expressed as

$$\Omega \frac{\partial \boldsymbol{W}_c}{\partial t} = -\boldsymbol{R}(\boldsymbol{W}_c) \tag{1}$$

Where Ω is the volume, $\boldsymbol{W}_c$ is the conservative variables and $\boldsymbol{R}(\boldsymbol{W}_c)$ is the residuals containing the convective and viscous fluxes. Multiple discretization schemes are implemented for the convective fluxes like the Roe [16], AUSM [9], JST [6], and others. The steady solution is reached solving iteratively using inexact Newton iterations as follows

$$\left[\frac{\Omega}{\Delta t} I + \frac{\partial \overline{\boldsymbol{R}}}{\partial \boldsymbol{W}_c}\right]^n \Delta \boldsymbol{W}_c^{n+1} = -\boldsymbol{R}(\boldsymbol{W}_c^n) \tag{2}$$

where $\frac{\partial \overline{\boldsymbol{R}}}{\partial \boldsymbol{W}_c}$ is a first-order approximation of the Jacobian matrix using the Thin Shear Layer (TSL) approximation [1] for the viscous fluxes. Note that the Jacobian matrix is a sparse matrix. To accelerate the convergence, local time-stepping with a Courant–Friedrichs–Lewy (CFL) number ramping up and residual smoothing are used.

Equation 2 is solved using a linear solver. In CHAMPS, the five stage hybrid Runge-Kutta and the Symmetric Gauss-Seidel (SGS) [1] schemes are implemented. Recently, the Generalized Minimal Residual (GMRES) method is also implemented both on CPU and GPU. This work focuses on the GMRES method and will be described more in detail in Section 3.

There are also multiple different turbulence models implemented in CHAMPS, the most commonly used being Sparlart-Allmaras [18] and $k\omega$ Menter SST [11]. The turbulence model is solved iteratively in a loosely coupled manner after every iteration of the flow.

3 GMRES SOLVER

GMRES is a Krylov-subspace solver, which is a general projection method that solves a linear system submitted to the Petrov-Galrkin condition

$$\boldsymbol{b} - A\boldsymbol{x}_m \perp \mathcal{L}_m \tag{3}$$

where $\mathcal{L}_m$ is a subspace of dimension m, A the Jacobian matrix and $\boldsymbol{x}_m$ an approximate solution $\in \mathcal{K}_m$, the Krylov subspace of dimension m

$$\mathcal{K}_m(A, \boldsymbol{r}_0) = span\{\boldsymbol{r}_0, A\boldsymbol{r}_0, A^2\boldsymbol{r}_0, ..., A^{m-1}\boldsymbol{r}_0\} \tag{4}$$

$\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$ and $\boldsymbol{x}_0$ is an initial guess. The initial guess is usually $\boldsymbol{x}_0 = 0$, giving

$$A^{-1}\boldsymbol{b} = \boldsymbol{x}_m \approx q_{m-1}(A)\boldsymbol{b} \tag{5}$$

where q_{m-1} is a polynomial of degree $m-1$ based on $\mathcal{K}_m$. GMRES, which is based on the Minimal-Residual (MINRES) method [14], uses $\mathcal{L}_m = A\mathcal{K}_m$. Algorithm 1 describes the basic algorithm of the GMRES method.

There exist many variations of this method. One variation that is particularly of interest is the restarted GMRES (also called GMRES(m)). As the number of Krylov iterations (m) increases, the memory and computational cost also increase. To remedy this situation, one can set a limit to the number of iterations and restart the GMRES with $x_0 = x_m$ once this limit is reached. This way, the memory and computational cost don't go too high. In CHAMPS, the restarted GMRES is implemented. However, for the sake of this article, the restarted GMRES is not used. In practical implementations, the orthogonal projection is usually performed using Arnoldi-Modified Gram-Schmidt method, which is represented in Algorithm 2. The

Algorithm 1: GMRES

```
$x_0 = 0$;
$v_1 = r_0/\|r_0\|_2$;
Define Hessenberg matrix $\bar{H}_m = \{h_{ij}\}_{1\le i\le m+1, 1\le j\le m}$;
$\bar{H}_m = 0$;
for $j = 1, 2, ..., m$ do
    $w_j = Av_j$;
    for $i = 1, ..., j$ do
        $h_{i,j} = (w_j, v_i)$;
        $w_j = w_j - h_{i,j}v_i$;
    end
    $h_{j+1,j} = \|w_j\|_2$;
    $v_{j+1} = w_{j+1}/h_{j+1,j}$;
end
$y_m = \text{argmin}_y \|\beta e_1 - \bar{H}_m y\|_2$;
$x_m = x_0 + V_m y_m$;
```

Hessenberg matrix, represented by $h_{i,j}$, is then formed and the least-square problem, $\|\beta e_1 - \bar{H}_m y\|_2$, is solved for y from which the final solution of x is retrieved. Solving the least-square problem is usually inexpensive since the number of Krylov iterations (m) is relatively small. Additionally, givens rotations are applied to the Hessenberg matrix to obtain a triangular matrix that can be solved efficiently. Most of the computational cost is incurred by the sparse Matrix-Vector multiplication ($w_j = Av_j$) and the Arnoldi process, which consist mainly of linear algebra operations. Note that the sparse matrix A is the Jacobian matrix of the flow as described previously. The time distribution of the GMRES algorithm is further discussed in Section 5.

The tolerance of the GMRES solver is another subject of interest. In our tests, we found that a tolerance between 10^{-1} and 10^{-2} is most efficient for converging a steady solution. On one hand, if the tolerance is too high, one can undersolve the linear system, resulting in the stagnation of the convergence or even divergence. On the other hand, if the tolerance is too low, one can find themselves in a situation where the linear system is oversolved, resulting in minimal gains on the convergence properties while greatly increasing the time of each iteration.

4 GPU IMPLEMENTATION

The GPU version of GMRES is programmed using NVIDIA's programming language CUDA. The original CPU algorithm in CHAMPS is unchanged and only the linear algebra operations are transferred to the GPU. These operations are rewritten in CUDA and linked as an external library to CHAMPS using Chapel's C-interoperability. The GPU operations are wrapped inside the sparse matrix and distributed vector objects of CHAMPS. Below, we see an example of the *norm* function defined inside the *DistributedVectorGPU_c* object in Chapel. The *cublas_Dot* function is a CUDA-c function (in this case, from the cuBLAS library).

```
class DistributedVectorGPU_c : Vector_c
{
    /** Number of local elements **/
    const niLocal_ : int;
    /** Local task id **/
    const localTaskId_ : int;
    /** Domain for the local values **/
    const localDomain_ : domain(1) = {0..#niLocal_};

    proc init(ni : int = 0, niLocal : int = 0,
 localTaskId : int = 0)
    {
        super.init(ni);

        niLocal_ = niLocal;
        localTaskId_ = localTaskId;
    }

    /**
        Computes the norm
    **/
    override proc norm(param comm : bool = true) : real_t
    {
        var res : real;
        local
        {
            if (updateGPU_)
            {
              this.GPUcopy(0);
            }

            res = cublas_Dot(cublasHandle_, niLocal_,
 d_values_, d_values_); // CUDA function
        }

        if (comm)
        {
            var val = globalReduction.reduceValues(
 localTaskId_, [res], REDUCE_OPERATION_t.SUM);
            res = val[0];
        }

        return sqrt(res);
    }
```

As a result, GPU matrix and vector objects are instantiated and the original CPU algorithm is used directly, avoiding unnecessary code duplication in CUDA. Thanks to Chapel's inheritance feature, the GPU functions are therefore seamlessly integrated alongside the CPU code as we can see with the small code portion below. that can execute both the CPU and GPU versions of GMRES.

```
/**
        Main procedure to solve the linear system

        :arg lhs: The LHS of the linear system to solve
        :arg rhs: The RHS of the linear system to solve
        :arg buildPreconditioner: If the LHS has changed,
     the preconditioner is updated
    **/
    proc solve(lhs : BSRmatrix_c, rhs : Vector_c,
 buildPreconditioner : bool = true) : int
    {
        var totalIteration : int = 0;

        var b_norm : real = rhs.norm();

        x_.reset();

        if (buildPreconditioner) then preconditioner_.
 buildPreconditioner();

        while (totalIteration < maxIterations_)
        {
            ref Q0 = krylovVectors_[0].Q_;
            Q0.set(rhs);

            if (totalIteration > 0)
            {
                x_.exchangeAuxValues(
    interfaceCommunicator_);
                lhs.dot(x_, workingArray_);
                Q0.addScale(workingArray_,-1.0);
            }
```

In GMRES, there are two types of operations: vector and matrix operations. For simple vector operations like the memset operation (ex. $x_0 = 0$ or $v_1 = r_0/\|r_0\|_2$ in Algorithm 1), fine-grain parallelism is implemented by associating one CUDA thread per dimension of the array. For more complex operations like the norm, the dot product and the add scale operation, the cuBLAS library is used. There is also a sparse Matrix-Vector multiplication for which the cuSPARSE library is used. Finally, all operations on the Hessenberg matrix are done on the CPU. The latter is a small matrix on which the givens rotation is applied sequentially. Since this operation represents less than 1% of the overall algorithm (see Figure 7), it is more efficient to use the CPU and avoid costly data transfer between the GPU and the CPU.

For the coarse-grain parallelism, each zone in the partitioned grid is assigned to a task by CHAMPS and distributed over the available *Locales*. Each of these tasks is assigned to an asynchronous CUDA stream, allowing for overlapping of the computations. However, the GPU allows only one process to make calls. Therefore, the CUDA Multi-Process Service (MPS) is required to enable concurrent calls between various threads on the same GPU. Although splitting the grid works well on GPU, this creates more interfaces, which increases the time spent for communication and synchronization between tasks. As a result, assigning one zone or one task per GPU is recommended to achieve maximum efficiency and avoid over-subscription of the GPU.

When using multiple GPUs, every process is evenly split between the available GPUs which is easily done with CUDA's library. In this work, the grid is partitioned into multiple zones and each zone is assigned to a GPU. The grid partitioning is accomplished using the external library METIS and a Reverse Cuthill-Mckee (RCM) reordering scheme is applied afterward to reduce the overall bandwidth of the sparse matrix. Consequently, the linear problem is distributed as illustrated in Figure 2 where the red values represent local blocks that interact only with values stored locally. Off-diagonal blocks, represented in blue, interacts with values belonging to a different task and possibly stored on a different *Locale*. It creates communication points for the sparse Matrix-Vector multiplication and within the orthogonalization process as shown in Algorithm 2.

Algorithm 2: Arnoldi-Modified Gram-Schmidt

```
$w_j = Av_j$ [communication point];
for $i = 1, ..., j$ do
    $h_{i,j} = (w_j, v_i)$ [communication point];
    $w_j = w_j - h_{i,j}v_i$;
end
$h_{j+1,j} = \|w_j\|_2$ [communication point];
$v_{j+1} = w_{j+1}/h_{j+1,j}$;
```

For the sparse Matrix-Vector operation, each task computes in parallel the operation on their attributed rows. The values of the vector corresponding to off-diagonal blocks are exchanged, see Figure 2, and every task computes the local part only of the resulting vector. For the Vector-Vector dot product, the dot product is computed on the local part and a reduction, involving communication, is performed afterward to sum the contribution of every task to obtain the final value. The communication involved in these operations are performed on the CPU with Chapel using the dedicated communication objects of CHAMPS. The overall process is illustrated in Figure 3. The dot operation is executed in parallel on the various GPUs on local data only. The resulting value for every GPU is copied to the CPU for the global reduction and the final value is returned to every task. Note that the same process is involved for the Matrix-Vector dot product with the exception that multiple values, corresponding to off-diagonal blocks, are copied from the GPU to the CPU to be exchanged and copied back to GPU to obtain the updated vector on the GPU. This process involves the use of dedicated buffer arrays to maximize efficiency when communicating values to the other task.

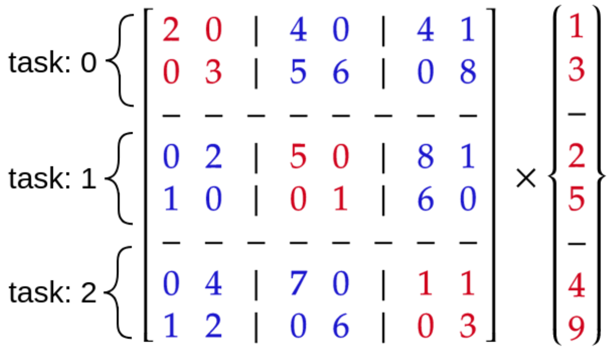


Figure 2: Illustration of a distributed matrix and vector for multiple GPUs (one task per GPU).

5 RESULTS

In this section, different test cases are presented to assess the performance of the GPU version of GMRES. In all those cases, a Cartesian grid with 2 million elements is used, see Figure 4, where the Euler equation is solved for the five unknowns (density, velocity and energy). Consequently, the linear system solved by GMRES has 10 million unknowns. Cases are run on Béluga which is a general-purpose cluster from Compute Canada. Table 1 shows a summary of the CPUs and GPUs on a node and the environment configuration of Chapel is described in Table 2. Chapel 1.23 is used for this work.

In the first test case, the GPU version of GMRES is run on 1, 2 and 4 GPUs for 1, 10, 25 and 50 Krylov iterations. Figure 5 shows the time per iteration of each run.

For this figure, we can see that when the number of Krylov iterations is increased, the time per iteration is also increasing. This is due to the Arnoldi method, where a Krylov vector is orthogonalized against all the previous vectors. It is therefore important to keep the number of Krylov iterations as low as possible to guarantee an efficient solver. To compare the CPU and GPU implementations, the CPU version is run on 2 CPUs (40 cores) for the same number of Krylov iterations. Speedups are shown in Figure 6.

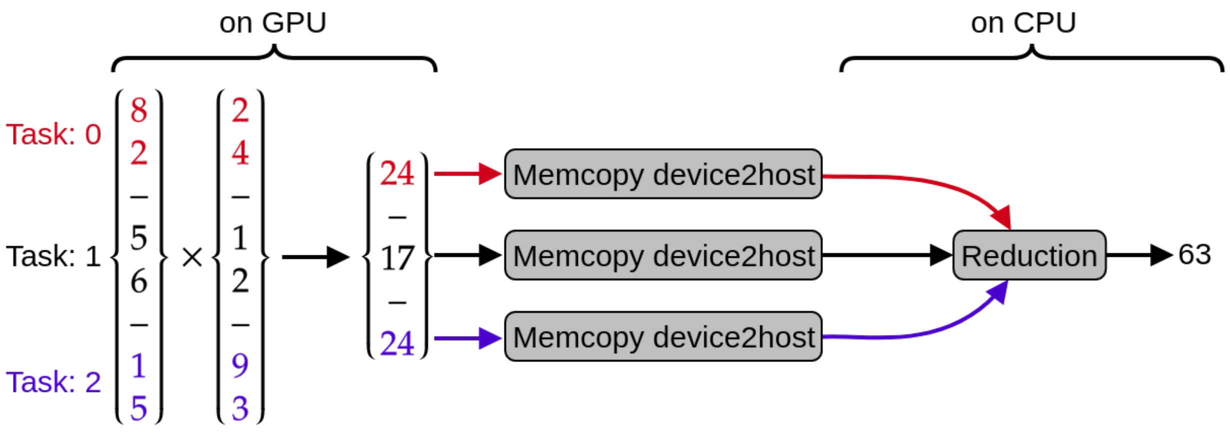


Figure 3: Illustration of a parallel dot product with multiple GPUs (one task per GPU).

Table 1: Summary of the CPUs and GPUs on one Beluga node.

	Intel Xeon Gold 6148 Skylake	**NVidia V100SXM2**
Number of CPUs/GPUs	2	4
CPU/GPU performance	2.4 GHz, 20 cores, 40 threads	1.53GHz, 80 SMs, 32 DP CUDA cores/SM
Peak performance per CPU/GPU	0.704 TFLOPS (DP)	7.8 TFLOPS (DP)
Peak memory bandwidth per CPU/GPU	128 GB/s	900 GB/s
Power per CPU/GPU	150W	300 W

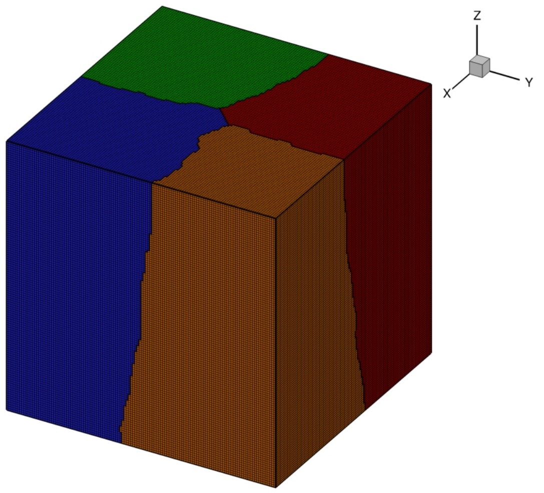


Figure 4: Cartesian grid with 2 million elements partitioned into four zones.

Table 2: Chapel 1.23 environment configuration.

Variable	**Value**
CHPL_COMM	gasnet
CHPL_COMM_SUBSTRATE	ibv
CHPL_GASNET_SEGMENT	large
CHPL_TASKS	qthreads
CHPL_LAUNCHER	gasnetrun_ibv

As stated a bit earlier, most of the functions are limited by the memory subsystem. Therefore, for that test case, we expect to have a speedup close to the ratio of the maximum memory bandwidth, which should be around 3.5 if we compare 1 GPU to 2 CPUs. For up to 50 Krylov iterations on one GPU, we have a speedup in line with the expected value. When increasing the number of GPUs, the speedup is around 6 and 9 for 2 and 4 GPUs respectively. It is normal to see a degradation in performance when running on more than one GPU, because the computation is slowed down by communication and synchronization between the GPUs. When the grid is partitioned, it creates more interfaces, which in turn increases the amount of data that is transferred between the different tasks. However, this represents a potential advantage for GPU implementations, since the number of GPU is usually much lower than the number of CPU, thereby reducing the number of partitions and interfaces.

The next test compares, in Figure 7, the distribution of the total time spent in the different functions of GMRES for 10 and 50 Krylov iterations. In this comparison, the Matrix-Vector multiplication is evaluated outside the Arnoldi part. Ten non-linear iterations are done for a total of 100 and 500 Krylov iterations respectively. This test is run on two GPUs and two CPUs.

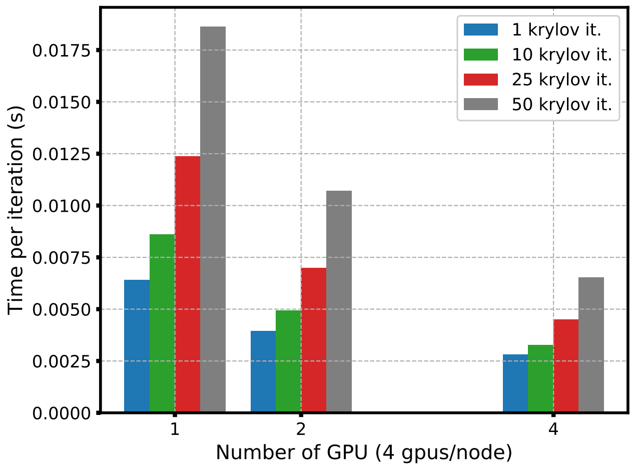


Figure 5: Real time per iteration in seconds for the GMRES algorithm on the GPU for different number of Krylov iterations.

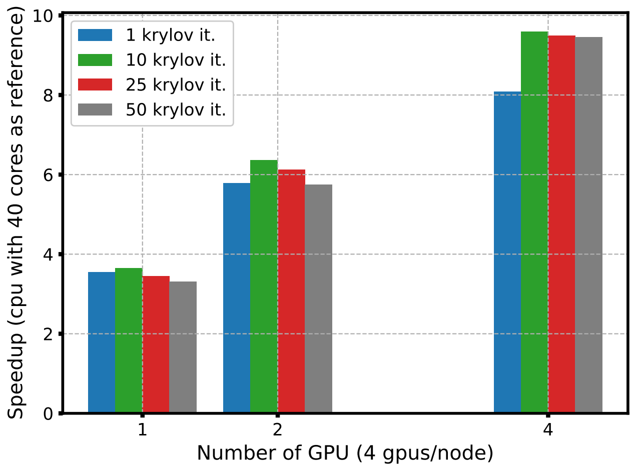
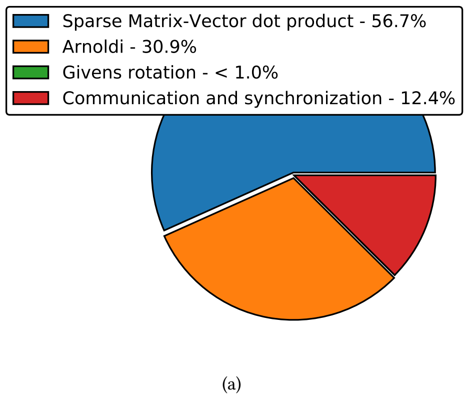


Figure 6: Speedup of the GPU version of GMRES compared to 1 CPU.

In the GMRES algorithm, the Matrix-Vector multiplication is only done one time per Krylov iteration. The Arnoldi orthogonalization is also done one time but loops over the previous Krylov vectors. Therefore, as the Krylov iteration is increased, the number of operations is also increased for the Arnoldi method. As a result, around 31% of the total time is consumed by Arnoldi for 10 Krylov iterations, while 64% is taken for 50 iterations. As for the communication, while it could be expected that the percentage increases with the number of Krylov iterations since there is a synchronization point in the Arnoldi orthogonalization, the most costly communication is in fact the one outside the Arnoldi orthogonalization for the Matrix-Vector operation. Indeed, the test case is run on a single node, making the global reductions inside the



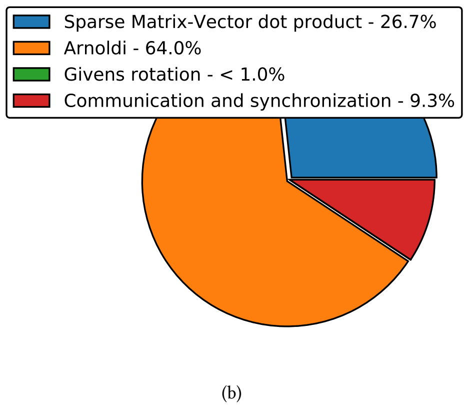


Figure 7: Comparison of the time distribution for (a) 10 Krylov iterations and (b) 50 Krylov iterations with 2 CPUs and GPUs

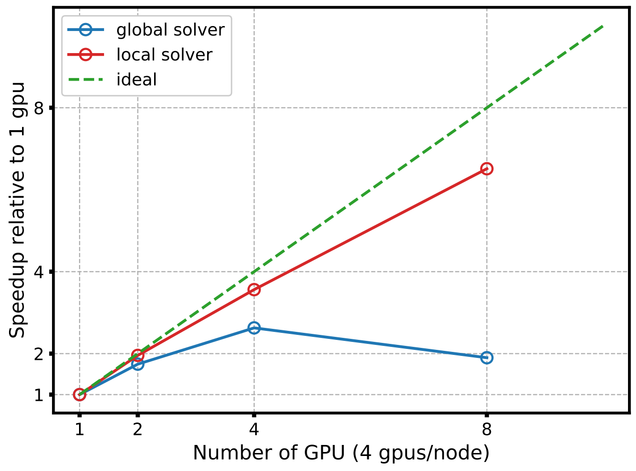
Arnoldi orthogonalization efficient when compared to the exchange of values needed for the Matrix-Vector multiplication.

The final test is a strong scalability analysis over 8 GPUs on 2 nodes. Figure 8 shows the speedup and efficiency when increasing the number of GPUs for the GPU version of GMRES with (global) and without (local) communication.

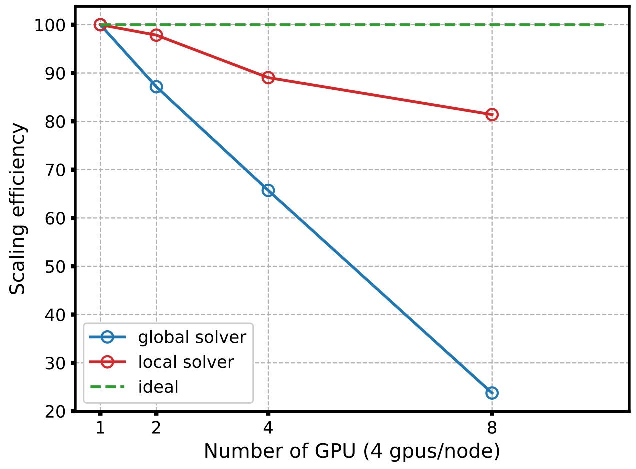
From these figures it is evident that there is a communication or synchronization problem between the *Locales*, which ruins the scalability of the algorithm. If all communication and synchronization points are removed, the GMRES solver maintains a relatively good scaling efficient above 80%. The communication and synchronization part of the algorithm represents roughly 65% of the total time when using two compute nodes, see Figure 9. This problem comes from the use of network atomics, which are working properly on Cray systems, but not on infiniband systems. Note that the same issue is also present with the CPU implementation.

6 CONCLUSION AND FUTURE WORK

In this article, a GPU version of GMRES is developed and compared with the CPU version already implemented in CHAMPS. The results show that the number of Krylov iterations increases the computational time of the algorithm exponentially. When compared to the



(a)



(b)

Figure 8: Strong scaling analysis of the GPU version of GMRES showing (a) the speedup and (b) the efficiency up to 8 GPUs.

CPU, speedups relatively close to the ratio of memory bandwidth are obtained with the GPU, which demonstrates that our implementation is efficient. It is also shown that when the number of Krylov iterations is higher, the relative time passed in the Arnoldi orthogonalization is also more important when compared to the other operations. Finally, the strong scaling analysis shows that a synchronization problem, due to the use of network atomics on infiniband systems, is preventing efficient scaling with more than one compute node.

For future work, the first step will be to investigate and fix the issue regarding the inter-nodes communication/synchronization. Then, like previously stated, the GMRES solver needs to be preconditioned. In CHAMPS, some preconditioners are already implemented on the CPU side. Block Jacobi and ILU(0) are good candidates to be ported over GPU. Finally, to have the JFNK solver fully on the GPU, there will be a need to port all functions related to the flow computation (fluxes, gradients, etc.).

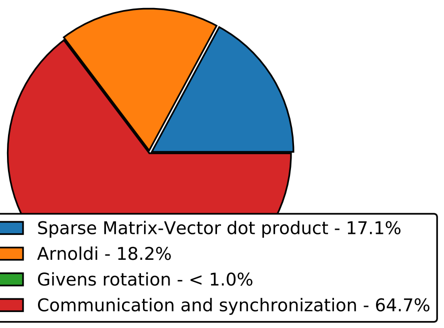


Figure 9: Time distribution with 8 GPUs distributed over two compute nodes for 10 Krylov iterations.

ACKNOWLEDGMENTS

This work is supported by the National Science and Engineering Research Council (NSERC) of Canada, the Consortium for Research and Innovition in Aerospace in Québec (CRIAQ), Bombardier Aerospace and Cray Canada. Computations were made on the supercomputer Béluga from École de Technologie Supérieur, managed by Calcul Québec and Compute Canada.

REFERENCES

[1] Jiri Blazek. 2005. *Computational Fluid Dynamics: Principles and Applications:(Book with accompanying CD).* Elsevier.
[2] Patrice Castonguay, David Williams, Peter Vincent, Manuel Lopez, and Antony Jameson. 2011. On the development of a high-order, multi-GPU enabled, compressible viscous flow solver for mixed unstructured grids. In *20th AIAA Computational Fluid Dynamics Conference.* 3229. https://doi.org/10.2514/6.2011-3229
[3] Todd T Chisholm and David W Zingg. 2009. A Jacobian-free Newton–Krylov algorithm for compressible turbulent fluid flows. *J. Comput. Phys.* 228, 9 (2009), 3490–3507. https://doi.org/10.1016/j.jcp.2009.02.004
[4] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, et al. 2011. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–12. https://doi.org/10.1145/2063384.2063396
[5] Dana Jacobsen, Julien Thibault, and Inanc Senocak. 2010. An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. In *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition.* 522. https://doi.org/10.2514/6.2010-522
[6] Antony Jameson, Wolfgang Schmidt, and Eli Turkel. 1981. Numerical solution of the Euler equations by finite volume methods using Runge Kutta time stepping schemes. In *14th fluid and plasma dynamics conference.* 1259. https://doi.org/10.2514/6.1981-1259
[7] IC Kampolis, XS Trompoukis, VG Asouti, and KC Giannakoglou. 2010. CFD-based analysis and two-level aerodynamic optimization on graphics processing units. *Computer Methods in Applied Mechanics and Engineering* 199, 9-12 (2010), 712–722. https://doi.org/10.1016/j.cma.2009.11.001
[8] Dana A Knoll and David E Keyes. 2004. Jacobian-free Newton–Krylov methods: a survey of approaches and applications. *J. Comput. Phys.* 193, 2 (2004), 357–397. https://doi.org/10.1016/j.jcp.2003.08.010
[9] Meng-Sing Liou and Christopher J Steffen Jr. 1993. A new flux splitting scheme. *Journal of Computational physics* 107, 1 (1993), 23–39. https://doi.org/10.1006/jcph.1993.1122
[10] Hongbin Liu, Xinrong Su, and Xin Yuan. 2018. Accelerating unstructured large eddy simulation solver with GPU. *Engineering Computations* (2018). https://doi.org/10.1108/EC-01-2018-0043

[11] Florian R Menter. 1994. Two-equation eddy-viscosity turbulence models for engineering applications. *AIAA journal* 32, 8 (1994), 1598–1605. https://doi.org/10.2514/3.12149

[12] Bahareh Mostafazadeh Davani, Ferran Marti, Behnam Pourghassemi, Feng Liu, and Aparna Chandramowlishwaran. 2017. Unsteady Navier-Stokes Computations on GPU Architectures. In *23rd AIAA Computational Fluid Dynamics Conference*. 4508. https://doi.org/10.2514/6.2017-4508

[13] MT Nguyen, P Castonguay, and E Laurendeau. 2018. GPU parallelization of multigrid RANS solver for three-dimensional aerodynamic simulations on multiblock grids. *The Journal of Supercomputing* (2018), 1–22. https://doi.org/10.1007/s11227-018-2653-6

[14] Christopher C Paige and Michael A Saunders. 1975. Solution of sparse indefinite systems of linear equations. *SIAM journal on numerical analysis* 12, 4 (1975), 617–629. https://doi.org/10.1137/0712047

[15] Matthieu Parenteau, Simon Bourgault-Cote, Frédéric Plante, Engin Kayraklioglu, and Eric Laurendeau. 2021. Development of Parallel CFD Applications with the Chapel Programming Language. In *AIAA Scitech 2021 Forum*. 0749. https://doi.org/10.2514/6.2021-0749

[16] P.L. Roe. 1981. Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes. *Journal of Computational Physics* 43 (1981), 357–372.

[17] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. Vol. 82. siam.

[18] PRaA Spalart and S1 Allmaras. 1992. A one-equation turbulence model for aerodynamic flows. In *30th aerospace sciences meeting and exhibit*. 439. https://doi.org/10.2514/6.1992-439

[19] Brian C Vermeire, Freddie D Witherden, and Peter E Vincent. 2017. On the utility of GPU accelerated high-order methods for unsteady flow simulations: A comparison with industry-standard tools. *J. Comput. Phys.* 334 (2017), 497–521. https://doi.org/10.1016/j.jcp.2016.12.049

[20] Peter Vincent, Freddie D Witherden, Antony M Farrington, George Ntemos, Brian C Vermeire, Jin S Park, and Arvind S Iyer. 2015. PyFR: next-generation high-order computational fluid dynamics on many-core hardware. In *22nd AIAA Computational Fluid Dynamics Conference*. 3050. https://doi.org/10.2514/6.2015-3050

[21] Jerry E Watkins, Joshua Romero, and Antony Jameson. 2016. Multi-GPU, implicit time stepping for high-order methods on unstructured grids. In *46th AIAA Fluid Dynamics Conference*. 3965. https://doi.org/10.2514/6.2016-3965

[22] Chuanfu Xu, Xiaogang Deng, Lilun Zhang, Yi Jiang, Wei Cao, Jianbin Fang, Yonggang Che, Yongxian Wang, and Wei Liu. 2013. Parallelizing a high-order CFD software for 3D, multi-block, structural grids on the TianHe-1A supercomputer. In *International Supercomputing Conference*. Springer, 26–39. https://doi.org/10.1007/978-3-642-38750-0_3