DOI: 10.14529/jsfi220405

Adapting a Scientific CFD Code to Industrial Applications on Hybrid Supercomputers

Andrey V. Gorobets[1]

The NOISEtte heterogeneous parallel code for simulating turbulent flow and aerodynamic noise is considered. In our previous works, high acceleration and parallel efficiency in scientific scale-resolving simulations using GPUs were reported. For parallelization, the MPI, OpenMP and OpenCL standards are used, the latter allows using GPUs from different vendors. However, the further transition to industrial-oriented applications brought more trouble. Instead of discussing the parallel algorithm, this work will focus on the problems that are not so obvious at first glance, which arise when developing a heterogeneous simulation code. How to deal with numerous simulation algorithm components, all those bells and whistles like wall functions, mixing plane and sliding interfaces, synthetic turbulence generators, a variety of boundary conditions, etc., that either need to be ported to the GPU side or incorporated directly from the CPU side? How to maintain and modify the OpenCL code in a growing number of source files? How to arrange the modularity of a complex heterogeneous software package? How to preserve reliability and fault tolerance, especially in the case of numerical schemes of increased accuracy, but reduced social responsibility? These issues are discussed here and some solutions will be proposed.

Keywords: heterogeneous code, computational fluid dynamics, turbulent flows, scale-resolving simulation, CPU+GPU, MPI+OpenMP+OpenCL.

Introduction

Computational fluid dynamics (CFD) software is one of the main burners of supercomputer time (as well as the associated electricity and taxpayers' money). Due to the high resource intensity, the efficient use of hybrid systems is critical, and GPU computing has long been widespread in CFD applications. CFD codes are constantly evolving in the use of hybrid supercomputers, improving scalability, enabling multi-GPU and heterogeneous computing capabilities. Many successful examples can be found in both scientific and commercial codes, see e.g. [1–3, 8] among many others, or specifications of commercial codes capable of multi-GPU computing, such as Simcenter STAR-CCM+, Ansys Fluent multi-GPU solver, GPU-optimized Altair software, including AcuSolve or its LBM-based flow solver ultraFluidX [6].

In the present work, a typical supercomputer time burner is considered, namely, the NOISEtte heterogeneous code for modeling turbulent flow and its aerodynamic noise. The parallel algorithm and performance on various GPU-based hybrid systems were already presented in detail in [4]. Here the focus will be not on the parallelization itself but on the problems that are not so obvious at first glance, which arise when developing a heterogeneous simulation code.

The article is organized as follows. Section 1 outlines numerical methods and algorithms. Section 2 is devoted to complexities of industrial applications. In Section 3, the modular architecture is presented. Section 4 is devoted to code reliability. Conclusions summarize the study.

1. Numerical and Parallel Framework

The turbulent flow of a viscous compressible gas is governed by the Navier–Stokes (NS) equations. The numerical algorithm is based on higher accuracy schemes on unstructured mixed-element meshes, hybrid RANS-LES approaches for turbulence modeling, implicit time integra-

[1]Keldysh Institute of Applied Mathematics, RAS Moscow, Russian Federation

tion. More information on our simulation technology and all its components can be found in [5] and references therein. For parallelization, the MPI, OpenMP and OpenCL standards are used, which allows us to cynically occupy an excessively large number of hybrid cluster nodes and to engage GPUs from different vendors. Two-level mesh partitioning is used for distribution of workload between cluster nodes and devices inside nodes. Overlapping computations and communications helps to achieve better parallel efficiency. Comprehensive information regarding the parallel algorithm is presented in [4]. The parallel performance demonstrated there includes CPU-only systems using up to about 10 thousand cores, GPU-based hybrid clusters using several dozen GPUs, obtaining the equivalent of about 150 to 200 CPU cores per GPU. The considered simple test cases were limited to an external flow on a static mesh.

2. Industrialization

When it comes to industrial problems, there are more components of the numerical methodology involved, for instance, wall functions, mixing plane and sliding interfaces, synthetic turbulence generators (STG) and sponge layers, a variety of boundary conditions (BC), deforming meshes, immersed boundary methods, more complex equations of state, among many others. Many of these components may seem insignificant as on CPUs they are responsible for a very minor fraction of the overall computing time, some even below 1%. Because of this "insignificance", there is a temptation to leave this functionality on the CPU side, since porting it to the GPU is no easier than the main parts of the simulation algorithm. But that does not work, since the GPU is an order of magnitude faster, hence the weight of the things left on the CPU becomes an order of magnitude bigger, respectively. The need to exchange extra data with the GPU enlarges this weight several times more. Thus, such small components, especially if there are many, can easily take up more than half of the computational time. It is easy to conclude that most of the effort has to be spent on such "insignificant" things in order to deal with industrial applications.

Porting so many things, in turn, imposes more problems related with maintainability and modifiability of so many OpenCL kernels. The OpenCL source code becomes organized in plenty of files. These files, when passed to the compiler at runtime of the CPU program, are concatenated with each other and with runtime-generated preprocessor definitions. This makes the compiler log hard to interpret, since the line numbers it reports are irrelevant. To solve this issue, the program module responsible for the OpenCL part tracks the number of lines in files and the assembly order of the source code, then it parses the compiler log and restores the actual files and line numbers, as explained in [4].

Apart from the code complexity, the time step in stationary RANS simulations typical of industrial applications is much bigger as it is not limited by the dynamics of turbulent structures in scale-resolving simulations. This, in turn, needs a more powerful linear solver for the Jacobi system in the implicit time integration. To mitigate this problem, our preconditioned BiCGSTAB solver had to be upgraded with slightly more complex preconditioners than the basic block Jacobi method. To prevent the solver from breaking down at large time steps, preconditioners based on the multicolor Gauss–Seidel (GS) method, typical for GPU computing, have been implemented in a heterogeneous way, combining MPI, OpenMP for CPUs and OpenCL for GPUs. This required notable additional effort, and to go further, it may be even necessary to use a multigrid approach and connect external solvers capable of working with sparse block matrices.

Finally, here are some particular examples. Boundary conditions used to take about 1–2% of the overall time when running on CPUs. When running on GPUs, leaving the BCs on the CPU side results in a cost of about 20%, where most of the time, around 3/4, is spent on the extra traffic between CPU and GPU devices (due to the need for transferring Jacobi matrix blocks). Porting the BCs to the GPU side, which is rather laborious, has reduced the cost to 2–3%, that is, an order of magnitude less. Similarly, on CPUs, the STG was consuming about 3%. Being too lazy to port it to the GPU also costs about 20%, where 1/4 goes to traffic, 3/4 to computational expenses. Porting the STG has shrinked its contribution to 4%. Regarding the linear solver upgrade, heterogeneous implementations of multicolor variants of GS-based preconditioners (GS, SGS, SOR, SSOR) demonstrate as high acceleration as for the block Jacobi method, which is about 8 times on NVIDIA V100 vs. 16-core Intel Xeon Gold.

3. Modularity

The ongoing expansion of functionality and complication of the code required a transition to a modular architecture. The code has been split into the core part and connectable modules and libraries, as shown in Fig. 1. The core contains the basic infrastructure and numerical methods.

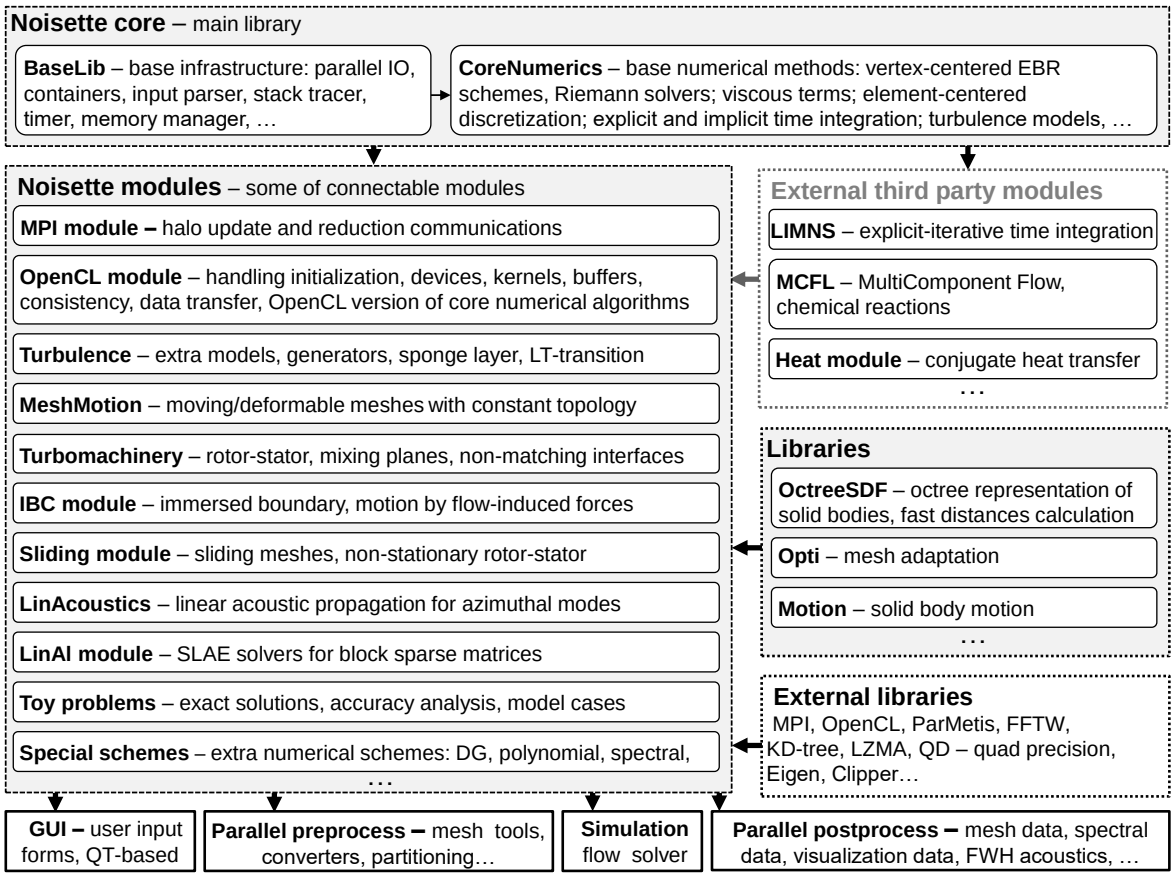

Figure 1. Modular structure of the simulation code

The infrastructure includes such things as parallel IO, containers for mesh data, user input parser, internal memory manager, stack tracer, timer, etc. The computing core contains basic numerical methods: the convective part of the NS equations, including vertex-centered EBR schemes and its cell-centered variants, Riemann solvers, low-Mach preconditioner; methods for calculating the viscous terms; explicit and implicit time integration; basic RANS and LES

turbulence models, etc. Modules contain extra functionality that works on a base of the core. Libraries, in contrast to modules, do not use the core. The code can be built without any library or module (even without MPI, OpenMP and OpenCL). Simply removing the source code folder of a module or library eliminates all of its functionality from the code. The build system automatically adapts to the available code configuration and sets the necessary definitions.

In the core simulation algorithm, functions are equipped wherever needed with connection points simply on a base of function pointers, allowing modules to override a function with its own implementation or to add its function calls into a function. If an operation has different options, such as different reconstructions, various Riemann solvers, etc., multiple-choice switches select the option defined by the user input. Such switches can be extended with more options using a special template class that stores more options, each given by a function pointer and a corresponding text label for user input (only the one selected by the user is active). Thus, modules can add their functions wherever needed, override basic ones, add more options, add its data to checkpoint records, visualization etc. For ideas on how this works, see Fig. 2. It appeared, that this very simple approach with function pointers allows to implement rather complex things without introducing changes into the core, such as multi component flows, chemical reactions, conjugated heat transfer, etc.

```
// Connection points to some function
typedef void (*tSomeFunc)(); // function pointer of some particular type
tSomeFunc ExtVersion=NULL; // external version of some function
vector<tSomeFunc> ExtActions; // external actions inside some function
void OverrideSomeFunction(tSomeFunc f){ ExtVersion = f; }
void AddActionToSomeFunction(tSomeFunc f){ ExtActions.push_back(f); }

void SomeFunction(){
  if(ExtVersion) return ExtVersion(); // replacement with an external version
  SomeBasicFunctionality();
  for(size_t i=0; i<ExtActions.size(); ++i) ExtActions[i](); // external actions
}

// Extensible implementation options
enum tBaseEnum{ BaseOption1 = 1, BaseOption2 = 2 };
tExtEnum<tBaseEnum, tSomeFunc> SomeChoice; // extensible enum wrapper
void SomethingWithVariousOptions(){
  switch((tBaseEnum)SomeChoice){
  case BaseOption1: return BaseFunction1(); // one of basic options
  case BaseOption2: return BaseFunction2(); // one of basic options
  default: // one of registered external options chosen in user input
    return SomeChoice.GetExtOption();
  }
}

// Registering module's functionality at initialization
void MyModuleInit(){
  OverrideSomeFunction(MyVersionOfSomeFunction);
  SomeChoice.RegisterOption(MyOptionForSomeChoice1, "myoption1");
  SomeChoice.RegisterOption(MyOptionForSomeChoice2, "myoption2");
}
```

Figure 2. An illustration of how connecting a module works

The last thing regarding modules is how to manage what is implemented for the GPU, what can be incorporated from the CPU, and what is not available for GPU computing. A special function checks this compatibility of the actual code configuration with the given user input and available implementations. If a module is requested in a GPU-enabled execution, and there is no GPU implementation available, the execution aborts with the information on what is missing and what should be changed to make it run. Note that it also takes some extra effort to implement and maintain such a compatibility check.

4. Reliability

Implementation consistency across different architectures is critical to heterogeneous software reliability. Additional measures, rather laborious, have been taken to protect the code from inconsistencies and errors. Such measures include expanding the quality assurance suite with tests covering all GPU-enabled features and implementing internal consistency checks inside the code. Internal checks are performed each time a GPU-enabled simulation is started by running several time integration steps on both the CPU and GPU. First, per-kernel CPU vs GPU version check for each OpenCL kernel involved in a particular numerical algorithm ensures that its CPU counterpart produces the same results (with some specified round-off tolerance). Second, a full time step check ensures that the whole time step algorithms are consistent by comparing mesh function after several time steps.

Another important issue is the fault tolerance of the simulation algorithm itself, regardless of the devices on which it is executed. Non-physical flow instabilities can appear due to insufficient mesh quality or resolution in the case of a low-dissipation scheme, too big time step size, insufficient linear solver accuracy, etc. To prevent simulation breakdown, the flow fields are checked every certain number of time steps to detect problems such as too high or too low density or pressure (these limits are case specific and are set by user depending e.g. on the actual Mach number), incorrect numbers, etc. In the case if flow fields turn out to be incorrect, the simulation automatically returns to the previously stored in memory restart checkpoint and adjusts the relevant parameters such as the time step size, solver tolerance, upwind and central difference weights, etc. When GPU computing is enabled, such checkpoints are stored in CPU memory to save scarce GPU memory. This requires additional transfers of mesh functions and processing routines in the GPU-enabled algorithm.

Conclusions

In CFD applications, GPUs increase performance by an order of magnitude compared to an equivalent number of CPUs. For this, the simulation algorithm must be adapted to the more restricted parallel paradigm used on GPUs. Reducing memory consumption is also critical for GPU computing. The underlying computational algorithm then needs to be efficiently implemented for the GPU architecture. But besides these obvious things, it turned out that there are still many problems that need to be solved, and a lot of work that needs to be done in order to effectively use a heterogeneous code in practice. This short communication provides a summary of such problems and how to solve them. A significant expansion of functionality towards industrial applications required a transition to a modular architecture. Porting additional components of the simulation algorithm to OpenCL, even those that take a negligible amount of computational time when running on the CPU, appeared to be unavoidable. Otherwise, the performance would be about twice as low. The growing amount of OpenCL code required additional measures to improve reliability. The need for stationary RANS simulations required the linear solver upgrade in the implicit time integration scheme. Eventually, the code seems to have become applicable for stationary and scale-resolving CFD simulations, including cases with synthetic turbulence; mixing-plane rotor-stator interfaces; various boundary conditions for solid surfaces, inflow, outflow; rotating coordinate system; various turbulence modeling approaches and models, etc. Practical performance on GPUs gives us the equivalent of 100 to 200 CPU cores per device, which is well worth the effort.

Acknowledgements

This work was funded by the RSF project No. 19-11-00299. The research is carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University [7], the equipment of Shared Resource Center of KIAM RAS (http://ckp.kiam.ru), the infrastructure of the Shared Research Facilities "High Performance Computing and Big Data" (CKP "Informatics") of FRC CSC RAS (Moscow). The author thankfully acknowledges these institutions.

References

1. Alvarez, X., Gorobets, A., Trias, F., *et al.*: HPC2 – A fully-portable, algebra-based framework for heterogeneous computing. Application to CFD. Computers & Fluids 173, 285–292 (2018), https://doi.org/10.1016/j.compfluid.2018.01.034

2. Bocharov, A., Evstigneev, N., Petrovskiy, V., *et al.*: Implicit method for the solution of supersonic and hypersonic 3D flow problems with Lower-Upper Symmetric-Gauss-Seidel preconditioner on multiple graphics processing units. Journal of Computational Physics 406, 109189 (2020), https://doi.org/10.1016/j.jcp.2019.109189

3. Borrell, R., Dosimont, D., Garcia-Gasulla, M., *et al.*: Heterogeneous CPU/GPU co-execution of CFD simulations on the POWER9 architecture: Application to airplane aerodynamics. Future Generation Computer Systems 107, 31–48 (2020), https://doi.org/10.1016/j.future.2020.01.045

4. Gorobets, A., Bakhvalov, P.: Heterogeneous CPU+GPU parallelization for high-accuracy scale-resolving simulations of compressible turbulent flows on hybrid supercomputers. Computer Physics Communications 271, 108231 (2022), https://doi.org/10.1016/j.cpc.2021.108231

5. Gorobets, A., Duben, A.: Technology for Supercomputer Simulation of Turbulent Flows in the Good New Days of Exascale Computing. Supercomputing Frontiers and Innovation 8(4), 4–10 (2021), https://doi.org/10.14529/jsfi210401

6. Niedermeier, C., Janssen, C., Indinger, T.: Massively-parallel multi-GPU simulations for fast and accurate automotive aerodynamics. In: Proceedings of the 7th European Conference on Computational Fluid Dynamics, Glasgow, Scotland, UK, June 11–15, 2018 (06 2018)

7. Voevodin, V., Antonov, A., Nikitenko, D., *et al.*: Supercomputer Lomonosov-2: Large Scale, Deep Monitoring and Fine Analytics for the User Community. Supercomput. Front. Innov. 6(2), 4–11 (2019), https://doi.org/10.14529/jsfi190201

8. Watanabe, S., Aoki, T.: Large-scale flow simulations using lattice Boltzmann method with AMR following free-surface on multiple GPUs. Computer Physics Communications 264, 107871 (2021), https://doi.org/10.1016/j.cpc.2021.107871