INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS
Int. J. Numer. Meth. Fluids 2012; **69**:1786–1796
Published online 11 August 2011 in Wiley Online Library (wileyonlinelibrary.com/journal/nmf). DOI: 10.1002/fld.2664

Semi-automatic porting of a large-scale CFD code to multi-graphics processing unit clusters

Andrew Corrigan and Rainald Löhner*,†

CFD Center, Department of Computational and Data Science M.S. 6A2, College of Science, George Mason University, Fairfax, VA 22030-4444, USA

SUMMARY

A typical large-scale CFD code based on adaptive, edge-based finite-element formulations for the solution of compressible and incompressible flow is taken as a test bed to port such codes to graphics hardware (graphics processing units, GPUs) using semi-automatic techniques. In previous work, a GPU version of this code was presented, in which, for many run configurations, all mesh-sized loops required throughout time stepping were ported. This approach simultaneously achieves the fine-grained parallelism required to fully exploit the capabilities of many-core GPUs, completely avoids the crippling bottleneck of GPU–CPU data transfer, and uses a transposed memory layout to meet the distinct memory access requirements posed by GPUs. The present work describes the next step of this porting effort, namely to integrate GPU-based, fine-grained parallelism with Message-Passing-Interface-based, coarse-grained parallelism, in order to achieve a code capable of running on multi-GPU clusters. This is carried out in a semi-automated fashion: the existing Fortran–Message Passing Interface code is preserved, with the translator inserting data transfer calls as required. Performance benchmarks indicate up to a factor of 2 performance advantage of the NVIDIA Tesla M2050 GPU (Santa Clara, CA, USA) over the six-core Intel Xeon X5670 CPU (Santa Clara, CA, USA), for certain run configurations. In addition, good scalability is observed when running across multiple GPUs. The approach should be of general interest, as how best to run on GPUs is being presently considered for many so-called legacy codes. Copyright © 2011 John Wiley & Sons, Ltd.

Received 9 February 2011; Revised 4 July 2011; Accepted 12 July 2011

KEY WORDS: CFD; GPUs; supercomputing hardware

1. INTRODUCTION

At present, there is considerable interest in the application of graphics processing units (GPUs) to CFD. This interest is primarily due to the tremendous increase in performance of GPUs over the past few years. The NVIDIA Tesla C2070 (Santa Clara, CA, USA) now achieves more than 0.5 teraflops of peak double-precision performance with a peak memory bandwidth of 144 GB/s. The consumer-market-oriented GeForce GTX 580 (NVIDIA) achieves more than 1.58 teraflops of peak single-precision performance with a peak memory bandwidth of 192.4 GB/s [1, 2]. GPUs can now be programmed via general-purpose interfaces such as CUDA [1] and OpenCL [3].

As a result, there has been a great deal of research investigating the implementation of CFD codes on GPUs, with many codes obtaining impressive speedups. Work in this direction began prior to the introduction of CUDA, when GPU programming was performed via traditional graphics APIs such as OpenGL. An excellent and comprehensive survey of work carried out during this era is given by Owens *et al.* [4]. Much of the effort in running CFD codes on GPUs has been directed towards the case of solvers based on structured grids. These solvers are particularly amenable to GPU

*Correspondence to: Rainald Löhner, CFD Center, Department of Computational and Data Science M.S. 6A2, College of Science, George Mason University, Fairfax, VA 22030-4444, USA.

†E-mail: rlohner@gmu.edu

implementation because of their regular memory access pattern. Work in this area includes those of Brandvik and Pullan [5–7] (two-dimensional and three-dimensional Euler and Navier–Stokes solvers for GPUs, with support for multiple GPUs via Message Passing Interface (MPI), achieved an order of magnitude gain in performance); Göddeke *et al.* [8] (multi-level, globally unstructured, locally structured, Navier–Stokes solver on GPUs); LeGresley *et al.* [9] (multi-block Euler solver for simulating hypersonic vehicle configurations on GPUs); Cohen and Molemaker [10] (3D finite-volume Boussinesq code in double precision on GPUs); Phillips *et al.* [11] (2D compressible Euler solver on a cluster of GPUs); Thibault and Senocak [12] and Jacobsen *et al.* [13] (3D incompressible Navier–Stokes solver for GPU clusters); Antoniou *et al.* [14] (high-order weighted essentially non-oscillatory finite-difference methods on multi-GPU systems); Jespersen [15] (port of Jacobi solver to GPUs for OVERFLOW, a Reynolds-averaged Navier–Stokes solver that uses multiple overset structured grids); Patnaik and Obenschain [16] (application of GPUs to the flux-corrected transport (FCT)-based structured grid solver FAST3D-CT, with a focus on reducing power requirements); and Griebel and Zaspel [17] (multi-GPU, two-phase incompressible Navier–Stokes solver).

There has also been interest in running unstructured grid-based CFD solvers on GPUs. Achieving good performance for such solvers is more difficult because of their data-dependent and irregular memory access patterns. Work in this area includes that of Klöckner *et al.* [18], who have implemented discontinuous Galerkin methods on unstructured grids. Markall *et al.* [19] developed a form of compiler to automatically generate a finite-element code from an abstract representation in the Unified Form Language, which will be used to generate a GPU-optimized CUDA code for the CFD code Fluidity. In previous work [20], the authors presented results for a cell-centered finite-volume Euler solver running on a Tesla 10 series card, which achieved a nearly 10× speedup factor over an OpenMP-parallelized CPU code running on a quad-core Intel CPU (Santa Clara, CA, USA). Similar results, for a two-dimensional edge-based solver, were also presented by Dahm and Fidkowski [21]. Asouti *et al.* [22] and Kampolis *et al.* [23] have also implemented a vertex-centered finite-volume code for unstructured grids on GPUs.

Whereas most of the existing work performed so far has either been for relatively small codes written from scratch or for a small portion of a large existing code, the goal of the present work is to port an existing large-scale code, used to perform production runs, in its entirety while obtaining a similar performance gain. The code under consideration is FEFLO, which is an adaptive, edge-based finite-element code for the solution of compressible and incompressible flow. It consists (like so many so-called legacy codes) of nearly one million lines of primarily Fortran 77 code and is optimized for many types of parallel architectures. This code is used for a number of compressible flow applications including supersonic jet noise, transonic flow, store separation, and blast–structure interaction, as well as incompressible flow applications including free-surface hydrodynamics, dispersion, and patient-based hemodynamics. As will be described in Section 2, large portions of the code have already been ported to run on single GPUs, by using a semi-automatic porting technique described in previous work [24–26]. A description of the currently ported capabilities is then given in Section 3. The porting effort for the MPI portion of FEFLO is presented in Section 4. Finally, performance results that assess the scalability of the code are given in Section 5.

2. SEMI-AUTOMATIC PORTING OF FEFLO TO GPUs

For the exploitation of the high performance of GPUs, codes must express sufficient fine-grained parallelism, minimize GPU–CPU data transfer, and achieve coalesced memory access. One option for writing the GPU code is to do so directly in CUDA [1] or OpenCL [27]. For a large Fortran code written well before the existence of GPUs, this entails a massive effort to rewrite, debug, and maintain a separate branch of the code. An alternative to performing such a monolithic code rewrite is to follow the 'accelerator design', as so called by Cohen and Molemaker [10], of porting bottleneck loops or subroutines in isolation, with memory transfer calls made just before and after these bottleneck loops or subroutines. Although this allows for the possibility of an incremental porting effort, it suffers the drawback of introducing the crippling bottleneck of a large amount of data transfer across the system bus, which has at least an order of magnitude lower bandwidth than that of the internal GPU memory, and thus severely restricts the possible gain in performance when porting a

DOI: 10.1002/fld

code to run on GPUs. Instead, all large arrays should be kept for as long as possible in the GPU memory space. In the context of a CFD code, this means that ideally all mesh-sized arrays should remain on the GPU throughout the course of time stepping, which in turn entails parallelizing all code within time stepping that makes use of such arrays.

As described in previous work [24–26], a semi-automatic approach is used to port FEFLO to run on GPUs, so that the GPU–CPU data transfer bottleneck can be avoided. The script is fully automatic for OpenMP-parallelized Fortran code, which meets the script's necessary restrictions, which ensure full GPU performance. This approach uses the original Fortran code by automatically generating GPU kernels and array management code from existing OpenMP-parallelized loops, while enforcing restrictions on array usage. Manual effort is required to first parallelize the original code using OpenMP and then to modify the code to ensure that GPU data is only accessed within GPU kernels or through other well-defined mechanisms and not via an arbitrary serial code, which executes on the CPU. In addition, manual effort is required to ensure that widely used assumptions regarding the layout of multi-dimensional Fortran arrays in memory are avoided, in order to employ a transposed memory layout. These techniques are implemented as a Python script based on FParser, a component of the F2PY package created by Peterson [28]. Because this approach automates the majority of the porting process, all mesh-sized loops can be immediately parallelized, thus avoiding the crippling bottleneck of data transfer between the GPU and the CPU. The primary reasons for this approach, in comparison to a manual approach, are the massive size of the FEFLO codebase ($O(10^6)$ lines) and the desire to continue using established coding practices under a unified codebase, thus avoiding the introduction of new bugs in an already reliable code and the perpetual process of maintaining divergent codebases. A further advantage of this approach is that it supports multiple output targets, both CUDA and OpenCL, thus maintaining flexibility as new parallel architectures emerge, GPU or otherwise.

Automatic approaches often compromise on performance, which contradicts the ultimate purpose of porting codes such as FEFLO to GPUs. That is not the case here because the translator was specialized to generate the same code that would be written via a manual translation. In summary, the script automatically does the following:

- Converts simple OpenMP loops to CUDA kernels, with support for arbitrary reduction operations;
- Exposes finer-grained parallelism in coarse-grained OpenMP loops using FEFLO-specific logic;
- Detects GPU arrays and enforces consistency between modules and subroutines;
- Tracks physical array sizes of subarrays across subroutine calls;
- Uses a transposed array layout appropriate for meeting coalescing requirements;
- Handles GPU array I/O (via read/write/print) and memory transfer;
- Allows 'difficult' subroutines to be ignored and left on the CPU, or overridden with custom implementations, usually according to Thrust [29]; and
- Integrates with a pure CPU code, particularly MPI code.

3. CURRENT GPU CAPABILITIES

FEFLO is used for a number of compressible flow applications including supersonic jet noise [30], transonic flow, store separation [31], and blast–structure interaction [32, 33], as well as incompressible flow applications including free-surface hydrodynamics [34], dispersion [35], and patient-based hemodynamics [36]. Codes with similar loop structures are those described in [37–41].

To date, approximately 1600 of the flow solver(s) subroutines have been automatically ported and tested on the GPU. These include the following:

- The ideal gas, explicit compressible flow solvers (FCT, total variation diminishing, approximate Riemann solvers, limiters);
- The projection schemes for incompressible flow with explicit advection, implicit viscous terms, and pressure Laplacian;

DOI: 10.1002/fld

- The explicit advection and implicit conductivity/diffusivity integrators of temperature and/or concentration;
- The free-surface modules based on the volume of fluid (VOF) and level set techniques;
- Some simple turbulence models (e.g., Smagorinsky, wall-adapting local eddy viscosity);
- Some of the diagnostic output options (global conservation sums, station time history points); and
- Subroutines required throughout the code/solvers to apply boundary conditions for periodic boundaries, overlapping grids, embedded surfaces, and immersed bodies.

Other parts that have been ported but not yet fully tested include the following:

- Other turbulence models (e.g., k-ϵ);
- Mesh movement modules (velocity smoothing, distance to wall, element quality analysis, etc.);
- H-refinement modules; and
- Radiation transport via the discrete ordinate method.

Testing, verification, and validation of these modules for the multi-GPU environment are currently underway, and it is expected that the remaining modules of FEFLO (which will undoubtedly be more challenging) will be ported to the multi-GPU environment during the next 2 years.

4. MPI SUPPORT

The code has been ported to vector, shared-memory parallel (via OpenMP [42]), and distributed-memory parallel (via MPI [43]) machines. Coarse-grained parallelism is orthogonal to fine-grained parallelism, and a combination of the two can be used. In the particular case of a CFD code, a cluster of multi-core CPUs may use domain splitting with inter-processor boundary communication implemented using MPI to achieve coarse-grained, distributed-memory parallelism and then using OpenMP-parallel loops to achieve fine-grained, shared-memory parallelism within the various subdomains. In fact, this is precisely what FEFLO does, and production runs performed with FEFLO often specify multiple OpenMP threads per MPI rank.

Distributed-memory parallelism is implemented in FEFLO using MPI [43]. For distributed-memory parallel machines, load balancing is achieved via domain splitting [44]. Throughout FEFLO, one layer of overlap elements is used [43, 45]. In this way, all subroutines remain the same as in the usual shared-memory case. Only once a right-hand side has been assembled (and this may involve calling many separate subroutines) can an exchange of values at the domain boundaries take place. Naturally, this implies that the number of elements and points is larger for the split problem than the original unsplit one. However, once the number of elements per domain exceeds 0.3 million elements (Mels), the extra number of points and elements is negligible. Moreover, once the domain has less than 0.3 Mels, even on current CPUs and the fastest networks, communication costs start to dominate.

FEFLO's domain splitter, named FESPLIT, offers a variety of splitting and load balancing options, including advancing front (greedy) techniques and several recursive bisection algorithms (coordinate, moment, etc.), which can be used in conjunction with linelet preconditioning, periodic boundary conditions, and so-called ring regions for turbomachinery applications. For the cases shown here, the advancing front algorithm was employed. The load imbalance in the different domains, as measured by the number of elements, was less than 1% for all cases shown.

The message-passing subroutines have been assembled into a library, which abstracts MPI to provide the high-level communication functionality required by FEFLO. The subroutines contained in this library, which are indicated by the prefix `mpp_`, can be used, essentially unmodified, to enable multi-GPU applications because, as mentioned in Section 2, the Python script automatically wraps these MPI calls with data transfer calls. In particular, when it detects a call to a subroutine to which it does not have access to the source code, it generates wrapper code to transfer all GPU arrays passed to that subroutine to and from the CPU memory space. This functionality is intentional: whereas GPU arrays are typically passed directly through subroutine calls without any data transfer, external libraries for which the source code is not available cannot be automatically ported. Although the

DOI: 10.1002/fld

```
      subroutine mpp_rsendrecv(mstag, sdata, nsdat,
     &                                 rdata, nrdat, kproc)
c
      use arrays_mpi
c
      implicit real*8 (a-h,o-z)
c
      include 'mpp.h'
c
c
c     -----this sub sends an real*8 array to the specified processor
c
c
c     -----arguments
c
      real*8 sdata(nsdat),rdata(nrdat)
      integer istat(MPI_STATUS_SIZE)
c
c     -----send
c
!$gpu  cpu(sdata)
!$gpu  cpu(rdata),notransfer
c
          call mpi_sendrecv(sdata,nsdat,MPI_REAL8,kproc,mstag,
     &                      rdata,nrdat,MPI_REAL8,kproc,mstag,
     &                      MPI_COMM_WORLD,istat,ierr)
c
!$gpu  end cpu(rdata)
!$gpu  end cpu(sdata),notransfer
c
c     -----done
c
 9999 continue
      return
      end
```

Listing 1. Additional !$gpu directives are used to avoid unnecessary data transfer between the GPU and the CPU.

automatically generated data transfer code is guaranteed to be correct, it can be optimized further, as the data transfer is only required in one direction because for many MPI calls the array arguments are read from or written to, but not both.

Consider the case of the `mpi_sendrecv` subroutine call, shown in Listing 1, which is called from the subroutine `mpp_rsendrecv`, the subroutine responsible for the majority of inter-processor boundary communication during time stepping. The array `sdata` is sent to `kproc`, whereas data are received into the array `rdata` from `kproc`. Because these arrays are used in OpenMP loops that appear in other subroutines, the translator designates them as GPU arrays, which cannot be directly passed to MPI and must be transferred to a temporary CPU buffer. Left unmodified, the script will correctly transfer the arrays to and from the CPU. But because each array only needs to be transferred once, the `!$gpu` directives are used to explicitly disable the superfluous memory transfer, as shown in Listing 2, by adding the `notransfer` option. The resulting automatically generated GPU version of the code is then shown in Listing 2. Notice that, as desired, the array `sdata` is sent to the CPU from the GPU, but it is not sent back afterwards. The reverse is the case for the array `rdata`.

5. RESULTS

The GPU version of FEFLO was tested for a variety of benchmark cases. Three cases are included here: blast in a room (compressible Euler, FEM-FCT), pipe flow (incompressible Navier–Stokes

DOI: 10.1002/fld

```
subroutine mpp_rsendrecv_gpu(mstag,sdata,nsdat,rdata,nrdat,kproc)

    use arrays_mpi
    use gpu
    implicit real(kind=8) (a-h,o-z)
    include 'mpp.h'    ! Note: The contents of mpp.h appear inline
                       ! in the automatically generated code
    !     -----this sub sends a real*8 array to the specified processor
    !     -----arguments
    type(da_real1) :: sdata
    integer :: i_host_array_offset_real_sdata
    type(da_real1) :: rdata
    integer :: i_host_array_offset_real_rdata
    integer :: istat(mpi_status_size)
    !     -----send
!         ----- Directive removed
!         ----- Directive removed
    i_host_array_offset_real_sdata=i_host_array_offset_real
    i_host_array_offset_real=i_host_array_offset_real+(nsdat)
    call memcpy_device_to_host_double1((/nsdat/),(/1/),(/nsdat/),      &
                                       host_array_real,                &
                                       i_host_array_offset_real_sdata, &
                                       sdata)
    i_host_array_offset_real_rdata=i_host_array_offset_real
    i_host_array_offset_real=i_host_array_offset_real+(nrdat)
    call mpi_sendrecv(host_array_real(i_host_array_offset_real_sdata), &
                      nsdat,mpi_real8,kproc,mstag,                     &
                      host_array_real(i_host_array_offset_real_rdata), &
                      nrdat,mpi_real8,kproc,mstag,                     &
                      mpi_comm_world,istat,ierr)
!         ----- Directive removed
!         ----- Directive removed
    !     -----done
    call memcpy_host_to_device_double1((/nrdat/),(/1/),(/nrdat/),      &
                                       host_array_real,                &
                                       i_host_array_offset_real_rdata, &
                                       rdata)
    i_host_array_offset_real=i_host_array_offset_real_rdata
    i_host_array_offset_real=i_host_array_offset_real_sdata
    9999 continue
end subroutine mpp_rsendrecv_gpu
```

Listing 2. The automatically generated data transfer code (manually formatted to fit in this listing).

with heat transfer), and a dam break problem (incompressible Navier–Stokes with VOF treatment of the free surface). The hardware used for performance benchmarking was a multi-GPU cluster, where each node contains two NVIDIA Tesla M2050 GPUs and two six-core Intel Xeon Processor X5670 CPUs. All CUDA kernels were executed with a thread block size of 128. Other thread block sizes were also tested but did not lead to any substantial differences in performance.

Previous studies have shown that in order to achieve proper performance on the GPUs the vector length should be set as high as possible, but not lower than nvecl = 12,000 (this number is, of course, dependent on the particular solver and implementation). On the other hand, the higher the vector length, the higher is the probability of cache misses. Therefore, for the Xeon CPU, the vector length is set as small as possible to achieve good vectorization/pipelining. For most of the solvers in FEFLO, this number is of the order of nvecl = 32. The mesh sizes shown are of the order of 1–4 Mels. This was the limit of what could be fitted into the GPUs used. Larger meshes could have been run on the Xeon CPUs, but this was not considered for a fair comparison. Therefore, the same grids were run for both the CPUs and the GPUs.

In the tables that follow, nelem denotes the number of elements in the mesh, nvecl the vector length used, and ndomn the number of domains (i.e., MPI processes) into which the original mesh was split.

DOI: 10.1002/fld

5.1. *Blast in a room*

This example considers the compressible flow resulting from a blast in a room. The geometry, together with the solution, can be discerned from Figure 1.

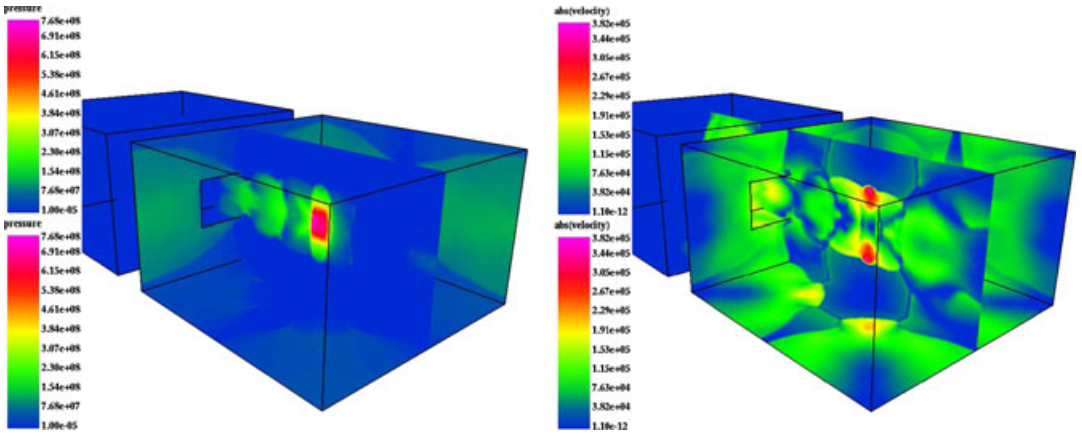
The compressible Euler equations are solved using an edge-based FEM-FCT technique [45–47]. The initialization was performed by interpolating the results of a very detailed 1-D (spherically symmetric) run. The timing studies were carried out with the following set of parameters:

- Compressible Euler, ideal gas equation of state;
- Explicit FEM-FCT;
- Initialization from a 1-D file; and
- Run for 300 steps.

Table I summarizes the runs performed. Note the rather quick degradation in scaling for the GPUs when working on domains with a small number of elements. This is due to the fact that once the subdomains become too small (less than a million elements), the communication between domains starts to dominate the overall computation times. One can see that for the larger meshes the explicit FEM-FCT solver on the GPU achieves speeds that are equivalent to approximately 16 Xeon cores. Remarkably, the MPI scaling on the six-core Xeon runs is superlinear up to eight domains (0.5 Mels/domain). Even the 12-core Xeon runs display an almost linear reduction in CPU with the number of domains.

5.2. *Pipe*

This example considers the laminar inlet flow of a cold fluid into a pipe with hot walls. The incompressible Navier–Stokes equations are solved using a projection technique for pressure increments [45, 48]. Therefore, a large portion of the compute time is consumed by the diagonally preconditioned conjugate solvers of the pressure increments. Additionally, the energy equation for the temperature is integrated simultaneously with the velocities and pressures. Figure 2 shows the



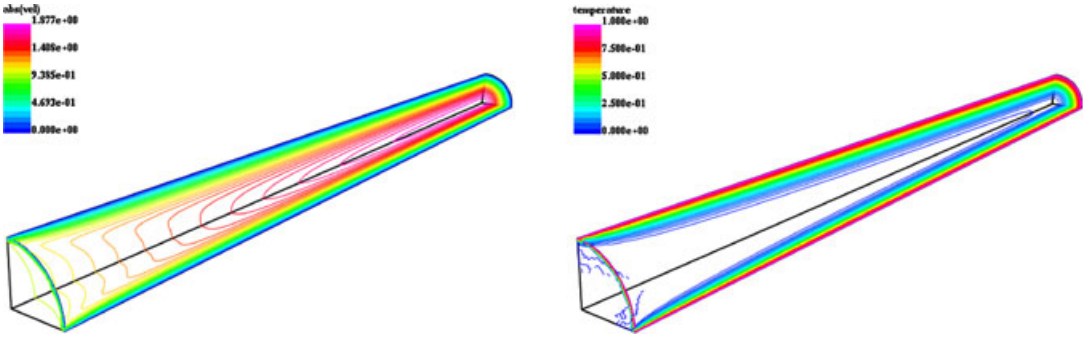
(a) Pressure (b) Velocity

Figure 1. Blast in a room: pressures and velocities.

Table I. Blast in a room.

nelem	CPU/GPU	nvecl	Time (s) ndomn = 1	ndomn = 2	ndomn = 4	ndomn = 8
4.0 M	Xeon X5670 (6)	32	315	136	68	36
4.0 M	Xeon X5670 (12)	32	191	102	59	33
4.0 M	Tesla M2050	51,200	151	84	48	30
2.0 M	Tesla M2050	51,200	—	—	15	11

DOI: 10.1002/fld



(a) Velocity (b) Temperature

Figure 2. Pipe: velocities and temperature.

typical results obtained. The timing studies were carried out with the following set of parameters:

- Incompressible Navier–Stokes + heat transfer;
- Advection: RK3, Roe, nlimi = 2;
- Pressure: Poisson (projection), deflated preconditioned conjugate gradient;
- Steady state, local time stepping;
- Re = 100; and
- Run for 100 steps.

The timing obtained have been summarized in Table II. For this case, the 4-Mels case did not fit on the GPU. Therefore, only the results from the partitioned domain are shown for the GPU. For this set of solvers in FEFLO, the MPI scaling is almost perfect for both the CPUs and the GPUs. Note the superlinear decrease from ndomn = 2 to ndomn = 4 for the Xeons and from ndomn = 4 to ndomn = 8 for the GPUs. These are remarkable results, requiring an explanation that is still lacking. Note that for the larger grids, for this set of solvers in FEFLO, the GPU speeds are equivalent to approximately eight Xeon cores. This is less favorable than the compressible FEM-FCT solvers and is due to the fact that a large portion of the CPU time is spent by the diagonally preconditioned conjugate solvers of the pressure increments. These have much more memory transfer per floating point operations as compared with the FEM-FCT solver, which explains the degradation in performance.

5.3. Dam break

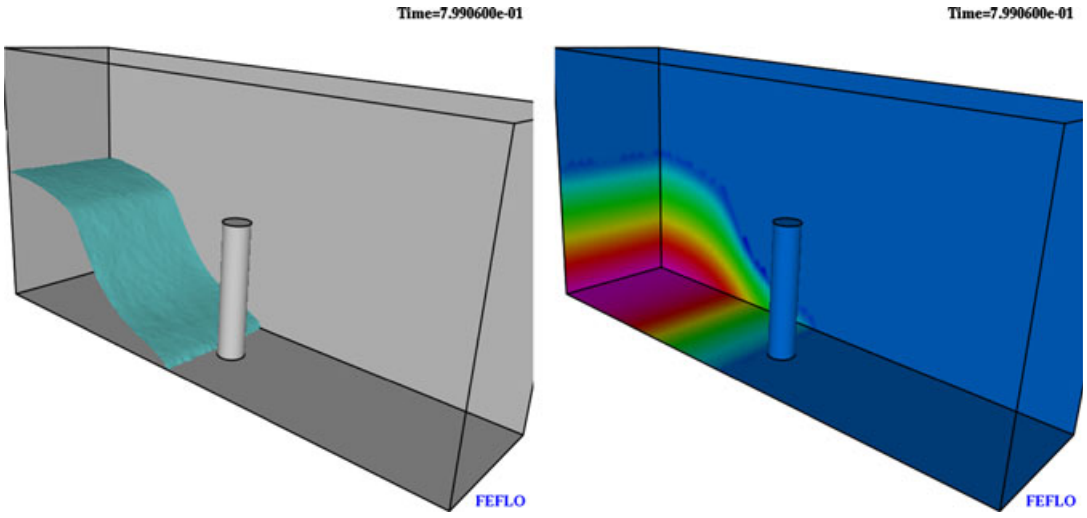
This example considers the classic dam break problem, where the wave impinges on a column situated in the middle of the channel. The incompressible Navier–Stokes equations are solved using a projection technique for pressure increments, together with a transport equation for the volume of fluid fraction. Details of the technique may be found in [34]. Figure 3 shows the typical results obtained. The timing studies were carried out with the following set of parameters:

- Incompressible Navier–Stokes;
- VOF for free surface;
- Advection: explicit RK3, Roe, nlimi = 2;
- Pressure: Poisson (projection), deflated preconditioned conjugate gradient;

Table II. Pipe.

nelem	CPU/GPU	nvecl	Time (s) ndomn = 1	ndomn = 2	ndomn = 4	ndomn = 8
4.0 M	Xeon X5670 (6)	32	1105	684	212	112
4.0 M	Xeon X5670 (12)	32	662	336	152	75
4.0 M	Tesla M2050	51,200	—	456	371	183

DOI: 10.1002/fld



(a) Free Surface (b) Pressure

Figure 3. Dam break: free surface and pressure.

Table III. Dam break.

nelem	CPU/GPU	nvecl	Time (s) ndomn = 1	ndomn = 2	ndomn = 4	ndomn = 8
4.0 M	Xeon X5670 (6)	32	195	110	64	46
4.0 M	Xeon X5670 (12)	32	142	77	48	36
4.0 M	Tesla M2050	51,200	—	125	85	68

- Transient; and
- Run for 100 steps.

The timing obtained have been summarized in Table III. As before, for this case, the 4-Mels case did not fit on the GPU. Therefore, only the results from the partitioned domain are shown for the GPU. For this set of solvers in FEFLO, the MPI scaling is not as favorable for both the CPUs and the GPUs. This is because the VOF solver deactivates regions of the mesh where no fluid is present. This can lead to load imbalances that are difficult to control when only one partition of the domain is performed at the beginning of the run. Note that for the larger grids, for this set of solvers in FEFLO, the GPU speeds are equivalent to approximately four Xeon cores. This is less favorable than the compressible FEM-FCT solvers and the incompressible solvers and is due to the large number of VOF options invoked for this case. Most of these work only at the interface, making it difficult to achieve large vector lengths.

6. CONCLUSIONS AND OUTLOOK

The application of semi-automatic techniques to combine GPU-based, fine-grained parallelism with MPI-based, coarse-grained parallelism has been successful in porting the large-scale CFD code FEFLO to run on multi-GPU clusters. Detailed performance benchmarks indicate up to a factor of 2 performance advantage of the NVIDIA Tesla M2050 GPU over the six-core Intel Xeon X5670 CPU running in double precision for certain solvers and run configurations. In addition, good scalability is observed when running across multiple GPUs. The approach should be of general interest, as how best to run on GPUs is being presently considered for many so-called legacy codes.

The porting process of FEFLO will continue, with further developments focused on porting many challenging subroutines that do not lend themselves immediately to GPU parallelization, because of either insufficient parallelism or load balancing issues in their present implementation. Future work

DOI: 10.1002/fld

will also investigate the optimization of the bottleneck GPU kernels of FEFLO for various solver configurations.

ACKNOWLEDGEMENTS

The authors would like to thank NVIDIA Corporation and, in particular, Stan Posey, Steve Rohm, and Tom Reed for providing hardware for testing and development, as well as Nathan Bell and Jared Hoberock of NVIDIA Research for their implementation and support of the Thrust library, which has been absolutely critical in porting FEFLO to GPUs.

REFERENCES

1. NVIDIA Corporation. NVIDIA CUDA 3.2 programming guide, 2010.
2. NVIDIA Corporation. Fermi compute architecture white paper, 2009.
3. Khronos OpenCL Working Group. The OpenCL specification: version 1.0 rev. 48, 2009.
4. Owens JD, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn AE, Purcell TJ. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum* 2007; **26**:80–113.
5. Brandvik T, Pullan G. Acceleration of a two-dimensional Euler flow solver using commodity graphics hardware. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science* 2007; **221**:1745–1748.
6. Brandvik T, Pullan G. Acceleration of a 3D Euler solver using commodity graphics hardware. *46th AIAA Aerospace Sciences Meeting and Exhibit*, number AIAA-2008-607, January 2008.
7. Brandvik T, Pullan G. An accelerated 3D Navier–Stokes solver for flows in turbomachines. *Proceedings of GT2009 ASME Turbo Expo 2009: Power for Land, Sea and Air*, June 2009.
8. Göddeke D, Buijssen SHM, Wobker H, Turek S. GPU acceleration of an unmodified parallel finite element Navier–Stokes solver. *High Performance Computing & Simulation*, 2009; 12–21.
9. LeGresley P, Elsen E, Darve E. Large calculation of the flow over a hypersonic vehicle using a GPU. *Journal of Computational Physics* 2008; **227**:10148–10161.
10. Cohen JM, Molemaker MJ. A fast double precision CFD code using CUDA. *Parallel Computations Fluid dynamics (ParCFD)*, Moffett Field, CA, 2009, May 18–22.
11. Phillips EH, Zhang Y, Davis RL, Owens JD. Rapid aerodynamic performance prediction on a cluster of graphics processing units. *47th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, number AIAA 2009-565, January 2009.
12. Thibault J, Senocak I. CUDA implementation of a Navier–Stokes solver on multi-GPU desktop platforms for incompressible flows. *47th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, number AIAA 2009-758, January 2009.
13. Jacobsen D, Thibault J, Senocak I. An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, number AIAA-2010-522, January 2010.
14. Antoniou AS, Karantasis KI, Polychronopoulos ED, Ekaterinaris JA. Acceleration of a finite-difference WENO scheme for large-scale simulations on many-core architectures. *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, number AIAA-2010-0525, January 2010.
15. Jespersen DC. Acceleration of a CFD code with a GPU. *Technical Report NAS-09-003*, NAS, November 2009.
16. Patnaik G, Obenschain KS. Using GPUs on HPC applications to satisfy low-power computational requirements. *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, number AIAA-2010-524, January 2010.
17. Griebel M, Zaspel P. A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier–Stokes equations. *Computer Science—Research and Development* May 2010; **25**(1–2):65–73.
18. Klöckner A, Warburton T, Bridge J, Hesthaven JS. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics* 2009; **228**:7863–7882.
19. Markall GR, Ham DA, Kelly PHJ. Towards generating optimised finite element solvers for GPUs from high-level specifications. *Proceedings of the 10th International Conference on Computational Science*, June 2010.
20. Corrigan A, Camelli FE, Löhner R, Wallin J. Running unstructured grid-based CFD solvers on modern graphics hardware. *International Journal for Numerical Methods in Fluids* 2010; **66**:221–229.
21. Dahm JPS, Fidkowski KJ. Employing coprocessors to accelerate numerical solutions to the Euler equations, 2009. Available from: http://www.johanndahm.com/research.php.
22. Asouti VG, Trompoukis XS, Kampolis IC, Giannakoglou KC. Unsteady CFD computations using vertex-centered finite volumes for unstructured grids on graphics processing units. *International Journal for Numerical Methods in Fluids* 19 May 2010. DOI: 10.1002/fld.2352.
23. Kampolis IC, Trompoukis XS, Asouti VG, Giannakoglou KC. CFD-based analysis and two-level aerodynamic optimization on graphics processing units. *Computer Methods in Applied Mechanics and Engineering* 2010; **199**(9–12):712–722.

24. Corrigan A, Camelli F, Löhner R, Mut F. Porting of an edge-based CFD solver to GPUs. *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, number AIAA-2010-522, January 2010.
25. Corrigan A, Camelli F, Löhner R, Mut F. Porting of FEFLO to GPUs. *ECCOMAS CFD 2010: Fifth European Conference on Computational Fluid Dynamics*, June 2010.
26. Corrigan A, Camelli F, Löhner R, Mut F. Semi-automatic porting of a large-scale FORTRAN CFD code to GPUs. *International Journal for Numerical Methods in Fluids* 2 May 2011. DOI: 10.1002/fld.2560.
27. Khronos OpenCL Working Group. The OpenCL specification: version 1.1 rev. 36, 2010.
28. Peterson P. F2PY: a tool for connecting Fortran and Python programs. *International Journal of Computational Science and Engineering* 2009; **4**:296–305.
29. Hoberock J, Bell N. Thrust: a parallel template library, version 1.3, 2010.
30. Liu J, Kailasanath K, Ramamurti R, Gutmark E, Munday D, Löhner R. Large-eddy simulations of a supersonic jet and its near-field acoustic properties. *AIAA Journal* 2009; **8**(47):1849–1864.
31. Baum JD, Luo H, Löhner R, Goldberg E, Feldhun A. Application of unstructured adaptive moving body methodology to the simulation of fuel tank separation from an F-16 C/D fighter. *AIAA Aerospace Sciences Meeting*, number AIAA-1997-0166, January 1997.
32. Baum JD, Luo H, Löhner R, Yang C, Pelessone D, Charman C. A coupled fluid/structure modeling of shock interaction with a truck. *AIAA Aerospace Sciences Meeting*, number AIAA-1996-0795, January 1996.
33. Baum JD, Luo H, Mestreau E, Löhner R, Pelessone D, Charman C. A coupled CFD/CSD methodology for modeling weapon detonation and fragmentation. *AIAA Aerospace Sciences Meeting*, number AIAA-1999-0794, January 1999.
34. Löhner R, Yang C, Oñate E. Simulation of flows with violent free surface motion and moving objects using unstructured grids. *International Journal of Numerical Methods in Fluids* 2007; **53**:1315–1338.
35. Camelli F, Löhner R. VLES study of flow and dispersion patterns in heterogeneous urban areas. *AIAA Aerospace Sciences Meeting*, number AIAA-2006-1419, January 2006.
36. Appanaboyina S, Mut F, Löhner R, Putman CM, Cebral JR. Computational fluid dynamics of stented intracranial aneurysms using adaptive embedded unstructured grids. *International Journal for Numerical Methods in Fluids* 2008; **5**(57):475–493.
37. Mavriplis D. Three-dimensional unstructured multigrid for the Euler equations. *AIAA CFD Conference*, number AIAA-1991-1549-CP, June 1991.
38. Peraire J, Peiro J, Morgan K. A three-dimensional finite element multigrid solver for the Euler equations. *AIAA Aerospace Sciences Meeting*, number AIAA-1992-0449, January 1992.
39. Anderson WK, Gropp WD, Kaushik DK, Keyes DE, Smith BF. Achieving high sustained performance in an unstructured mesh CFD application. *Supercomputing 1999, IEEE Computer Society*, November 1999.
40. Aumann P, Barnewitz H, Becker K, Heinrich R, Roll B, Galle M, Kroll N, Gerhold T, Schwamborn D, Franke M. MEGAFLOW: parallel complete aircraft CFD. *Parallel Computing* 2001; **27**:415–440.
41. Nakahashi K, Ito Y, Togashi F. Some challenges of realistic flow simulations by unstructured grid CFD. *International Journal for Numerical Methods in Fluids* 2008; **43**:769–783.
42. Löhner R. Renumbering strategies for unstructured-grid solvers operating on shared-memory, cache-based parallel machines. *Computer Methods in Applied Mechanics and Engineering* 1998; **163**:95–109.
43. Ramamurti R, Löhner R. A parallel implicit incompressible flow solver using unstructured meshes. *Computers and Fluids* 1996; **5**:119–132.
44. Löhner R, Ramamurti R. A load balancing algorithm for unstructured grids. *International Journal of Computational Fluid Dynamics* 1995; **5**:39–58.
45. Löhner R. *Applied CFD Techniques*. J. Wiley & Sons: Hoboken, 2008.
46. Löhner R, Morgan K, Peraire J, Vahdati M. Finite element flux-corrected transport (FEM-FCT) for the Euler and Navier–Stokes equations. *International Journal for Numerical Methods in Fluids* 1987; **7**(10):1093–1109.
47. Luo H, Baum JD, Löhner R. Edge-based finite element scheme for the Euler equations. *AIAA Journal* 1994; **32**(6):1183–1190.
48. Löhner R, Yang C, Cebral JR, Camelli F, Soto O, Waltz J. Improving the speed and accuracy of projection-type incompressible flow solvers. *Computer Methods in Applied Mechanics and Engineering* 2006; **23–24**(195): 3087–3109.