INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS
Int. J. Numer. Meth. Fluids 2012; **69**:314–331
Published online 3 May 2011 in Wiley Online Library (wileyonlinelibrary.com/journal/nmf). DOI: 10.1002/fld.2560

Semi-automatic porting of a large-scale Fortran CFD code to GPUs

Andrew Corrigan[‡], Fernando Camelli, Rainald Löhner[*,†] and Fernando Mut

Center for Computational Fluid Dynamics, Department of Computational and Data Sciences, M.S. 6A2, College of Science, George Mason University, Fairfax, VA 22030-4444, U.S.A.

SUMMARY

The development of automatic techniques to port a substantial portion of FEFLO, a general-purpose legacy CFD code operating on unstructured grids, to run on GPUs is described. FEFLO is a typical adaptive, edge-based finite element code for the solution of compressible and incompressible flows, which is primarily written in Fortran 77 and has previously been ported to vector, shared memory parallel and distributed memory parallel machines. Owing to the large size of FEFLO and the likelihood of human error in porting, as well as the desire for continued development within a single codebase, a specialized Python script, based on FParser (*Int. J. Comput. Sci. Eng.* 2009; **4**:296–305), was written to perform automated translation from the OpenMP-parallelized edge and point loops to GPU kernels implemented in CUDA, along with GPU memory management. The results of verification benchmarks and performance indicate that performances achieved by such a translator can rival those of codes rewritten by specialists. 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 14 October 2010; Revised 23 December 2010; Accepted 22 February 2011

KEY WORDS: graphics hardware; GPUs; Fortran; computational fluid dynamics

1. INTRODUCTION

At present, there is considerable interest in the application of graphics processing units (GPUs) to computational fluid dynamics (CFD). This interest is largely due to the tremendous increase in the performance of GPUs over the past few years. The latest Fermi architecture from The NVIDIA Tesla C2050 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 480 achieves more than 1.25 Teraflops of peak single-precision performance with a peak memory bandwidth of 177.4 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 speed-ups. Work in this direction began prior to the introduction of CUDA, when GPU programming was done via traditional graphics APIs such as OpenGL. An excellent and comprehensive survey of work done during this era is given by Owens *et al.* [4]. Much of the effort in running CFD codes on GPUs has been directed toward the case of solvers based on structured grids. These solvers are particularly amenable to GPU implementation due to their regular memory access pattern. Work in this area includes that

[*]Correspondence to: Rainald Löhner, Center for Computational Fluid Dynamics, Department of Computational and Data Sciences, M.S. 6A2, College of Science, George Mason University, Fairfax, VA 22030-4444, U.S.A.
[†]E-mail: rlohner@gmu.edu
[‡]Current address: Center for Reactive Flow and Dynamical Systems, Laboratory for Computational Physics and Fluid Dynamics, Naval Research Laboratory, Washington, DC 20375, U.S.A.

of Brandvik and Pullan [5–7], who have developed 2D and 3D Euler and Navier-Stokes solvers for GPUs, with support for multiple GPUs via MPI, and achieved an order of magnitude gain in the performance. Göddeke *et al.* [8] have implemented a multi-level, globally unstructured, locally structured, Navier–Stokes solver. LeGresley *et al.* [9] have implemented a multi-block Euler solver for simulating a hypersonic vehicle configuration, while Cohen and Molemaker [10] have implemented a 3D finite volume Boussinesq code in double precision. Further work on regular grid solvers includes that of Phillips *et al.* [11], who have developed a 2D compressible Euler solver on a cluster of GPUs. Jacobsen *et al*. [12, 13] have implemented a 3D incompressible Navier–Stokes solver for GPU clusters, while Antoniou *et al.* [14] have implemented high-order WENO finite difference methods on multi-GPU systems. Jespersen [15] has implemented a Jacobi solver for OVERFLOW, a Reynolds-averaged Navier–Stokes solver which uses multiple overset structured grids. Patnaik and Obenschain [16] have studied the application of GPUs to the FCT-based structured grid solver FAST3D-CT, with a focus on reducing power requirements.

There has also been interest in running unstructured grid-based CFD solvers on GPUs. Achieving good performance for such solvers is more difficult due to their data-dependent and irregular memory access patterns. Work in this area includes that of Klöckner *et al.* [17], who have implemented discontinuous Galerkin methods on unstructured grids. Markall *et al.* [18] developed a form compiler to automatically generate finite element code from an abstract representation in the Unified Form Language (UFL), which will be used to generate GPU-optimized CUDA code for the CFD code fluidity. In the previous work [19] the authors presented results for a cell-centered finite volume Euler solver running on a Tesla 10 series card, which achieved a nearly 10× speed-up factor over an OpenMP-parallelized CPU code running on a quad-core Intel CPU. Similar results, for a 2D edge-based solver, were also presented by Dahm and Fidkowski [20]. Asouti *et al.* [21, 22] have also implemented a vertex-centered finite volume code for unstructured grids on GPUs.

While most of the work done 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 and obtain a similar performance gain. We consider the solution of this task to be of general interest, as many CFD codes that have been developed by teams over sometimes decades currently face the same basic questions when pondering a port to GPU-hardware. The code under consideration is FEFLO, which is a typical adaptive, edge-based finite element code for the solution of compressible and incompressible flows. It consists (like 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 [23], transonic flow, store separation [24] and blast–structure interaction [25, 26], as well as incompressible flow applications including free-surface hydrodynamics [27], dispersion [28] and patient-based hemodynamics [29]. Codes with similar loop structures are those described in [30–34].

In the previous work [35], the authors introduced the concept of a Python script capable of automatically generating a version of FEFLO in which all relevant portions are run on the GPU using CUDA with minimal data transfer to or from the CPU throughout the course of a run. In this paper, the performance issues which guided many of the decisions made in writing the translator and porting FEFLO are first reviewed in Section 2. The main features of the Python script are then described in Section 3. The status of FEFLO, particularly with respect to parallelism, prior to its translation into GPU code is then described in Section 4, where the focus is on the performance critical point and edge loops. In Section 5, the overall structure of FEFLO and flow of GPU data is described. Finally, the results of runs used for validation purposes and performance tests are given in Section 6.

2. GPU PERFORMANCE ISSUES

In order to obtain high performance on NVIDIA GPUs there are a number of criteria that must be met. Three of the most important criteria are highlighted in the following.

DOI: 10.1002/fld

2.1. Fine-grained parallelism

Modern multi-core CPUs typically have 4–8 cores available, for which coarse-grained parallelism can be sufficient due to the small number of threads needed to achieve a speed-up. In the context of CFD solvers, such parallelism is often implemented via domain decomposition, and in fact, running multiple MPI ranks on a single multi-core CPU can lead to satisfactory performance scaling.

For a number of reasons GPUs work best for codes which expose fine-grained parallelism. In contrast to CPUs, GPUs could be described as many-core processors, with the latest NVIDIA Tesla C2050 GPU providing 448 CUDA cores, and the Fermi architecture supporting up to 512 cores [1]. In terms of CFD solvers this implies parallelizing loops over individual points, edges, faces, elements or cells. This alone requires a large number of active threads to fully populate the available CUDA cores. Furthermore, even full population of the CUDA cores with threads is insufficient for achieving full performance on NVIDIA GPUs. As pointed out by Bell and Garland [36], this is due to the need for GPUs to hide memory latency via multi-threading: once the warp (a group of 32 threads executed in parallel by the hardware [1]) currently being executed idles due to a memory transaction, the GPU thread scheduler will attempt to continue the execution of another warp in order to avoid the hardware idling. Using this approach, the more warps that are available for execution, the better the GPU can hide memory latency, and thus achieve high performance.

In addition, the requisite number of threads will only increase across future generations, as evidenced by the rate at which the number of CUDA cores has increased (nearly a factor of two) across successive past and present generations of Tesla hardware [1]. To scale transparently across future generations of hardware, codes should expose as much parallelism as possible [1, 37].

A further reason for fine-grained parallelism is implied by the memory access requirements of GPUs, which will be elaborated upon in Section 2.3. Generally speaking, in order to read global memory (the main off-chip memory space or RAM of a GPU [1, Section 2.3]) most efficiently, (half-)warps (architectures prior to the Fermi architecture serviced memory transactions on a half-warp basis, while Fermi services memory transactions on a per-warp basis) of threads are expected to read in contiguous segments of memory. In comparison to a coarse-grained approach, where each thread processes a large subarray, such a requirement is more readily fulfilled using fine-grained parallelism with consecutive threads processing individual elements of arrays.

The practical implication of the requirement to achieve large fine-grained parallelism is that loops have to be long and do (if possible) the same operations for each loop item. For those young enough to still remember, this same requirement was placed upon developers when vector machines (CDC-205, Cray, NEC) appeared in the mid-1980s.

2.2. CPU–GPU data transfer

Data transfer between the CPU and GPU should be minimized, due to the order of magnitude lower memory bandwidth across the system bus as compared with internal GPU memory bandwidth [37, Section 3.1]. This means that the straightforward ‘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 would in most cases severely restrict any performance gain possible using GPUs.

Ignoring this issue would have greatly simplified the task of porting FEFLO. In fact, this was the initial approach taken. As a test, a Roe solver subroutine of FEFLO (which for certain run configurations is responsible for a large portion of the run time) was rewritten manually in CUDA. Every time step, the arrays used by this subroutine were passed to the GPU. The results were disappointing: while the CUDA kernels in isolation performed very efficiently in comparison with the corresponding loops running on the CPU, no gain whatsoever was achieved once the cost of data transfer between the GPU and CPU was factored in. Summarizing, if the ‘accelerator design’ approach had been taken, even taking advantage of hardware features such as using pinned memory and asynchronous data transfers, the performance gain would have been minimal in comparison with what could be achieved without the data transfer. Instead, arrays processed by CUDA kernels should be kept as much as possible on the GPU. One of the main features—and the most complicated task of the Python script—is to carefully track how each array is used throughout

DOI: 10.1002/fld

the code, and enforce consistency throughout the entire subroutine call graph. An array which is used in a parallel loop in one subroutine is prohibited from being accessed freely in serial code. Of course, exceptions will need to be made in any large-scale, general-purpose code, and mechanisms were implemented to enable access on both the CPU and GPU when necessary.

2.3. Coalesced memory access

The third issue highlighted is that of making the most effective use of memory bandwidth within the GPU by achieving coalesced memory access. This issue is considered to be 'perhaps the single most important performance consideration in programming for the CUDA architecture' [37, p. 20]. This issue shares the purpose of maximizing cache hits on CPUs, i.e. reading memory in low-latency cache in order to reduce slower off-chip memory access. Since GPUs service memory transactions in 32, 64 or 128 byte segments, coalesced memory access minimizes the number and size of segments required to fully service a particular global memory transaction for a given (half-)warp. Given a particular array access in a CUDA kernel, coalesced memory access means that the threads of a (half-)warp access a contiguous, aligned segment of memory (see the CUDA Programming Guide [1] for full technical details). If the threads of (half-)warp were to access memory in completely disparate locations, then each thread would require a separate segment to be read from or written to in global memory, leading to an enormous degradation in the performance [37, Figure 3.9]. Hardware up until the latest Fermi architecture simply discarded unused portions of these segments, while the latest generation of GPUs introduces a cache which can retain previously read segments on-chip, coalescing remains of the utmost importance for achieving high performance.

2.4. Other performance issues

Shared memory is a feature of NVIDIA GPUs which can be used to avoid redundant global memory access between threads within a thread block, which must be explicitly managed in software by the programmer [1]. Owing to its low latency, shared memory can provide substantial performance benefits; however, it is not applicable to FEFLO due to the fact that there is no redundant memory access within each parallel loop, as described in Section 4.2.1. An additional hindrance to employing shared memory is that the data-dependent memory access patterns used in FEFLO make it difficult if not impossible to explicitly manage shared memory in software. Therefore, the code generated by the current version of the Python script does not manage shared memory. Utilizing shared memory would however be of utmost importance for codes with a fixed memory, data-independent memory access pattern, such as structured grid solvers.

The texture cache is also not utilized, a hardware feature which has been shown to improve the performance for unstructured memory access patterns in the work on sparse matrix–vector multiplication solvers of Bell and Garland [36]. Texture memory is read-only, and thus could only account for half of the potentially uncoalesced memory access encountered in the edge loops of FEFLO, since such loops also involve unstructured writes to global memory. Instead, the more general-purpose L1 and L2 cache space of the latest Fermi architecture [2], which shares resources with the shared memory space, will be relied upon to alleviate the performance hit due to any remaining uncoalesced memory access.

3. THE PYTHON SCRIPT

While GPUs offer high performance, arbitrary code cannot simply be recompiled and expected to run efficiently on GPUs. Instead GPUs require that code be rewritten using interfaces such as CUDA [1] or OpenCL [3]. While writing new code in CUDA or OpenCL is not intrinsically difficult, the manual translation of a code on the scale of FEFLO necessarily introduces a large number of bugs, and would involve an overwhelming amount of tedious work. There are more than 10 000 parallel loops in FEFLO to be translated. Translating loops is actually relatively simple compared with handling the intricate bookkeeping required to properly track arrays across the

DOI: 10.1002/fld

subroutine call graph in order to ensure their consistent placement into either the GPU or CPU memory space, deducing sub-array semantics, and correctly calling data transfer subroutines when necessary. Just translating the code in its current state would be a formidable task. Furthermore, FEFLO remains under continuous development, and therefore a manual approach would result in a perpetual translation process and necessitate the creation of two separate codebases.

Another issue is that of choosing the right interface to access GPU hardware. CUDA has the disadvantage of being proprietary, while OpenCL is an open standard. However, CUDA is more mature and has extensive C++ support. Both these factors have contributed to the availability of libraries such as Thrust [38], which are used extensively in the GPU version of FEFLO. A similar library, with such a breadth of generic, data parallel algorithms, does not appear to be available for OpenCL. In addition, even if a GPU code is written in OpenCL, it still needs to be optimized to satisfy the performance requirements of different hardwares. Therefore, an approach is needed which can be rapidly adapted to new combinations of hardware and software.

Owing to these issues, an automatic translator is used in this work, which avoids most issues plaguing manual translation, and is flexible enough to satisfy future hardware and software requirements. Automatic approaches often compromise on the performance, which contradicts the ultimate purpose of porting codes such as FEFLO to GPUs. This is not the case here, since the translator was specialized to generate the same code that would be written via a manual translation. As a result, the purpose of this script is not to automatically parallelize any arbitrary code, or even translate any arbitrary OpenMP-parallelized code. Rather its purpose is to enable the main developers of FEFLO to continue development in Fortran, with certain additional necessary restrictions. Most of the script, however, is general purpose, and as a side benefit, it is entirely conceivable that this translator could be adapted to also translate other codes automatically by adapting to the coding style used. *At this point it must be stressed again that the key requirement for the success of such translators is the consistency and uniformity in coding style of the legacy code*.

Currently, the translator has complete support for CUDA, with less complete support for CUDA Fortran and OpenCL. CUDA, while freely available and very capable with a growing ecosystem of useful libraries and supporting software, it is still proprietary. In the long term, the script will be adapted to emit GPU kernels implemented in OpenCL, optimized for future hardware platforms, by simply adding functions to the Python classes used to represent and transform the decomposed Fortran code. The main impediment in the short term to a complete OpenCL implementation is the lack of availability of a library similar to Thrust, which itself only provides a CUDA and OpenMP backend due to the necessity of C++ support, which OpenCL lacks.

There were a few crucial factors which enabled such an automatic approach. The first is the availability of FParser, a component of the F2PY package [39] which allows for Fortran source code parsing in the highly productive programming language Python. The second is the strict and uniform coding style used in FEFLO, due to most code having been either originally developed, rewritten, or cleaned up by its main developer, in order to follow FEFLO's strict coding standards. While the script is capable of automatically generating optimal code from a very large class of OpenMP loops which already exhibit fine-grained parallelism, there are many OpenMP loops in FEFLO which do not explicitly exhibit the fine-grained parallelism needed to run efficiently on GPUs. Automatically restructuring such loops essentially amounts to auto-parallelization, but the strict, uniform coding style in FEFLO allowed for specialized Python code to be written to automatically handles this.

In summary, the resulting translator is just a few thousand lines of Python code, which automatically:

- Converts simple OpenMP loops into 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 across the subroutine call graph.
- Tracks physical array sizes of subarrays across subroutine calls.
- Uses a transposed array layout appropriate for meeting coalescing requirements.

DOI: 10.1002/fld

- Handles GPU array I/O and memory transfer.
- Allows 'difficult' subroutines to be ignored and left on the CPU, or overridden with custom implementations, usually based on Thrust [38] and
- Integrates with pure CPU code, including MPI code.

3.1. Alternative translators

A number of alternative automatic translators and compilers were considered.

The PGI accelerator programming model uses compiler directives to parallelize Fortran and C code for execution on the GPU. If explicit control over the code's execution on the GPU, then code must be written manually, using PGI's CUDA Fortran compiler [40]. The obvious advantage of this approach is that it would be relatively automatic in that only compiler directives would need to be specified. A disadvantage is that it offers less control, as it only implicitly specifies the parallel decomposition of the computation performed by a parallel loop. In addition, it would be preferred to reuse, as much as possible, the existing OpenMP directives already in place, since the introduction of another set of directives can be error prone.

The work of Lee *et al.* [41] also describes a compiler framework which also automatically converts OpenMP loops into CUDA kernels. Their work includes more general and much more advanced automatic optimization techniques than those present in the translator used in this work. However, most loops in FEFLO can be either directly translated, restructured automatically using specialized Python code, or overriden with calls to the Thrust library. Another crucial issue is that the compiler used by Lee *et al.* only reduces memory transfer based on analyzing array names. In contrast, the Python translator used in this work tracks arrays across subroutine calls to ensure that arrays are kept on the GPU throughout an entire run, with exceptions made only when absolutely necessary, for example to achieve IO or make use of intentionally serial code. Furthermore, the presented compiler does not appear to support Fortran.

The F2C-ACC translator developed by Govett [42] is another option for automatically translating Fortran code based on custom directives, similar to the Python script used here. There are a few reasons that it could not be used for FEFLO. The first is that it does not appear to be possible to pass GPU arrays across subroutines, which is an essential feature of the Python translator used for FEFLO. Additionally, it lacks support for IO. Finally (and this is not a strictly scientific argument), the developer of the translator used in this work, simply prefers the productivity of programming in Python, for the purpose of customizing the translator to FEFLO, in comparison to the C language used in F2C-ACC.

4. PARALLELISM IN FEFLO

FEFLO is a typical adaptive, edge-based finite element code for the solution of compressible and incompressible flows. It is based on linear elements (tetrahedra, i.e. unknowns stored at nodes) for geometric flexibility and adaptivity. The edge-based formulation is employed in order to reduce indirect addressing, and facilitate the implementation of limiting and upwinding [43]. The code has been ported to vector, shared memory parallel (via OpenMP [44]) and distributed memory parallel (via MPI [45]) machines.

In FEFLO, the vast majority of parallel loops fall into two categories. The first category consists of loops without any indirect addressing, or with only gather operations to achieve a result that is stored in arrays that are commensurate with the loop index. These loops can be over points, faces, edges, elements, etc. In the sequel, we will denote these as point loops. These point loops, c.f. Section 4.1, are parallelized using OpenMP in a way that exhibits fine-grained parallelism, which enables their direct translation into GPU kernels. The second category of loops consists of loops where fluxes, source-terms or other right-hand side (RHS) terms are built using a scatter-add process. These loops can be over faces, edges, elements, etc. As most of these occur over edges, we will denote these as edge-loops. The edge-loops, Section 4.2, which are also parallelized using OpenMP, but in a way that exhibits domain decomposition-like coarse-grained parallelism. One of the main ways in which the Python script is specialized for FEFLO is that it

DOI: 10.1002/fld

automatically recognizes these edge loops and restructures them using specialized Python code, in order to expose fine-grained parallelism in the edge loops. Additional loop types present in FEFLO include those related to random number generation, index operations and array compaction, c.f. Section 4.3.

FEFLO is also parallelized to simultaneously take advantage of distributed memory parallelism on CPU clusters. Fortunately, the existing MPI calls are expected to be able to be used unmodified, but the application of FEFLO to GPU clusters is left as future work, as will be described in Section 7.

4.1. Point loops

The point loops in FEFLO are directly translated into CUDA code, since they already explicitly exhibit fine-grained parallelism in the original Fortran code. Loops such as these (they could also be element, edge, face, etc.), are treated automatically by the translator, in which case, the parallelization of the CUDA kernel exactly mirrors that of the OpenMP parallel loop, with each CUDA thread corresponding to a loop iteration, and OpenMP private items treated as per-CUDA-thread variables which are stored in on-chip registers. Consider a loop from the `locfct` subroutine in Listing 1 taken from the flux-corrected transport module of FEFLO, used for the benchmarks in Section 6.

Listing 1: An OpenMP point loop.

```
c
c     ------multiply by the mass-matrix
c
!$omp  parallel do private(ip,cmmat)
!cdir  inner
!cdir  concur
c
          do 1600 ip=npami,npamx
          cmmat       =mmatm(ip)
          delun(1,ip)=cmmat*delun(1,ip)
          delun(2,ip)=cmmat*delun(2,ip)
          delun(3,ip)=cmmat*delun(3,ip)
          delun(4,ip)=cmmat*delun(4,ip)
          delun(5,ip)=cmmat*delun(5,ip)
 1600     continue
```

Once processed by the translator, this OpenMP-parallelized loop is replaced with a subroutine call, as shown in Listing 2.

Listing 2: The subroutine call which replaces the OpenMP point loop.

```
    !     ------multiply by the mass-matrix
    !           ------ Directive removed
    call locfct_loop2(delun,mmatm,npami,npamx)
```

This subroutine call is to a C++ function which invokes a CUDA kernel, as shown in Listing 3. The CUDA kernel is invoked with a number of threads greater than or equal to the number of loop iterations, rounded up to be a multiple of the thread block size, which is a user-specified parameter in the Python script.

Listing 3: The C++ function which invokes the CUDA kernel.

```
extern "C"
void locfct_loop2_(da_double2* delun, da_double1* mmatm,
                   int* npami, int* npamx)
{
  dim3 dimGrid = dim3(round_up((*npamx)-((*npami))+1), 1, 1);
  dim3 dimBlock = dim3(256,1,1);
  locfct_loop2<<<dimGrid, dimBlock>>>
    (delun->a, delun->shape[1], mmatm->a, *npami, *npamx);
}
```


DOI: 10.1002/fld

The invoked CUDA kernel, shown in Listing 4, is essentially the inner body of the OpenMP loop from Listing 1. In this example the transposed multi-dimensional array layout is apparent, which is used to ensure coalesced memory access into `delun`. This ensures that the array is accessed with a unit stride, avoiding the serious performance loss that would occur using the original Fortran array layout, c.f. Section 2.3 and [37, Section 3.2.1.4]. The physical array size of `delun` is also used to index into the array, which is tracked at run time, since it may differ from the size specified in its declaration at the beginning of the `locfct` subroutine. The result of Fortran's 1-based indexing is also apparent, which is not translated into 0-based indexing in order to maintain consistency with the many 1-based connectivity and index arrays which appear throughout FEFLO. Apart from the unsimplified index arithmetic, which should be taken care of by the CUDA compiler optimizer, this code is essentially the same code that would be produced with manual translation.

Listing 4: Point loop CUDA kernel.

```
__global__
void locfct_loop2(double* delun, int delun_s1, double* mmatm,
                  int npami, int npamx)
{
  double cmmat;

  const unsigned int ip=blockDim.x*blockIdx.x+threadIdx.x+npami;
  if(ip > npamx) return;

  cmmat=mmatm[ip-1];
  delun[ip-1+delun_s1*(1-1)]=cmmat*delun[ip-1+delun_s1*(1-1)];
  delun[ip-1+delun_s1*(2-1)]=cmmat*delun[ip-1+delun_s1*(2-1)];
  delun[ip-1+delun_s1*(3-1)]=cmmat*delun[ip-1+delun_s1*(3-1)];
  delun[ip-1+delun_s1*(4-1)]=cmmat*delun[ip-1+delun_s1*(4-1)];
  delun[ip-1+delun_s1*(5-1)]=cmmat*delun[ip-1+delun_s1*(5-1)];
}
```

4.2. Edge loops

The edge loops in FEFLO are not as easy to translate into CUDA code since they are written in a way that does not explicitly exhibit fine-grained parallelism. However, such parallelism is implicit and can be exposed automatically by the translator. This relies, for example, on the consistent naming of loop variables, and the uniform loop structure. In Section 4.2.3, the CUDA translation will be described, after first showing the vectorization and parallelization of the CPU code. The code in Listing 5 shows a generic, serial edge loop.

Listing 5: Basic edge loop.

```
      do 1600 iedge=1,nedge
      ipoi1=lnoed(1,iedge)
      ipoi2=lnoed(2,iedge)
      redge=geoed(   iedge)*(unkno(ipoi2)-unkno(ipoi1))
      rhspo(ipoi1)=rhspo(ipoi1)+redge
      rhspo(ipoi2)=rhspo(ipoi2)-redge
1600  continue
```

The operations to be performed can be grouped into the following three major steps:

(a) Gather point information into the edge.
(b) Perform the required mathematical operations at the edge level (for the loop shown—a Laplacian—this is a simple multiplication and subtraction; for an Euler solver the approximate Riemann solver and the equation of state would be substituted in).
(c) Scatter-add the edge RHS to the assembled point RHS.

This simple loop already illustrates the type of data-dependent, indirect memory access which occurs in the majority of edge loops present in FEFLO (and any similar CFD code for that matter). For CPUs, cache-misses are likely to occur if the nodes within each edge are widely spaced in memory, necessitating slower off-chip memory access. To minimize such cache-misses, algorithms

DOI: 10.1002/fld

based on bandwidth minimization are typically employed [43, 46]. The situation for GPUs is different: for such loops meeting coalescing requirements replaces avoiding cache misses as being the main goal of optimizing memory access patterns, c.f. Section 2.3.

4.2.1. Edge loops on vector processors. The possibility of memory contention inhibits vectorization, which can occur if two of the nodes within the same group of edges being processed by vector registers coincide. If the edges can be reordered into groups such that within each group none of the nodes are accessed more than once, vectorization can be enforced using compiler directives. Denoting the grouping array by `edpas`, the vectorized edge loop is shown in Listing 6.

Listing 6: Vectorized edge loop.

```
      do 1400 ipass=1,npass
      nedg0=edpas(ipass)+1
      nedg1=edpas(ipass+1)
!dir$ ivdep                          ! Ignore Vector Dependencies
       do 1600 iedge=nedg0,nedg1
       ipoi1=lnoed(1,iedge)
       ipoi2=lnoed(2,iedge)
       redge=geoed(   iedge)*(unkno(ipoi2)-unkno(ipoi1))
       rhspo(ipoi1)=rhspo(ipoi1)+redge
       rhspo(ipoi2)=rhspo(ipoi2)-redge
 1600  continue
 1400  continue
```

Algorithms that group the edges into such non-conflicting groups fall into the category of coloring schemes [43].

4.2.2. Edge loops on multicore processors. In order to obtain optimal performance on shared multicore processors via OpenMP, care has to be taken not to increase cache misses while allowing for parallelization and vectorization. The compromise typically chosen is similar to that of domain decomposition: assign to each processor groups of edges that work on the same group of points [44]. In order to avoid the explicit declaration of local and shared variables, a sub-subroutine technique is used [43]. The result of this is shown in Listing 7.

Listing 7: The vectorized edge loop parallelized using OpenMP.

```
c
      do 1000 imacg=1,npasg,nproc
      imac0=           imacg
      imac1=min(npasg,imac0+nproc-1)
!$omp parallel do private(ipasg)     ! Parallelization directive
       do 1200 ipasg=imac0,imac1
       call loop2p(ipasg)
 1200  continue
 1000 continue
c
c   ---------------------------------------------------
c
      subroutine loop2p(ipasg)
      npas0=edpag(ipasg)+1
      npas1=edpag(ipasg+1)
      do 1400 ipass=npas0,npas1
      nedg0=edpas(ipass)+1
      nedg1=edpas(ipass+1)
!dir$ ivdep                          ! Ignore Vector Dependencies
       do 1600 iedge=nedg0,nedg1
       ipoi1=lnoed(1,iedge)
       ipoi2=lnoed(2,iedge)
       redge=geoed(   iedge)*(unkno(ipoi2)-unkno(ipoi1))
       rhspo(ipoi1)=rhspo(ipoi1)+redge
       rhspo(ipoi2)=rhspo(ipoi2)-redge
 1600  continue
 1400 continue
```


DOI: 10.1002/fld

For relatively simple Riemann solvers and simple equations of state, this type of loop structure works well. However, once more esoteric Riemann solvers and equations of state are required, one is forced to split the inner loop even further. In particular, for equations of state one may have a table look-up with many `if`-tests, and even parts that do not vectorize well. The usual recourse is to split the inner edge-loop even further, as shown in Listing 8.

Listing 8: Split vectorized edge loops.

```
c      inner loop, processed in groups
         do 1600 iedge=nedg0,nedg1,medgl
         iedg0=iedge-1
         iedg1=min(iedg0+medgl,nedg1)
         nedgl=iedg1-iedg0
c
c        transcribe variables for local edge-group
         call tranvariab(...)
c
c        obtain fluxes for local edge-group
         call apprxriem(....)
c
c        obtain RHS for local edge-group
         call rhsvisc(...)
c
!dir$ ivdep                          ! Ignore Vector Dependencies
         do 1800 iedgl=1,nedgl
         ipoi1=lnoel(1,iedgl)
         ipoi2=lnoel(2,iedgl)
         rhspo(ipoi1)=rhspo(ipoi1)+redgl(iedgl)
         rhspo(ipoi2)=rhspo(ipoi2)-redgl(iedgl)
 1800   continue
 1600  continue
```

Note that the transcription of variables, the fluxes and the RHS obtained at the local edge-group level may involve many branches (`if`-statements), but that care has been taken that these innermost loops (which reside in separate subroutines) are, if at all possible, vectorized.

4.2.3. Edge loops on GPUs. The parallel edge loop in Listing 7 is implemented using coarse-grained parallelism via domain decomposition. Such parallelization is inappropriate for GPUs, and the edge loops cannot be directly parallelized from this OpenMP code. While not exactly equivalent, analogies can be drawn between the requirements for vectorization and GPU parallelization, for example:

- Both vectorized loops and GPUs perform more efficiently without branching, and FEFLO has been coded to avoid branching in inner loops. GPUs do allow some amount of branching, but branching results in serialized execution, if different branches are taken by the threads of a warp [37, Chapter 6].
- Vectorized loops are required to be tight inner loops, consisting primarily of arithmetic operations and intrinsic functions, which leads to fine-grained parallelism with minimal branching.

Because of these similarities, the vectorized inner loops are ideal candidates for GPU parallelization, as they contain implicit fine-grained parallelism. Edge loops in FEFLO are consistently indicated by naming the loop variable of the outermost loop `ipasg`. This indicates to the translator to restructure this loop to propagate coarse-grained parallelism in the loop over `ipasg` inwards toward the vectorized loop over `iedge`. Issues that are accounted for throughout this process include:

- The lack of an OpenMP private items list for the inner loop, specifying per-thread variables for the inner loop, information which instead must be deduced via inner-loop-variable dependency.
- The presence of serial reductions within the inner loop, for which custom reduction directives are needed.
- The necessity to propagate parallelism through subroutine calls, as in the split vectorized edge loop shown in Listing 8.

DOI: 10.1002/fld

- Variations in code structure due to flow control statements and
- The absence of vectorization directives in the case of edge-to-edge operations indicating that the loop contains fine-grained parallelism.

The resulting GPU code is analogous to the example shown in Section 4.1, with the inner vectorized edge loop replaced with a call to a CUDA kernel.

4.3. Other loops

Some subroutines in FEFLO contain loops which cannot be automatically translated to the CPU. Fortunately such cases are few, and tend to be general-purpose algorithms which are well studied on the GPU, with implementations available in the library Thrust [38]:

- Reduction (thrust::reduce).
- Indexes of Extrema (`thrust::min_element`, `thrust::max_element`).
- Array Compression/compaction and index gathering (`thrust::copy_if`).
- Prefix sum/scan (`thrust::inclusive_scan`).
- Random number generation (`thrust::uniform_real_distribution`) and
- Histogram Computation (`thrust::sort`, `thrust::upper_bound`, `thrust::adjacent_difference`).

The subroutines containing these algorithms are automatically replaced with manually implemented lightweight wrappers which call the corresponding Thrust function. A benefit of this approach is that these algorithms are all non-trivial to implement, and FEFLO will automatically inherit from Thrust any future improvements to these algorithms and fine-tuning made to meet evolving performance requirements.

5. FEFLO STRUCTURE AND GPU DATA FLOW

The main steps during a FEFLO run are similar to that of any other field solver:

(a) Field (points, elements, unknowns, boundary conditions, etc.) and control (nr. of time steps, diagnostics output, etc.) data is read in.
(b) Data are renumbered for the particular machine architecture.
(c) Derived data required for the flow solver (edges, faces, geometry parameters, etc.) are computed.
(d) The loop over the time steps is performed; within each step the solution is advanced in time;
(e) Restart data are outputted.

As far as the flow of information between the CPU and GPU, it proceeds as follows:

(a) Field data is read into the CPU and automatically passed to the GPU; control data always resides on the CPU.
(b) Before data renumbering (which is largely scalar) the required data are passed to the CPU (`!$gpu cpu(...)`); renumbering is then carried out on the CPU; thereafter, the renumbered data are transferred back to the GPU (`!$gpu end cpu(...)`).
(c) The computation of the missing data required for the flow solver is carried out in the GPU.
(d) Within the time-stepping loop the solution is advanced in time entirely on the GPU.
(e) Field restart data are passed automatically to the CPU and outputted.

6. RESULTS

The GPU version of FEFLO was tested for a variety of benchmark cases. Three different compressible flow cases were considered. The first two cases are drawn from blast applications, while the

DOI: 10.1002/fld

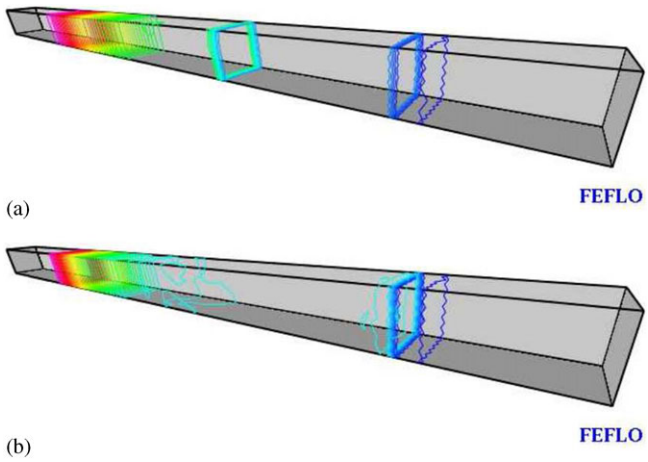

Figure 1. Sod shock tube: densities and pressures. (a) density; (b) pressure.

Table I. Blast in room (60 time steps).

Nelem	CPU/GPU	Mvecl	Time (s)
1.0 M	Core i7 940 (1)	32	35
1.0 M	Core i7 940 (2)	32	25
1.0 M	Core i7 940 (4)	32	18
1.0 M	Core i7 940 (8)	32	17
1.0 M	GTX 285	51200	10

third is from steady-state aerodynamics. The other two cases are incompressible flows, the first of which is a pipe flow, while the last is a two-phase dam break problem.

The system used for the timing studies consisted of an Intel Core i7 940 CPU and an NVIDIA GTX 285 GPU. The Intel Core i7 940 CPU is a quad-core CPU which supports up to eight threads via hyper-threading and features a peak memory bandwidth of 25.6 GB/s. The NVIDIA GTX 285 is a GeForce 200 series GPU (same architecture as the Tesla 10 series), which contains 240 cores (30 multiprocessors) and a peak memory bandwidth of 159.0 GB/s.

In each of the subsequent tables `nelem` denotes the number of elements (mesh size), CPU/GPU the hardware used, `mvecl` the maximum vector length allowed when reordering elements and edges to avoid memory contention, and Time the total run time in seconds.

6.1. Sod shock tube

The first example is the classic Sod Shock Tube. A diaphragm that separates two states of compressible flow bursts at time $T = 0.0$, resulting in a shock, a contact discontinuity and an expansion wave. The geometry, together with the solution, can be discerned from Figure 1. The compressible Euler equations are solved using the edge-based FEM-FCT technique [43, 47, 48].

This case was run for verification purposes only. Figure 1 shows overlaid contour lines at the same time for the CPU and GPU runs. Note that the results are essentially the same, with a few very minor differences (as would be expected due to minor differences in floating point arithmetic).

6.2. Blast in room

The second example is the compressible flow resulting from a blast in a room (Table I). The geometry, together with the solution, can be discerned from Figure 2.

The compressible Euler equations are solved using an edge-based FEM-FCT technique [43, 47, 48]. The initialization was performed by interpolating the results of a very detailed

DOI: 10.1002/fld

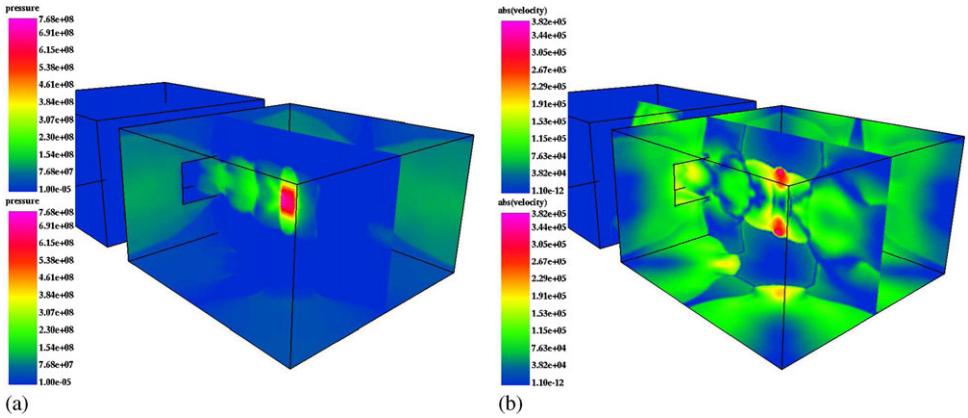

Figure 2. Blast in room: pressures and velocities. (a) pressure; (b) velocity.

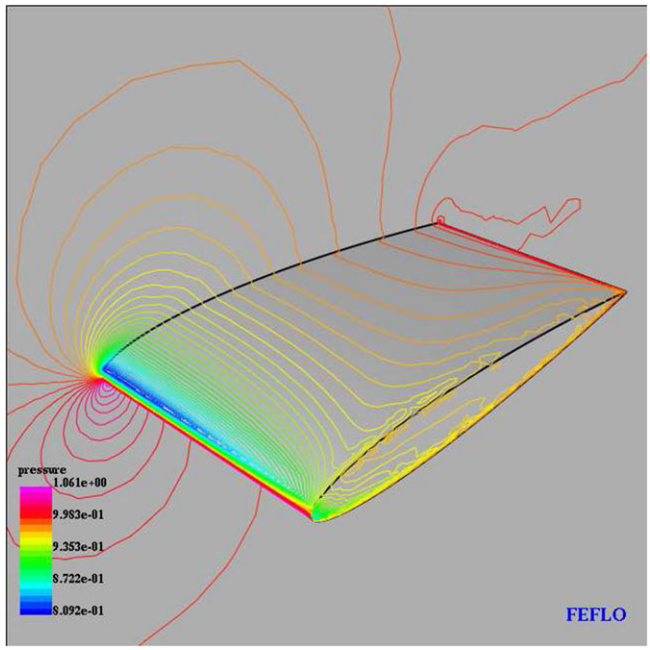

Figure 3. NACA0012: surface pressures.

1D (spherically symmetric) run. The timing studies were carried out with the following set of parameters:

- Compressible Euler.
- Ideal Gas EOS.
- Explicit FEM-FCT.
- Initialization From A 1D File.
- 1.0 million elements.
- Run for 60 steps.

6.3. NACA0012 wing

The third example is a classic aerodynamics example: steady inviscid compressible flow past a NACA0012 wing. The geometry, together with the solution, is shown in Figure 3. The incoming Mach number and angle of attack were set to $M_\infty = 0.3$ and $\alpha = 15^\circ$, respectively. The Euler

DOI: 10.1002/fld

Table II. NACA0012 (100 time steps).

Nelem	CPU/GPU	Mvecl	Time (s)
1.0 M	Core i7 940 (1)	32	184
1.0 M	Core i7 940 (2)	32	104
1.0 M	Core i7 940 (4)	32	60
1.0 M	Core i7 940 (8)	32	52
1.0 M	GTX 285	51 200	32

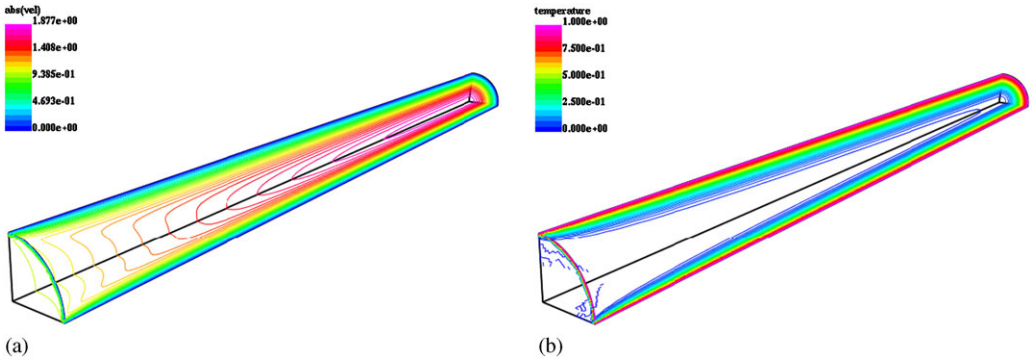

Figure 4. Pipe: velocities and temperature. (a) velocity; (b) temperature.

equations were integrated using a three-stage Runge–Kutta scheme with local timestepping and residual smoothing. The approximate Riemann solver was HLLC. The solution was obtained after 3000 steps (Table II).

The timing studies were carried out with the following set of parameters:

- Compressible Euler.
- Ideal Gas EOS.
- Explicit RK3, HLLC, nlimi=0.
- Steady state.
- Local timestepping.
- Residual damping.
- Ma=2, AOA = 15°.
- 1.0 million elements.
- Run for 100 steps.

6.4. *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 used a projection technique for pressure increments [43, 49]. 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 4 shows typical results obtained. The timing studies were carried out with the following set of parameters (Table III):

- Incompressible Navier–Stokes + Heat transfer.
- Advection: RK3, Roe, nlimi=2.
- Pressure: Poisson (Projection), DPCG.
- Steady state.
- Local timestepping.
- *Re*=100.

DOI: 10.1002/fld

Table III. Pipe.

Nelem	CPU/GPU	Mvecl	Time (s)
0.6 M	Core i7 940 (1)	32	300
0.6 M	Core i7 940 (2)	32	179
0.6 M	Core i7 940 (4)	32	126
0.6 M	Core i7 940 (8)	32	121
0.6 M	GTX 285	25 600	115

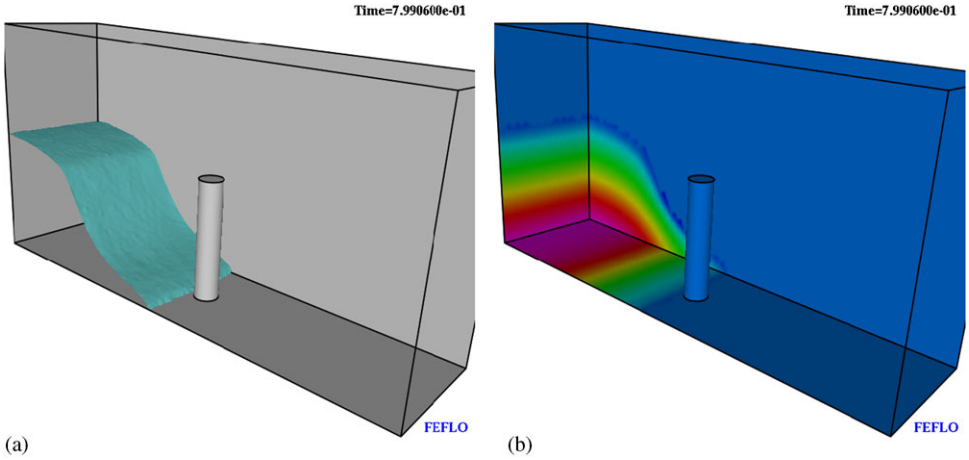

Figure 5. Dam break: free surface and pressure. (a) free surface; (b) pressure.

- 0.6 million elements.
- Run for 100 steps.

6.5. 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 [27]. Figure 5 shows typical results obtained. The timing studies were carried out with the following set of parameters (Table IV):

- Incompressible Navier–Stokes.
- VOF for free surface.
- Advection: Explicit RK3, Roe, nlimi=2.
- Pressure: Poisson (Projection), DPCG.
- Transient.
- 0.7 million elements.
- Run for 100 steps.

7. CONCLUSIONS AND OUTLOOK

The development of automatic techniques to port a substantial portion of FEFLO, a general-purpose legacy CFD code operating on unstructured grids, to run on GPUs, has been described. Owing to the large size of FEFLO and the likelihood of human error in porting, as well as the desire for continued development within a single codebase, a specialized Python script, based on FParser, was written to perform automated translation from the OpenMP-parallelized edge and point loops

DOI: 10.1002/fld

Table IV. Dam break.

Nelem	CPU/GPU	Mvecl	Time (s)
0.76 M	Core i7 940 (1)	32	93
0.76 M	Core i7 940 (2)	32	58
0.76 M	Core i7 940 (4)	32	42
0.76 M	Core i7 940 (8)	32	42
0.76 M	GTX 285	51 200	42

to GPU kernels implemented in CUDA, along with GPU memory management, while integrating with the existing framework for distributed memory parallelism via MPI.

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

- The ideal gas, explicit compressible flow solvers (FCT, TVD, approximate Riemann solvers, limiters).
- The projection schemes for incompressible flow with explicit advection, implicit viscous terms and pressure Laplacian.
- The explicit advection, implicit conductivity/diffusivity integrators of temperature and/or concentration/
- Some simple turbulence models (e.g. Smagorinsky).
- Some of the diagnostic output options (global conservation sums, station time history points).
- Subroutines required throughout the code/solvers to apply boundary conditions for periodic boundaries, overlapping grids, embedded surfaces and immersed bodies.

The results of verification benchmarks and performance tests were presented. In double precision, speed-up factors of up to 1.7x were obtained for the compressible flow solvers on the Geforce 285 GTX over the quad-core Intel Core i7 940. These speed-ups are of the same order of magnitude to those obtained from a small, hand-coded Finite Volume pilot code [50]. Given the relative ease of handling GPU computing in this way, this is viewed as very successful. For incompressible flow solvers, which are dominated by the conjugate gradient solvers of the pressure-Poisson equation, there is only a minor performance gain. This is due to the incompressible flow solver being almost completely memory bandwidth bound. While FEFLO has a variety of optimized numbering grid numbering schemes optimized for CPUs, there is currently no option which is optimized for GPU coalescing requirements. Therefore, future work will address this issue by employing numbering schemes tailored to meet GPU coalescing requirements. Moreover, the GPU considered in this work has limited double precision performance, and future work will employ the newer Fermi architecture, which has full double precision performance.

The translator approach presented here should be of general interest, as how best to run on GPUs is being presently explored for many so-called legacy codes.

Further developments will center on porting and debugging more FEFLO-options on the GPU, using the Python script within a multi-GPU environment, and developing optimal renumbering techniques for the GPU.

REFERENCES

1. NVIDIA Corporation. *NVIDIA CUDA 3.1 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. *Journal of 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. *The 46th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, NV, AIAA-2008-607, January 2008.

DOI: 10.1002/fld

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*, Orlando, FL, 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 and Simulation*, Leipzig, Germany, 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 CFD*, Moffet Field, CA, 2009.
11. Phillips EH, Zhang Y, Davis RL, Owens JD. Rapid aerodynamic performance prediction on a cluster of graphics processing units. *The 47th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, Reno, NV, 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. *The 47th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, Reno, NV, AIAA 2009-758, January 2010.
13. Jacobsen D, Thibault J, Senocak I. An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. *The 48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, Orlando, FL, 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. *The 48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, Orlando, FL, 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. *The 48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, Orlando, FL, AIAA-2010-524, January 2010.
17. Klöckner A, Warburton T, Bridge J, Hesthaven JS. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics* 2009; **228**:7863–7882.
18. 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*, Amsterdam, The Netherlands, June 2010.
19. 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; DOI: 10.1002/fld.2254.
20. Dahm JPS, Fidkowski KJ. Employing coprocessors to accelerate numerical solutions to the Euler equations, 2009. Available from: http://www.johanndahm.com/research.php.
21. 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* 2010; DOI: 10.1002/fld.2352.
22. 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.
23. Ramamurti R, Munday D, Gutmark E, Liu J, Kailasanath K, Löhner R. Large-eddy simulations of a supersonic jet and its near-field acoustic properties. *AIAA Journal* 2009; **8**(47):1849–1864.
24. Löhner R, Goldberg E, Baum JD, Luo H, 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*, Reno, NV, AIAA-1997-0166, January 1997.
25. Löhner R, Yang C, Pelessone D, Baum JD, Luo H, Charman C. A coupled fluid/structure modeling of shock interaction with a truck. *AIAA Aerospace Sciences Meeting*, Reno, NV, AIAA-1996-0795, January 1996.
26. Mestreau E, Löhner R, Pelessone D, Baum JD, Luo H, Charman C. A coupled CFD/CSD methodology for modeling weapon detonation and fragmentation. *AIAA Aerospace Sciences Meeting*, Reno, NV, AIAA-1999-0794, January 1999.
27. Yang C, Löhner R, Oñate E. Simulation of flows with violent free surface motion and moving objects using unstructured grids. *International Journal for Numerical Methods in Engineering* 2007; **53**:1315–1338.
28. Camelli F, Löhner R. Vles study of flow and dispersion patterns in heterogeneous urban areas. *AIAA Aerospace Sciences Meeting*, Reno, NV, AIAA-2006-1419, January 2006.
29. Löhner R, Putman CM, Appanaboyina S, Mut F, 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.
30. Mavriplis D. Three-dimensional unstructured multigrid for the Euler equations. AIAA-91-1549-CP, 1991.
31. Peraire J, Peiro J, Morgan K. A three-dimensional finite element multigrid solver for the Euler equations. *AIAA Aerospace Sciences Meeting*, Reno, NV, AIAA-92-0449, 1992.
32. Kaushik DK, Keyes DE, Anderson WK, Gropp WD, Smith BF. Achieving high sustained performance in an unstructured mesh cfd application. *Supercomputing 1999*. IEEE Computer Society: Silver Spring, MD, 1999.
33. Becker K, Heinrich R, Roll B, Galle M, Kroll N, Gerhold T, Schwamborn D, Aumann P, Barnewitz H, Franke M. Megaflow: parallel complete aircraft cfd. *Parallel Computing* 2001; **27**:415–440.
34. Ito Y, Nakahashi K, Togashi F. Some challenges of realistic flow simulations by unstructured grid cfd. *International Journal for Numerical Methods in Engineering* 2008; **43**:769–783.

35. Corrigan A, Camelli F, Löhner R, Mut F. Porting of an edge-based CFD solver to GPUs. *The 48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*. Orlando, FL, AIAA-2010-522, January 2010.
36. Bell N, Garland M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. *SC '09*: *Proceedings of the Conference on High Performance Computing Networking*, *Storage and Analysis*. ACM: New York, NY, U.S.A, 2009; 1–11.
37. NVIDIA Corporation. *NVIDIA CUDA 3.1 Best Practices Guide*, 2010.
38. Hoberock J, Bell N. *Thrust*: *a parallel template library*, 2009; Version 1.2.
39. Peterson P. F2PY: a tool for connecting Fortran and Python programs. *International Journal of Computational Science and Engineering* 2009; **4**:296–305.
40. The Portland Group. PGI Fortran & C Accelerator programming model, 2009.
41. Lee S, Min SJ, Eigenmann R. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *PPoPP '09*: *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2009.
42. Govett M. *F2C-ACC Code Translator*, 2009; Version 2.3.
43. Löhner R. *Applied CFD Techniques*. Wiley: New York, 2008.
44. 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.
45. Ramamurti R, Löhner R. A parallel implicit incompressible flow solver using unstructured meshes. *Computers and Fluids* 1996; **5**:119–132.
46. Cuthill E, McKee J. Reducing the bandwidth of sparse symmetric matrices. *Proceeding of ACM National Conference*, 1969; 151–172.
47. 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.
48. Luo H, Baum JD, Löhner R. Edge-based finite element scheme for the Euler equations. *AIAA Journal* 1994; **32**(6):1183–1190.
49. Cebral JR, Camelli F, Soto O, Löhner R, Yang C, 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.
50. Corrigan A, Camelli FE, Löhner R, Wallin J. Running unstructured grid-based CFD solvers on modern graphics hardware. *AIAA Fluid Dynamics Meeting*, Austin, TX, AIAA-2009-4001, June 2009.

DOI: 10.1002/fld