J Supercomput (2014) 67:47–68
DOI 10.1007/s11227-013-0985-9

CU++: an object oriented framework for computational fluid dynamics applications using graphics processing units

Dominic D.J. Chandar · Jayanarayanan Sitaraman · Dimitri Mavriplis

Published online: 7 August 2013

Abstract The application of graphics processing units (GPU) to solve partial differential equations is gaining popularity with the advent of improved computer hardware. Various lower level interfaces exist that allow the user to access GPU specific functions. One such interface is NVIDIA's Compute Unified Device Architecture (CUDA) library. However, porting existing codes to run on the GPU requires the user to write *kernels* that execute on multiple cores, in the form of Single Instruction Multiple Data (SIMD). In the present work, a higher level framework, termed CU++, has been developed that uses object oriented programming techniques available in C++ such as polymorphism, operator overloading, and template meta programming. Using this approach, CUDA *kernels* can be generated automatically during compile time. Briefly, CU++ allows a code developer with just C/C++ knowledge to write computer programs that will execute on the GPU without any knowledge of specific programming techniques in CUDA. This approach is tremendously beneficial for Computational Fluid Dynamics (CFD) code development because it mitigates the necessity of creating hundreds of GPU *kernels* for various purposes. In its current form, CU++ provides a framework for parallel array arithmetic, simplified data structures to interface with the GPU, and smart array indexing. An implementation of heterogeneous parallelism, i.e., utilizing multiple GPUs to simultaneously process a partitioned grid system with communication at the interfaces using Message Passing Interface (MPI) has been developed and tested.

Keywords Graphics processing units (GPUs) · Parallel computing · Computational Fluid Dynamics (CFD)

D.D.J. Chandar (✉) · J. Sitaraman · D. Mavriplis
University of Wyoming, 1000E. University Avenue, Dept. 3295, Laramie, WY 82072, USA
e-mail: dchandar@uwyo.edu

1 Introduction and background

Graphics processing units (GPUs) have recently been used to solve a wide range of problems, and are becoming the cornerstone of high performance computing. Its exceptional performance compared to CPUs can be attributed to the fact that GPUs have a large number of cores with multi-threading capability, and are capable of executing tens of thousands of threads concurrently [1]. Since the GPU computing architecture relies on a SIMD model, most of the CFD codes will be able to reap benefits through this form of parallelism. The last few years have seen a steep growth in the application of GPUs towards general-purpose applications, such as numerical modeling of fluid flows, image processing, and molecular dynamics [2]. Some examples include, a three-dimensional computation by Hagen et al. [3] on the 7800GTX graphics card where a speed-up of 11.5 was observed on 530,000 points for a Raleigh–Taylor instability problem and Elsen et al. [4] who reported a speed-up of 20 for an Euler computation on a full hypersonic vehicle with complex geometry. Brandvik and Pullan [5] investigated two different GPU front end codes, BrookGPU [6] and CUDA [7] for accelerating a three-dimensional Euler solver. A speed-up of 29 and 16 were obtained for two-dimensional problems with a grid size of 40,000 points and three-dimensional problems with a grid size of 400,000 points, respectively. Cohen et al. [1] have provided ample information on improving GPU performance using a three-dimensional Raleigh–Bernard convection problem. A speed-up of 8.5 was obtained for 28 Million points on a Tesla C1060 graphics card. Using a multi-GPU programming paradigm with MPI, Phillips et al. [8] obtained a speed-up of 496 using 32 GPUs on 6 Million points for a two-dimensional Euler calculation. Apart from standard finite-volume or finite-difference methods on the GPUs, there have been calculations using Lattice Boltzmann methods as described by the analysis in [9]. Utilizing the NAS parallel benchmarks [10], Lu et al. [11] performed large scale computations on the TianHe-1A Supercomputer, and discussed various strategies for multiple GPU implementations.

Writing codes that run on a GPU requires an intermediate low level interface (a GPU front end) that can transfer data between the CPU and GPU and perform the required computation on the GPU. NVIDIA's CUDA architecture [7] is one such interface that supports native C/C++ language constructs. Similar to the MPI standard, where commands are concurrently executed on various processors, the CUDA programming model relies on *kernels* that execute on multiple threads. *Kernels* are similar to standard programming language functions, except for the manner in which these functions are invoked from the main program. One can write *kernels* for each arithmetic expression, or wrap a set of expressions into one *kernel*. A single call to a *kernel* will automatically spawn as many as processes the user wants, provided the number of processes is within the limits of the GPU. The aforementioned method of writing a *kernel* is widely practiced and is quite popular among the CUDA community. However, there arise situations where one has to write different *kernels* for different expressions, thereby making the code bulky and sometimes difficult to manage. For example, a three-dimensional Euler solver will require *kernels* to compute the fluxes, derivatives in each direction, residual, and to do the time stepping. If viscous terms are needed at a later time, another *kernel* has to be written. The complexity of the code thus increases as more features are added over a period of time. To

ease the pressure off the user while writing codes using CUDA, a novel higher level framework has been developed that encapsulates *kernels* using operator overloading and expression templates [12], and leaves the user to use normal arithmetic expressions without having the need to manipulate *kernels*. Similar higher level strategies for writing simpler GPU codes can be found in Cohen [13], Chen et al. [14], and Enmyren [15]. There are also semi-automatic GPU porting strategies available in Corrigan [16] and Poole [17]. The present framework is similar to the work described previously, but it is directed at porting the A++P++ [18] numerical library which has been extensively used for the Overture [19] numerical solvers. Overture [19] has been used widely for incompressible flow computations on moving overset grids [20–22]. A notable difference between this framework and other implementations is that array indexing is also overloaded, and that indexing operations can be performed within the arguments specified to the vector under consideration (cf. Sect. 2). Users can reuse existing C++ codes without having to make major changes to the programming strategy. The end result of this approach is that one is able to write clear and concise GPU based codes with a little sacrifice to the speed-up.

As a first step towards getting the current framework implemented in a larger scale, the following codes have been developed: (1) ARC3D-GPU code based on ARC3D code [23]. ARC3D is a sixth-order accurate finite-difference based flow solver that has been widely used for the inviscid flow computations using the Euler equations. ARC3D is also the compute engine for the off-body solver used in the HELIOS infrastructure [24], (2) GPUEULER unstructured code [25], and (3) GPUINS unstructured incompressible viscous code [26]. In this paper, emphasis is given to the higher level interface (CU++), and how this has been used to develop the above set of codes. Towards the end, we discuss a unique code environment with a blended CPU–GPU framework where CPUs and GPUs are simultaneously involved in solving a problem. All computations reported in this paper were carried out using several Tesla C2050 GPUs using CUDA 3.2 and several Intel Xeon 5160 dual core processors.

2 A comparison of CPU based serial codes and GPU based CU++ codes

To start with, we will describe through simple examples, how CU++ programs bear similarity to their conventional CPU versions. Implementing a GPU version of the corresponding CPU code is simple and straightforward. As an example, consider the discretization of a Laplace equation $\nabla^2 u = 0$ on a square $0 \leq x \leq 1, 0 \leq y \leq 1$. Using a point Jacobi iterative procedure on an $N \times N$ grid, one can write a C++ version of the discretized equation as shown in Listing 1. The CU++ version is shown in Listing 2. As seen, both versions look alike except for the fact that the loops have been avoided and the indices for the arrays are now *Index* objects for the CU++ version. This is, however, not possible with the usual GPU implementation found in existing GPU front ends. During the compilation stage using the CUDA compiler *nvcc* [7], the compiler scans through each expression, and builds an abstract object that represents the expression. This abstract object is unrolled during run time inside a common GPU *kernel*. All arithmetic expressions that exist in the code are converted to these abstract objects and use only a single *kernel* for expression unrolling.

Listing 1 C++ version of the Jacobi iteration scheme for solving $\nabla^2 u = 0$

```
//u is a int/float/double array
for ( step = 0 ; step < maxNumberofSteps ; step++ )
  {
   for ( i = 1 ; i < N-1 ; i++ ) // Loop around internal nodes along Y
    {
      for ( j = 1 ; j < N-1 ; j++) // Loop around internal nodes along X
       {
         u(i,j) = 0.25*( u(i,j+1) + u(i,j-1) + u(i+1,j) + u(i-1,j) ;
       }
    }
  }
```

Listing 2 CU++ version of the Jacobi iteration scheme for solving $\nabla^2 u = 0$

```
// Index objects are used to represent the base and bound of the array
Index i(1,N-2), j(1,N-2);
// u is an instance of the distributed array class defined as follows:
distArray u(N,N);
for ( step = 0 ; step < maxNumberofSteps ; step++ )
   {
        u(i,j) = 0.25*( u(i,j+1) + u(i,j-1) + u(i+1,j) + u(i-1,j) ;
   }
```

Listing 3 CU++ source code for the 1-D diffusion equation $u_t = u_{xx}$

```
// Number of grid points
int N = 100; float dx = 1.0/(N-1), dt = 0.0001;
// Initialize the GPU
distArray::setCudaProperties(N,50); //( Tell CU++ to execute 50 threads
per block )
// u is a distributed array object defined as follows:
distArray u(N);
// Set the Boundary condition;
u(0) = 0.0; u(N-1) = 1.0;
// Index objects are used to represent the base and bound of the array
Index i(1,N-2);
for ( step = 0 ; step < 10 ; step++ )
  {
       u(i) = u(i) + (dt/(dx*dx))*( u(i+1) - 2*u(i) + u(i-1) );
  }
// grab the results from the gpu on to cpu memory
u.pull();
// print the results
u.display();
// End of code
distArray::cleanUp();
```

By using this methodology, one avoids the need to write *kernels* for each expression.

Listing 3 is the full source code for solving the one-dimensional diffusion equation $u_t = u_{xx}$ using an explicit Euler time stepping method, and second order central differences for the diffusion term. It is clearly observed that the code in this listing is very similar to any standard C/C++/FORTRAN code, and one does not need to know the basic functions of GPU to develop this piece of code.

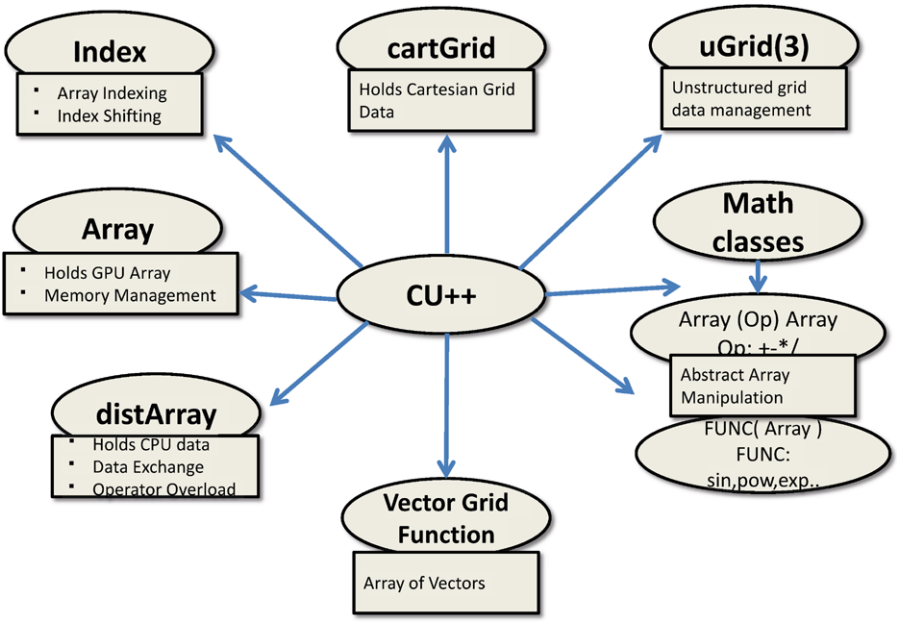

Fig. 1 An overview of the CU++ classes

3 A description of the CU++ framework

The CU++ framework is heavily based upon vector array arithmetic using C++ classes and template programming as in Fig. 1. Generally, all arrays are declared using the main class known as *distArray* (distributed array). This class holds the data for the CPU/GPU version of any array, and has simple functions to transfer data between the CPU and GPU and to perform arithmetic operations. For example, this class overloads the operator = on the GPU, so that one can perform $u(I) = some\ function$ where I is an *Index* object which dictates where this assignment operation needs to be performed. The class *Array* is declared inside class *distArray* and points to the GPU version of the array, and is responsible for the memory management. Each time an instance/object of *distArray* is created, two copies of the same array are generated, of which one resides in the CPU and the other in the GPU. The CPU version of the array is used only when data needs to be used for post-processing or during a multi-GPU computation using MPI. In the latter case, the CPU version of the array is first populated along fringe points from the data on the GPU. Standard MPI communication routines are then used to transfer the data between domains. Following the transfer, the GPU arrays are populated from the corresponding CPU arrays. This three-way procedure is due to the fact that GPUs residing on different *compute nodes* cannot communicate with themselves without bypassing the CPU. Unless the *MPI* standard accepts GPU memory addresses, a direct transfer of data between GPUs is not possible. The class *Index* is one of the simplest of all the classes, and it mimics the *Index* class of the package *A++* [18], a serial/parallel array class used for vector arithmetic. This class is used to store the base and bound of a given array. For exam-

Listing 4 Indexing for unstructured data

```
// Declare an array to hold the solution
distArray Q(number_of_nodes);

// Declare an array to hold the boundary node indices and use simple names
distArray bnodeIndex( number_of_boundary_nodes );
Index I(0,number_of_boundary_nodes-1);
#define BI bnodeIndex(I);

// get the boundary node indices
getBoundaryNodeIndex( bNodeIndex );

// Do a small computation on the boundary nodes
// x, y are distArrays that hold the x- and y-coordinates of the grid
// Kernel gets automatically generated at compile time
Q(BI) = Q(BI) + SIN(x(BI))*COS(y(BI));
```

ple, in Listing 2, i, j are *Index* objects, and the indices run from 1 to $N-2$, i.e., on internal nodes. For structured type of data, an instance of the *Index* class is passed as an argument to a *distArray* as in Listings 2 and 3. Instances of the class *Index* can also be made to perform arithmetic operations such as index shifting $i+k$ where k represents the displacement in the array position. This function is very useful for finite-difference calculations like the one shown in Listings 2 and 3. For unstructured data, if one is interested in updating the boundary nodes or a specific region, this is, however, not possible using the *Index* class alone, as nodal indices are not ordered. Hence indices are stored as *distArrays* themselves for unstructured problems. For example in Listing 4, we increment the values at the boundary nodes by an arbitrary function. The variable *BI* in the code is a *distArray* which represents the index of all boundary nodes. Index shifting operations are not implemented for unstructured data as they make little sense in an unstructured representation. For example, if i represents indices of all cell centers which are not ordered, $i+1$ may not represent the indices of its neighbors. We resort to standard *kernel* implementations in the case of unstructured data involving dependency from neighbors. To simplify the data structure in the case of multiple components of an array, the class *vectorGridFunction* is created, and is used to represent an array of *distArrays*.

At the heart of the CU++ framework, are the math classes which are solely responsible for automatic GPU *kernel* generation. Most of the classes in this category are *abstract*, in the sense that their *types* are unknown when the code is written. During compile time, all arithmetic expressions in the code are bundled into abstract objects which are unrolled at run time inside the GPU *kernel*. Figure 2 shows how this is achieved for the expression in Listing 2. Since the GPU version of *distArray* lies in the class *Array*, only the class *Array* is referenced in Fig. 2(a). Every time when two *distArrays* are required to be added, it returns an abstract object of type *Gen* at compile time. This object *Gen* is then made to perform other numerical operations dictated by the expression given by the user. At the end of the compilation phase, each expression would represent an abstract object. Inside each abstract object's type definition, we overload the operator [ ] to be able to point to the desired array location which is useful for stencil calculations such as index shifting. This operation is performed until the compiler hits the $=$ symbol. We then write one GPU *kernel* which will accept the abstract object using templates, un-roll the individual compo-

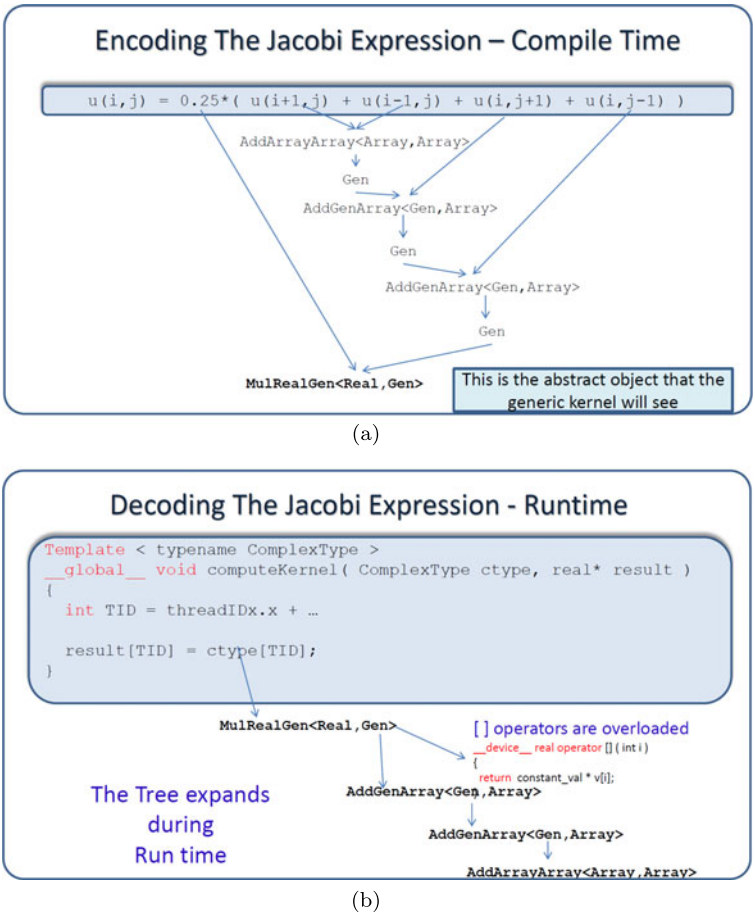

Fig. 2 (**a**) Building the abstract object during compilation; (**b**) run time un-rolling of the abstract object

nents of this abstract object and perform the required operation as in Fig. 2(b). The variable *TID* is well known in CUDA literature as the *ThreadID* and this represents the index(or grid point/location) where the operation is being performed. All vector expressions in this method call only one *kernel*, thus avoiding the necessity to write many *kernels*.

Some of other miscellaneous operations that CU++ can handle is given in Fig. 3. Note that there are no explicit GPU *kernel* invocations or *kernel* definitions for each of these statements, and that all of these statements call a single *kernel* of the type described in Fig. 2(b). To compile CU++ codes, a compiler tool *mpiugc* is shipped along with the CU++ package as an executable for Linux x86-64 architectures. This compiler tool is a wrapper using C++ string functions that calls the required NVIDIA CUDA compiler, and adds the required CU++ libraries(serial/parallel) depending on a compile time flag. All other source codes as a part of this framework are open-source and can be obtained by a written permission from the corresponding author.

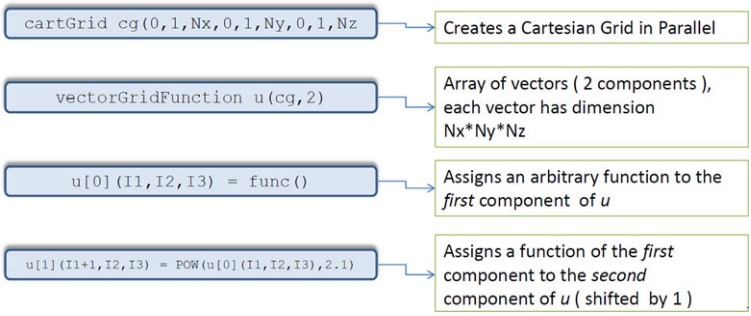

Fig. 3 CU++ miscellaneous functions

The following command compiles a CU++ code and builds an executable.

mpiugc -arch = sm_xx test.cu -o test

The option *-arch = sm_xx* specifies the GPU compute capability xx. For example, the Tesla C2050 has a compute capability of 2.0, hence $xx = 20$. One does not need to change the actual source code *test.cu* if only serial execution is required. More details on this can be found in Sect. 5.1.

4 Computational modeling

The CU++ framework is tested by solving standard partial differential equations occurring in fluid dynamics such as the (A) 3D Compressible Euler equations on a structured grid and (B) 2D Incompressible Navier–Stokes equations on an unstructured grid. Code snippets are provided to demonstrate the ease of programming using CU++.

4.1 3D compressible flow

The governing equations for inviscid compressible flow are given by

$$\frac{\partial Q}{\partial t} + \frac{\partial E}{\partial x} + \frac{\partial F}{\partial y} + \frac{\partial G}{\partial z} = 0 \tag{1}$$

The vectors Q, E, F, and G are all declared as *vectorGridFunctions* having five components, and are given by

$$Q = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ e \end{pmatrix}, \quad E = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uw \\ (e+p)u \end{pmatrix}, \quad F = \begin{pmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho vw \\ (e+p)v \end{pmatrix}, \quad G = \begin{pmatrix} \rho w \\ \rho uw \\ \rho vw \\ \rho w^2 + p \\ (e+p)w \end{pmatrix} \tag{2}$$

The finite-difference spatial discretizations can be expressed in pseudo-finite-volume form as

$$\frac{\partial E}{\partial x} = \frac{\hat{E}_{i+1/2} - \hat{E}_{i-1/2}}{\Delta x} \tag{3}$$

where $\hat{E}$ represents the total inviscid flux evaluated at the cell-face:

$$\hat{E}_{i+1/2} = \tilde{E}_{i+1/2} - \tilde{D}_{i+1/2} \tag{4}$$

In the above expression, $\tilde{E}$ represents the physical flux and $\tilde{D}$, the artificial dissipation. Using central differences of second-, fourth- and sixth-order accuracy, one can write the physical flux as

$$\tilde{E}^{II}_{i+1/2} = \frac{1}{2}(E_{i+1} + E_i) \tag{5}$$

$$\tilde{E}^{IV}_{i+1/2} = \tilde{E}^{II}_{i+1/2} + \frac{1}{12}(-E_{i+2} + E_{i+1} + E_i - E_{i-1}) \tag{6}$$

$$\tilde{E}^{VI}_{i+1/2} = \tilde{E}^{IV}_{i+1/2} + \frac{1}{60}(E_{i+3} - 3E_{i+2} + 2E_{i+1} + 2E_i - 3E_{i-1} + E_{i-2}) \tag{7}$$

The artificial dissipation terms can be similarly formulated in their discrete forms as

$$\tilde{D}^{II}_{i+1/2} = \frac{|\sigma|_{i+1/2}}{2}(Q_{i+1} - Q_i) \tag{8}$$

$$\tilde{D}^{IV}_{i+1/2} = \tilde{D}^{II}_{i+1/2} - \frac{|\sigma|_{i+1/2}}{12}(Q_{i+2} + 3Q_{i+1} - 3Q_i - Q_{i-1}) \tag{9}$$

$$\tilde{D}^{VI}_{i+1/2} = \tilde{D}^{IV}_{i+1/2} + \frac{|\sigma|_{i+1/2}}{60}(Q_{i+3} - 5Q_{i+1} + 5Q_i - Q_{i-2}) \tag{10}$$

Here, σ is the spectral radius of the inviscid flux Jacobian. In the current implementation, functions are written for a combination of sixth-order physical flux + sixth-order dissipation, and second order physical flux + fourth-order dissipation. Time integration is performed using a low storage, three-stage Runge–Kutta scheme described in Kennedy et al. [27].

4.1.1 GPU Implementation

Implementation in the case of a structured grid follows closely the Listings 2 and 3. For example, to compute the derivatives $\frac{\partial E}{\partial x}$ using Eqs. (4), (5), and (8), Listing 5 shows how this is achieved. A similar methodology is adopted for other derivatives. The expression to the right hand side of the = symbol is converted to an abstract object during compile time as discussed earlier. This object is then passed to a unique GPU *kernel* similar to Fig. 2(b), which is then unrolled into individual components at run time. As can be seen, one does not write a *kernel* each time an expression needs to be executed on the GPU and that each coded expression relates directly to its algorithmic form written on paper.

Listing 5 Computing derivatives in the structured grid formulation

```
// Define a Cartesian Grid
// min, max are the boundaries of the domain
// N-xyz are the number of points in each direction
cartGrid cg(xmin, xmax, ymin, ymax, zmin, zmax, Nx, Ny, Nz);

// Create a Vector Grid Function to hold 5 components of the Euler Equations
vectorGridFunction dEdx(cg,5), E(cg,5), Q(cg,5);
vectorGridFunction sigma_right(cg,1), sigma_left(cg,1);

// Index objects to represent the discretization space
Index i(0,Nx-1), j(0,Ny-1), k(0,Nz-1);

// Compute All Derivatives on the GPU
for ( int component = 0 ; component < 5 ; component++ )
  {
     int & c = component ;
     dEdx[c](i,j,k) = 0.5*(E[c](i+1,j,k) - E[c](i-1,j,k))/dx
                    - 0.5*sigma_right(i,j,k)*( Q[c](i+1,j,k) - Q[c](i,j,k) )
                    + 0.5* sigma_left(i,j,k)*( Q[c](i,j,k) - Q[c](i-1,j,k)) ;
  }
```

4.2 2D Incompressible flow

We also demonstrate the application of this framework towards solving unsteady incompressible flow on unstructured grids. The 2D compressible version on unstructured grids has been successfully implemented in Soni et al. [25]. The equations governing unsteady incompressible flow with Arbitrary-Lagrangian–Eulerian (ALE) terms are given by

$$\frac{\partial \mathbf{U}}{\partial t} + (\mathbf{U} - \mathbf{U_G}) \cdot \nabla \mathbf{U} = -\nabla p + \nu \nabla^2 \mathbf{U} \tag{11}$$

$$\nabla \cdot \mathbf{U} = 0 \tag{12}$$

where $\mathbf{U}$ is the velocity vector, p is the pressure normalized by density, and $\mathbf{U_G}$ is a vector of grid speeds. We use the Pressure-Poisson formulation (PPE) of Henshaw [28], where the divergence constraint Eq. (12) is replaced by a Pressure-Poisson equation by taking the divergence of the momentum equation.

$$\nabla \cdot (\nabla p) = -\nabla \cdot \big((\mathbf{U} - \mathbf{U_G}) \cdot \nabla \mathbf{U} \big) + \nabla \cdot (-\nu \nabla \times \nabla \times \mathbf{U}) \tag{13}$$

For the purposes of discretization, Eq. (11) is written in conservative form for each node i and discretized in a finite-volume framework (Fig. 4) as follows:

$$V_i \frac{\partial \mathbf{U}_i}{\partial t} + \sum_k \mathbf{F} \cdot \mathbf{n} \, dS_k = \nu \sum_k \nabla \mathbf{U} \cdot \mathbf{n} \, dS_k \tag{14}$$

where k represents the dual face index, and $\mathbf{F}$, the non-linear terms that represent the inviscid flux (inclusive of the pressure). Over any dual face k, the non-linear and viscous fluxes are computed as follows:

$$\mathbf{F}_k = \frac{1}{2}(\mathbf{F}_{e1} + \mathbf{F}_{e2}) \tag{15}$$

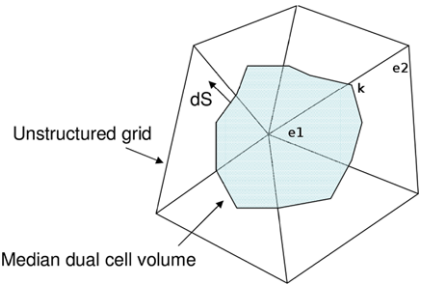

Fig. 4 A portion of the unstructured grid showing the dual cell

$$\nabla \mathbf{U}_k = \nabla \bar{\mathbf{U}}_k - \left(\nabla \bar{\mathbf{U}}_k \cdot \delta_{12} - \frac{\mathbf{U}_{e1} - \mathbf{U}_{e2}}{|\mathbf{x}_{e1} - \mathbf{x}_{e2}|} \right) \delta_{12} \tag{16}$$

where

$$\delta_{12} = \frac{\mathbf{x}_{e1} - \mathbf{x}_{e2}}{|\mathbf{x}_{e1} - \mathbf{x}_{e2}|} \tag{17}$$

$\nabla \bar{\mathbf{U}}_k$ represents the average of the gradients at nodes e_1 and e_2. The gradients at any node i are computed using Green–Gauss theorem. Note that there is no implicit upwinding for the convective terms, and the additional terms appearing in Eq. (16) are used to damp the high frequency modes occurring due to a central scheme [29]. Without this term, the solution will exhibit odd–even type of oscillations.

For temporal discretization, we use a second-order Predictor–Corrector method, with the non-linear terms treated explicitly, and the viscous terms implicitly as described in Henshaw [28]. Reference is made to Chandar [30] for a detailed description of the algorithm and for standard verification test cases.

4.2.1 GPU Implementation

For unstructured grids however, as the algorithm is not so straightforward as that of the structured grid case, not all parts of the code are written using the CU++ format, although the same data structures are retained like the *distArray* class. As the algorithm involves solution to a set of Poisson equations, *kernels* for gradient and Laplacian computations are explicitly written. For simple vector operations such as computing divergence, vorticity, or the *R.H.S.* of the Pressure Poisson Equation, we use the CU++ framework, as they are easily translated to generic *kernels*. In the following Listing 6, the CUDA keywords *grid* and *block* occurring in the *kernel* call represent the number of blocks of data, and the number of threads per block [7].

5 Multiple GPU framework

When there are many GPUs on a compute node, it is possible to use all of them by dividing the workload across each GPU and perform communication using the Message Passing Interface (MPI). A schematic of this procedure is given in Fig. 5. In this work, the multiple GPU framework is implemented for 2D/3D structured grid problems. The domain under consideration can be partitioned and each partition can

Listing 6 CU++ operations for unstructured grid computations

```
// Index objects
Index I(0,nnodes-1);

// The array nodeIndex holds the index of all nodes
#define i nodeIndex(I);

// Compute Divergence and Vorticity
divergence(i) = ux(i) + vy(i);
vorticity(i) = uy(i) - vx(i);

// Compute the right hand side of the Pressure-Poisson equation
// First compute convective and diffusive terms using standard CUDA kernel
convectionDiffusion<<<grid,block>>>(condiff_x, condiff_y,.....);

// Next compute gradients of condiff
gradient_x<<<grid,block>>>(condiff_x, gradcondiff_x,...);
gradient_y<<<grid,block>>>(condiff_y, gradcondiff_y,...);

// RHS of the PPE is now:
rhs(i) = gradcondiff_x(i) + gradcondiff_y(i);
```

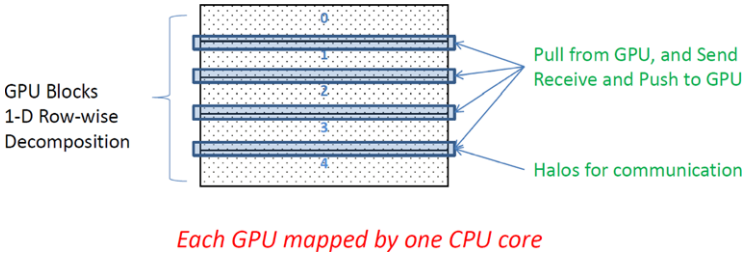

Fig. 5 CPU–GPU mapping for multi GPU computations

be handled by one GPU. In each partition, an extra layer of grid points is added at the boundary and these grid points are termed as "fringe points" and appear as *halos* for communication purposes. As mentioned earlier, since MPI routines only accept CPU based memory addresses, communication between GPUs is not possible without the transfer of data between the GPU and the CPU. Since the data transfer between the GPU and CPU takes place using a PCIe link (which is roughly 100× slower than the GPU bandwidth for 32 Mb data size on Tesla C2070), care should be taken to minimize this data transfer. Hence, only the data on the fringe points are used for communication. The size of the halo will depend on the order of discretization of the PDE being solved. NVIDIA has an easy interface for direct transfer of data between GPUs only within a compute node using the recent version of CUDA [31], termed as *GPUDirect*-peer to peer memory access, but this feature has not been tested in the current implementation.

5.1 The case with insufficient GPUs

Let us assume that we have a total of n_g GPUs and that each of these GPUs is mapped by a CPU core. Thus, we require n_g CPU cores for a one-to-one mapping between the GPU and CPU cores. When we have insufficient number of GPUs required to

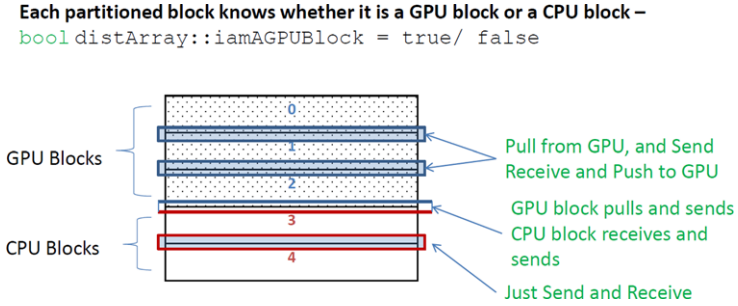

Fig. 6 CPU–GPU mapping for multi GPU computations when the number of GPUs is less than the number of CPU cores

be mapped by a CPU core, we would have for example, additional n_c CPU cores, and these CPU cores do not have to be idle. These CPU cores can be made to work efficiently by proper load balancing. Figure 6 shows a schematic for the present case, and this approach currently works only for 2D structured grid problems. Each block or partition will know whether the local computation needs to be performed on the GPU or on a CPU core by the flag *iamAGPUBlock*. If this flag is *false* then the GPU arrays will point to the CPU version of the arrays. If both blocks sharing a boundary are GPU blocks such as process 0 and 1 in Fig. 6, the communication methods described in the previous section are adopted. If the blocks are of a different kind (such as process 2 and 3), only the GPU block would transfer data between its corresponding CPU core, and the CPU block will receive and send data directly without any GPU interference. When both blocks are CPU blocks such as process 3 and 4, only standard communication procedures are used (MPI send and receive). Since the GPU is an order of magnitude faster than one CPU core, partitioning the domain into equal blocks will not be beneficial. Instead one needs to do proper load balancing to reap the benefits of using the CPU cores as well. We describe this load balancing procedure below:

Let us assume that T is the total problem size (e.g. number of grid points), s the speed-up of a GPU relative to one CPU core, n_g the number of GPUs, n_c the number of CPU cores, N_1 the partitioned problem size on the GPU and N_2 the partitioned problem size on the CPU. Based on these variables, the total problem size can be expressed as

$$T = n_g N_1 + n_c N_2 \tag{18}$$

For the load to be balanced between a CPU core and GPUs, one must have

$$N_1 = s N_2 \tag{19}$$

The above is only an assumption, as GPU speed-ups may not always vary linearly with grid size. Using Eq. (19) in Eq. (18), we obtain

$$N_1 = \frac{sT}{n_g s + n_c} \tag{20}$$

$$N_2 = \frac{T}{n_g s + n_c} \tag{21}$$

Equations (20) and (21) represent the load-balanced problem size on respective (GPU/CPU) partitions. To compute that however, one needs to know the quantity s, which represents the speed-up of one GPU relative to one CPU core. Hence, the problem is solved initially without any partitioning to compute the speed-up of the GPU code. To make a direct comparison between GPU and CPU-serial versions, we have developed a unified framework where one maintains a single copy of the code, and different partitions can be executed on the GPU or on the CPU core based on one's need. This is done by adding an extra argument to the *mpirun* command as follows:

mpirun -np n_c ⟨*progname*⟩ *-ngpu n_g*

Thus if one would want to run the entire problem on a single CPU core without GPU capability, n_c would be 1 and n_g would be 0. To run on one GPU n_g would be 1. Thus the speed-up s can be computed in a very straightforward manner without changes to the original code. Taking a step further, let us examine the speed-up gained by making use of idle CPU cores. If only GPUs are used, then the time spent by each GPU is roughly $t_{GPU1} \approx \frac{T}{n_g}$. By using additional CPU cores with load balanced, the time spent by a GPU is $t_{GPU2} \approx N_1$. Thus the speed-up gained by using additional CPU cores is given by

$$t_{GPU1}/t_{GPU2} = 1 + \left(\frac{n_c}{n_g}\right)\frac{1}{s} \tag{22}$$

Note that this expression does not involve any communication time. Further discussion on this can be found in Sect. 6.2.2.

6 Results and discussions

In this section, we demonstrate our computational framework towards solving standard fluid dynamics problems on single and multiple GPUs.

6.1 Single GPU computations

6.1.1 3D compressible flow

The inviscid convection of a vortex is used as a test problem to validate the ARC3D-GPU solver on the Fermi (Tesla C2050). A vortex situated initially at an arbitrary location will convect without dissipation depending on the free-stream conditions. An exact solution exists for this problem, and is given by

$$\vec{U}(\vec{r}) = \frac{\Gamma}{2\pi h}\left(1 - e^{-\sigma h^2}\right)\hat{e}_\theta \tag{23}$$

where Γ is the circulation, $\vec{r}$ is the position vector, σ is the strength of the vortex, h is the orthogonal distance from $\vec{r}$ to the axis of the vortex and $\hat{e}_\theta$ is the vector

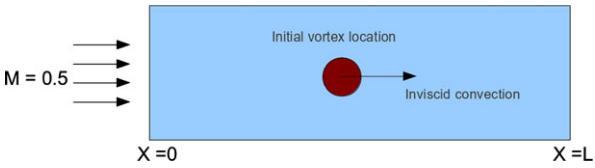

Fig. 7 Problem setup for the Lamb vortex propagation on a single GPU

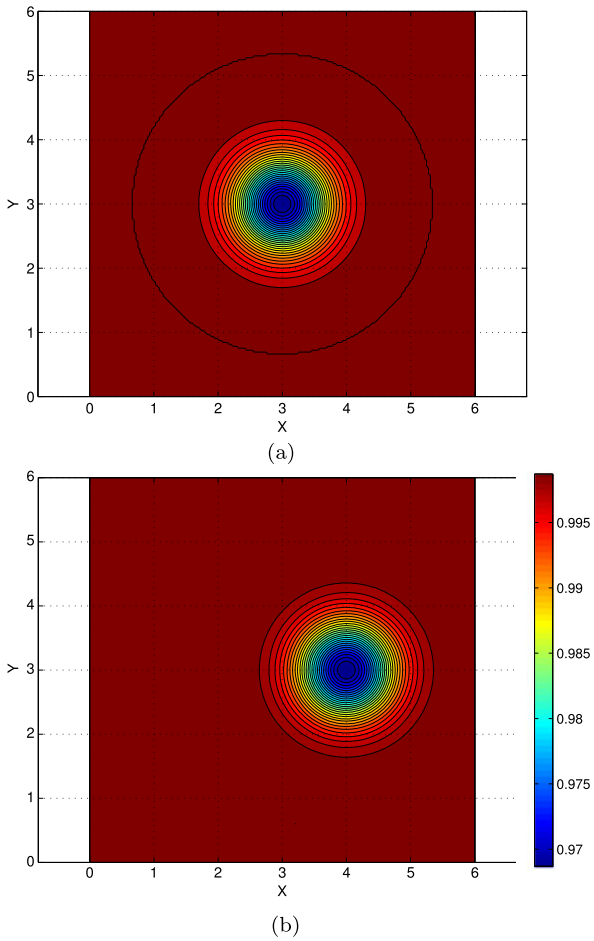

Fig. 8 Density contours at (**a**) $t = 0$ and (**b**) $t = 200$ time steps

orthogonal to the plane containing $\vec{r}$ and the axis of the vortex. The vortex is initially placed in a box ($0 \leq x \leq 6$, $0 \leq y \leq 6$, $0 \leq z \leq 0.6$) at a location of (3,3) and extends along the full z-direction as given in Fig. 7. Using a grid size of $200 \times 200 \times 150$ and a sixth-order spatial differencing scheme for the physical fluxes and dissipation terms, for a free-stream Mach number of $M = 0.5$, the solution is computed for 200 time steps using a time step of $\Delta t = 0.01$. Figure 8 shows the contours of density at $t = 0$, and after 200 time steps.

The vortex has convected with minimal dissipation and its location is consistent with that of the exact solution ($x = 4$). Table 1 shows a comparison of the wall clock

Table 1 Timing comparisons for the inviscid vortex convection problem

Grid points	Solver	Wall clock time per time step (s)
6 Million	ARC3D(FORTRAN)	50
6 Million	CU++	0.7

Table 2 Computational parameters for the plunging airfoil case

	0012	0014
Reduced frequency k, $\omega c/U_{\text{inf}}$	12.3	2.0
Plunge amplitude h_0	0.12	0.4
Reynolds number	500	10000
Number of triangles	30000	32000

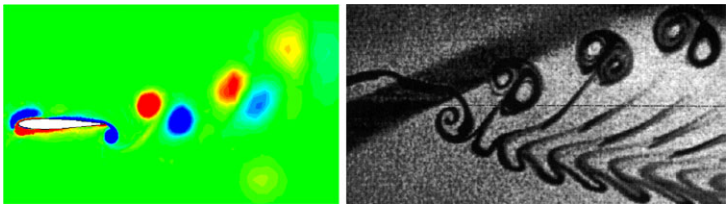

Fig. 9 A comparison of flow structures behind a plunging airfoil between GPUINS computation (*left*) and experiments from Jones et al. [32] (*right*)

time spent per time step using the present approach with that of ARC3D. It can be observed that speed-up of 70 is obtained.

6.1.2 2D incompressible flow

To validate the incompressible flow solver on unstructured grids, we consider two cases of a plunging NACA 0012, 0014 airfoil at Reynolds number (Re) = 500, and 10000, respectively. For Re = 500, current computations are compared with flow visualization results of Jones et al. [32], and for Re = 10000, with the overset grid computations of Tuncer and Kaya [33]. A sinusoidal motion of the form $h = h_0 \sin(\omega t)$ is prescribed for the airfoil. Table 2 shows various parameters for the two test cases. Solutions are computed for six cycles of oscillation, and the corresponding vorticity contours for NACA0012 airfoil are shown in Fig. 9 in comparison with the flow visualization results of Jones et al. [32]. A satisfactory comparison is obtained in terms of the wake deflection. Computed drag coefficients in Fig. 10 also show good comparison with the computations of Tuncer and Kaya [33] giving us confidence in the implementation of the algorithm. All computations presented in this section were performed in double precision. Over one time step, the serial version of the same code consumed 6.6 s of CPU time, whereas the GPU code took 1.2 s, resulting in 5.5× speed-up. Noting that GPU performance increases with the number of grid points [25], it is anticipated that three-dimensional problems will benefit to a greater extent.

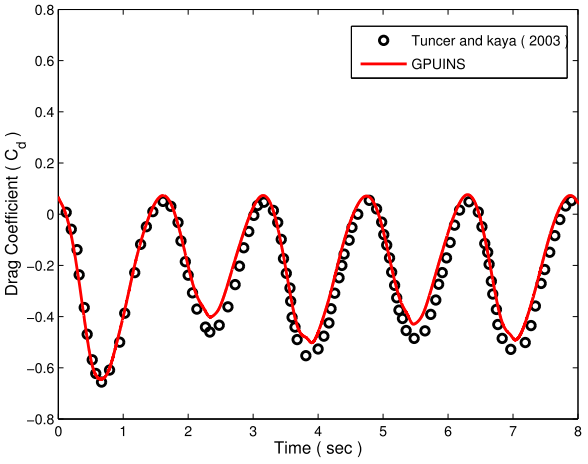

Fig. 10 Time history of the Drag coefficient for a plunging NACA0014 airfoil at Re = 10000

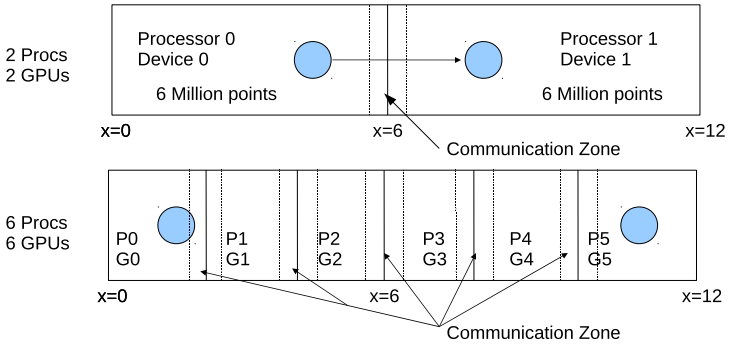

Fig. 11 Problem setup for the Lamb vortex propagation on multiple GPUs

6.2 Multi-GPU computations

6.2.1 Compressible flow multi-GPU computations with equal number of GPU and CPU cores

We consider the same test case of an inviscid vortex convection, but on a bigger domain. Figure 11 describes the partition of the domain on two and six processes, respectively. The vortex is initially placed in a box ($0 \leq x \leq 12$, $0 \leq y \leq 6$, $0 \leq z \leq 0.6$) at a location (4, 3) and extends along the full z-direction. The grid size for this problem is $400 \times 200 \times 150$. Each partitioned chunk of the domain is mapped to one GPU. The interface between the partitioned domains is not disjoint, but has three layers of fringe points on each side of the interface (to maintain formal sixth-order accuracy), each belonging to each respective domain. When it is required to update the solution on the boundaries, each process pulls only the fringe layer data from the GPU on to the CPU, and sends it to the neighboring domain using the procedure described in Sect. 5. The neighboring domain receives the data on CPU, then pushes it on to the GPU, and continues with the computation. This incurs some overhead, as

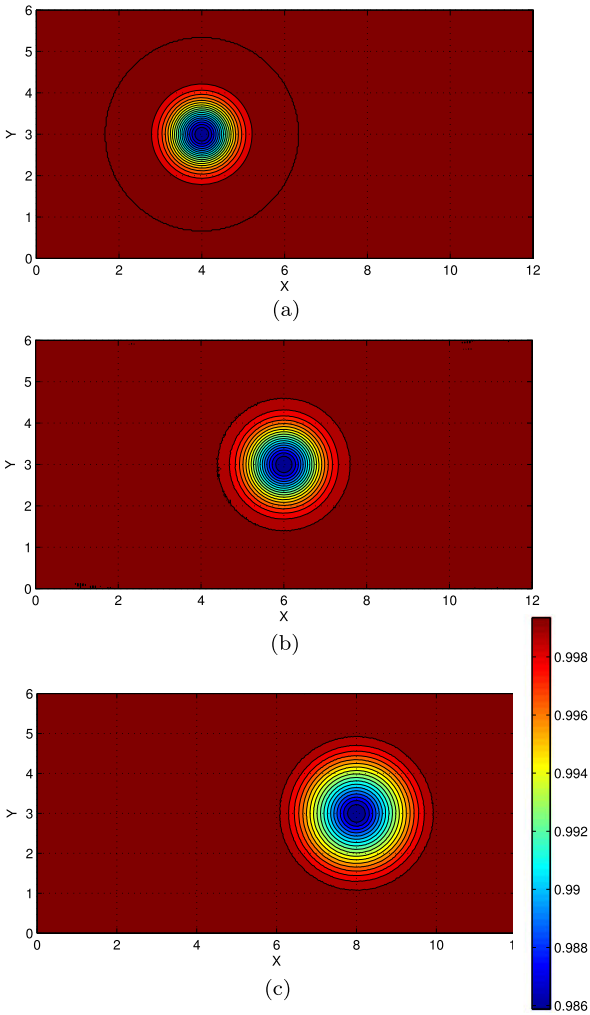

Fig. 12 Density contours at time (**a**) $t = 0$, (**b**) $t = 400$ time steps, and (**c**) 800 time steps using multiple GPUs

data is copied back and forth between the CPU and GPU, hence perfect scalability might not be obtained. Figure 12 shows the contours of density at three different time instants for a Mach number $M = 5.0$, and $\Delta t = 0.001$. After 400 time steps, the vortex is situated exactly at the interface of two domains. The smooth contours indicate that communication between the two domains has been established without errors. Also in Table 3 a comparison of the wall clock times spent with that of the previous approaches (serial, single GPU) is provided.

6.2.2 Multi-GPU computations with unequal number of GPUs and CPU cores

We demonstrate the multi-GPU framework in the case when the number of GPUs are less than the number of CPUs by solving a 2D Poisson equation $\nabla^2 U = f$ on a unit rectangular domain (8192×8192 grid) with constant Dirichlet boundary conditions

Table 3 Timing comparisons for the inviscid vortex convection problem with and without MPI

Grid points	Solver	Wall clock time per time step (s)
6 Million	ARC3D(FORTRAN)	50
6 Million	CU++ using one GPU	0.70
12 Million	CU++ using one GPU	1.37
12 Million	CU++ + MPI using two GPUs	0.80
12 Million	CU++ + MPI using six GPUs	0.39

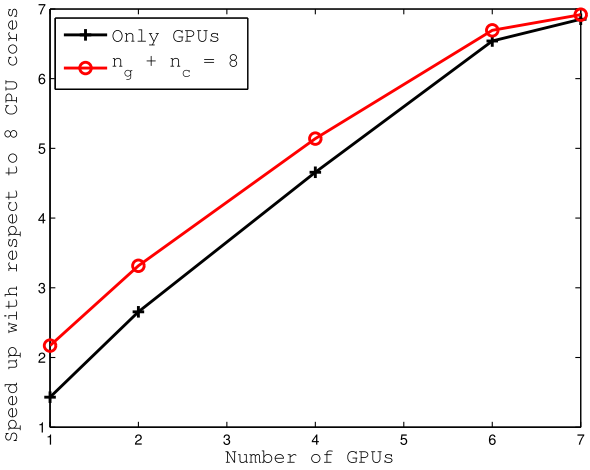

Fig. 13 A comparison of the GPU speed-up with respect to an eight CPU core parallel code for mixed GPU and CPU cores

using a point Jacobi method. Our compute node features seven Tesla C2050 GPUs and eight Intel Xeon CPU cores. In the first case, a one-to-one mapping of each GPU to each CPU core is performed, hence the seven GPUs are controlled by CPU cores. One CPU core remains idle in the present case. In the second case, the mixed GPU–CPU implementation described in Sect. 5.1 is turned on, and the CPU cores also perform some computations such that the sum of GPUs and CPU cores equals 8. Both these cases are compared to the performance of the same code executed on eight CPU cores without GPU functionality. Figure 13 shows a comparison of these cases. We can see that if the number of GPUs is much greater than the number CPU cores, little benefit can be gained by using the additional CPU core (7 GPUs + 1 CPU core). However, when one has very limited GPU resources (1 GPU + 7 CPU cores), using the CPU cores to do part of the work results in an additional 50 % increase in the speed-up. In Table 4, we compare the theoretical (Eq. (22)) and computed speed-up gained by using both GPU and CPU cores. The differences in each of the cases are essentially due to the communication time between partitions. As for the 7 GPU + 1 CPU core case, the speed-up values differ in the third decimal place with the computed speed-up being less than the theoretical speed-up, hence they appear to be the same.

Table 4 A comparison of the theoretical (Eq. (22)) and computed speed-up for mixed GPU–CPU core implementation of the Poisson problem

nGPUs	nCores	Theoretical speed-up	Computed speed-up
1	7	1.7	1.51
2	6	1.3	1.25
4	4	1.1	1.09
6	2	1.03	1.02
7	1	1.02	1.02

Table 5 Wall clock time (s) for the Poisson problem using CUDA-C and CU++

Order of discretization	CUDA-C	CU++
2	79.1	83.8
6	106.5	112.2

Table 6 Number of coded lines for the Poisson problem using CUDA-C and CU++

CUDA-C	CU++
33	**7**

7 CU++ vs. CUDA-C Comparison

Having described the advantage of CU++, it is worth mentioning a few points as to how this performs relative to a standard CUDA-C implementation. We consider solving a 2D Poisson problem on a very fine grid 12288×12288 on a unit rectangular domain using a point Jacobi method. For the CUDA-C implementation, we write *kernels* for computing the discrete Laplacian using both second- and sixth-order finite differences. For the CU++ implementation, no *kernels* are written, and the discrete Laplacian as in Listing 2 gets converted to *kernels* automatically at compile time. The wall clock times are computed for all cases after 1000 iterations, and are listed in Table 5. It is observed that CU++ is only 5 % slower than the CUDA-C implementation, and this is due to the fact that CU++ uses a lot of operator overloading techniques to achieve a cleaner code. Not counting the lines in the code which are common between CUDA-C and CU++, using CU++ results in a very compact code (about 78 % reduction in the number of lines) as seen in Table 6.

8 Summary

In this paper, a framework to write CUDA compatible GPU codes using standard C++ language constructs has been developed and tested. The novelty of this application is two-fold: (1) *kernel* generation for vector operations with indexing is automatic, and is achieved at compile time using C++ expression templates and (2) one maintains a unique code, and the same code can be made to execute on either GPUs or CPU cores. Several examples describing the ease of implementation were also discussed. Using this framework, a three-dimensional Cartesian based Euler solver was developed, and

improved performance was achieved compared to the serial CPU version of the same code. Using multiple GPUs, with each GPU mapped to one process using MPI on a single node, further improvements to speed-up was obtained. Noting that scalability was an issue due to CPU–GPU transfer of data, further computations have been planned using CUDA's *GPUDirect* peer to peer memory access, where one obviates the necessity to transfer data between CPU and GPU on a single compute node, and that GPUs can communicate directly without CPU interference. An unstructured grid based incompressible Navier–Stokes solver has also been developed partly using the CU++ framework, and has proven to reproduce some of the results from available data in literature. In-depth validation and integration of the above solvers using a parallel multi-GPU overset grid framework is under development [34] and the results of these computations would be published in the near future.

Acknowledgements We gratefully acknowledge support from the Office of Naval Research under ONR Grant N00014-09-1-1060.

References

1. Cohen JM, Molemaker MJ (2009) A fast double precision code using CUDA. In: Proceedings of parallel CFD, Moffett Field, CA
2. General-purpose computation on graphics hardware. http://gpgpu.org
3. Hagen TR, Lie K-A, Natvig JR (2006) Solving the Euler Equations on Graphics Processing Units/ In. Lecture Notes in Computer Science, vol 3994. Springer, Berlin, pp 220–227
4. Elsen E, LeGresley P, Darve E (2008) Large calculation of the flow over a hypersonic vehicle using a GPU. J Comput Phys 227(24):10148–10161
5. Brandvik T, Pullan G (2008) Acceleration of a 3D Euler solver using commodity graphics hardware. 46th AIAA aerospace sciences meeting and exhibit, AIAA-2008-0607, Reno, NV
6. Buck I (2003) Data parallel computing on graphics hardware. Graphics Hardware
7. NVIDIA CUDA C programming Guide 4.0. http://developer.nvidia.com/cuda-toolkit-40
8. Phillips EH, Zhang Y, Davis RL, Owens JD (2009) Rapid aerodynamic performance prediction on a cluster of graphics processing units. In: 47th aerospace sciences meeting and exhibit, AIAA-2009-0565, Orlando, FL
9. Bailey P, Myre J, Walsh SDC, Lilja DJ (2009) Accelerating lattice Boltzmann fluid flow simulations using graphics processors. In: Parallel processing, Vienna, Austria, pp 550–557. doi:10.1109/ICPP.2009.38
10. NAS parallel benchmarks. http://www.nas.nasa.gov/publications/npb.html. Accessed 10 June 2013
11. Lu F, Song J, Cao X, Zhu X (2011) Acceleration for CFD applications on large GPU clusters: an NPB case study. In: Computer sciences and convergence information technology, Seogwipo, South Korea, pp 534–538. ISBN:978-1-4577-0472-7
12. Vandevoorde D, Josuttis N (2003) C++ templates: the complete guide. Pearson Education, Upper Sadle River
13. Cohen J (2012) Processing device arrays with C++ metaprogramming. In: GPU computing gems, Jade edition. Morgan Kaufmann, San Mateo. doi:10.1016/B978-0-12-385963-1.00044-7
14. Chen J, Joo B, Watson W, Edwards R (2012) Automatic offloading C++ expression templates to CUDA enabled GPUs. In: Parallel and distributed processing symposium workshops and PhD forum, Shanghai, China, pp 2359–2368. doi:10.1109/IPDPSW.2012.293
15. Enmyren J, Kessler CW (2010) SkePU: A multi-backend skeleton programming library for multi-GPU systems. In: Proc 4th int workshop on high-level parallel programming and applications (HLPP-2010), Baltimore, Maryland, USA, September 2010. ACM, New York
16. Corrigan A, Camelli F, Lohner R, Mut F (2011) Semi-automatic porting of a large-scale Fortran CFD code to GPUs. Int J Numer Methods Fluids 69(6):314–331
17. Poole D (2012) Introduction to OpenACC directives. In: NVIDIA GPU technology conference
18. Quinlan D (2000) A++P++ manual. UCRL Report No: UCRL-MA-136511, Lawrence Livermore National Laboratory

19. Brown DL, Chesshire GS, Henshaw WD, Quinlan DJ (1997) Overture: an object oriented software system for solving partial differential equations in serial and parallel environments. In: Eighth conference on parallel processing for scientific computing. Society for Industrial and Applied Mathematics, Paper CP97
20. Chandar D, Damodaran M (2008) Computational study of unsteady low Reynolds number airfoil aerodynamics on moving overlapping meshes. AIAA J 46(2):429–438
21. Chandar D, Damodaran M (2010) Numerical study of the free flight characteristics of a flapping wing in low Reynolds numbers. J Aircr 47(1):141–150
22. Chandar D, Damodaran M (2009) Computation of low Reynolds number flexible flapping wing aerodynamics on overlapping grids. AIAA 2009-1273, presented at the 47th AIAA aerospace sciences meeting and exhibit, Orlando, FL, USA, January 2009
23. Pulliam TH (1984) Euler and thin layer Navier–Stokes codes: ARC2D, ARC3D. UTSI E02-4005-023-84. Computational fluid dynamics, University of Tennessee Space Institute
24. Sankaran V, Sitaraman J, Wissink A, Datta A, Jayaraman B, Potsdam M, Mavriplis D, Yang Z, O'Brien D, Saberi H, Cheng R, Hariharan N, Strawn R (2010) Application of the Helios computational platform to rotorcraft flowfields. In: 48th AIAA aerospace sciences meeting and exhibit, AIAA-2010-1230, Orlando, FL
25. Soni K, Chandar DDJ, Sitaraman J (2011) Development of an overset grid computational fluid dynamics solver on graphical processing units. In: 49th AIAA aerospace sciences meeting and exhibit, AIAA-2011-1268, Orlando, FL
26. Chandar D, Sitaraman J, Mavriplis D (2012) Dynamic overset grid computations for CFD applications on graphics processing units. Paper ICCFD7-12-2. In: Proceedings of the international conference on computational fluid dynamics, Big Island, Hawaii
27. Kennedy CA, Carpenter MH, Lewis RM (1999) Low-storage, explicit Runge–Kutta schemes for the compressible Navier–Stokes equations. NASA/CR 1999-209349
28. Henshaw WD (2011) Cgins reference manual: an overture solver for the incompressible Navier–Stokes equations on composite overlapping grids. Lawrence Livermore National Laboratory Report LLNL-SM-455871, 2011
29. Crumpton PI, Moinier P, Giles MB (1997) An unstructured algorithm for high Reynolds number flows on highly stretched grids. In: Numerical methods in laminar and turbulent flow. Pineridge Press, Whiting, pp 561–572
30. Chandar D, Sitaraman J, Mavriplis DJ (2012) On the integral constraint of the pressure Poisson equation for incompressible flows on an unstructured grid. Int J Comput Fluid Dyn. doi:10.1080/10618562.2012.723127
31. NVIDIA GPUDirect Technology, Mellanox technologies white paper, http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf. Accessed 25 July 2012
32. Jones KD, Dohring CM, Platzer MF (1998) Experimental and computational investigation of the Knoller–Betz effect. AIAA J 36(7):1240–1246
33. Tuncer IH, Kaya M (2003) Thrust generation caused by flapping airfoils in a biplane configuration. J Aircr 40:509–515
34. Chandar D, Sitaraman J, Mavriplis DJ (2013) Overset grid based computations for rotary wing flows on GPU architectures. Presented at the American helicopter society forum, AHS69, May 2013