20th AIAA Computational Fluid Dynamics Conference
27 - 30 June 2011, Honolulu, Hawaii

AIAA 2011-3222

CU++ET: An Object Oriented Tool for Accelerating Computational Fluid Dynamics Codes using Graphical Processing Units

Dominic D.J. Chandar[*] , Jay Sitaraman[†]
and Dimitri Mavriplis[‡]

The application of graphical 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. CUDA has been applied previously to solve the Three-Dimensional Euler equations, and a speed-up of the order of 500 has been reported in literature using multiple GPU units. 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 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++ET 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. It allows the user to reuse existing C/C++ CFD codes with minimal changes. This approach is tremendously beneficial for CFD code development because it mitigates the necessity of creating hundreds of GPU *kernels* for various purposes. In its current form, CU++ET provides a framework for parallel array arithmetic, simplified data structures to interface with the GPU, and smart array indexing. Using this framework, a higher-order 3D Euler solver (ARC3D-GPU) has been developed with a performance improvement of about 70x on a single GPU compared to traditional FORTRAN/CPU execution. An implementation of heterogeneous parallelism, i.e., utilizing multiple GPUs to simultaneously process a partitioned grid system with communication at the interfaces using MPI has been developed and tested. An unstructured version of CU++ET is also demonstrated with its application towards solving the incompressible Navier-Stokes equations.

I. Introduction and Background

Graphical Processing Units (GPU) 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 has 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]. Hagen et al.[3] describes a Three-Dimensional Euler solver on the 7800GTX graphics card. A speed-up of 11.5 was observed on 530,000 points for a Raleigh-Taylor instability problem. Elsen et al.[4] 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

*Postdoctoral Research Associate, Department of Mechanical Engineering, University of Wyoming
†Assistant Professor, Department of Mechanical Engineering, University of Wyoming
‡Professor, Department of Mechanical Engineering, University of Wyoming

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] has 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 speedup of 496 using 32 GPUs on 6 Million points for a Two-Dimensional Euler calculation.

Writing codes that run on a GPU require 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 arises 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 (ET)[9], and leaves the user to use normal arithmetic expressions without having the need to manipulate *kernels*. The present framework allows users to reuse existing C++ codes without having to make major changes to the programming strategy.

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[10]. ARC3D is a 6^{th} 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[11], (2) GPUEULER unstructured code[12], and (3) GPUINS unstructured incompressible viscous code. In this paper, emphasis is given to the higher level interface (CU++ET), and how this has been used to develop the above set of codes.

II. A Comparison of CPU and CU++ET Programs

To start with, we will describe through simple examples, how CU++ET 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 a $N \times N$ grid, one can write a C++ version of the discretized equation as shown in listing 1. The CU++ET 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++ET 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. 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 it is very similar to any standard C/C++/FORTRAN code, and one does not need to know the basic functions of GPU to execute this piece of code.

Listing 1. C++ version of the Jacobi Iteration

```
//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++ET version of the Jacobi Iteration

```
// Index objects are used to represent the base and bound of the array
Index i(1,N-2), j(1,N-2);
// u is a distributed array object 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++ET source code for the 1-D Diffusion Equation

```
// 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++ET 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();
```

III. A Description of the CU++ET Framework

The CU++ET framework is heavily based upon vector array arithmetic using C++ classes, and template programming as in figure (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, has simple functions to transfer data between CPU and GPU, perform arithmetic operations and manages other classes. 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.

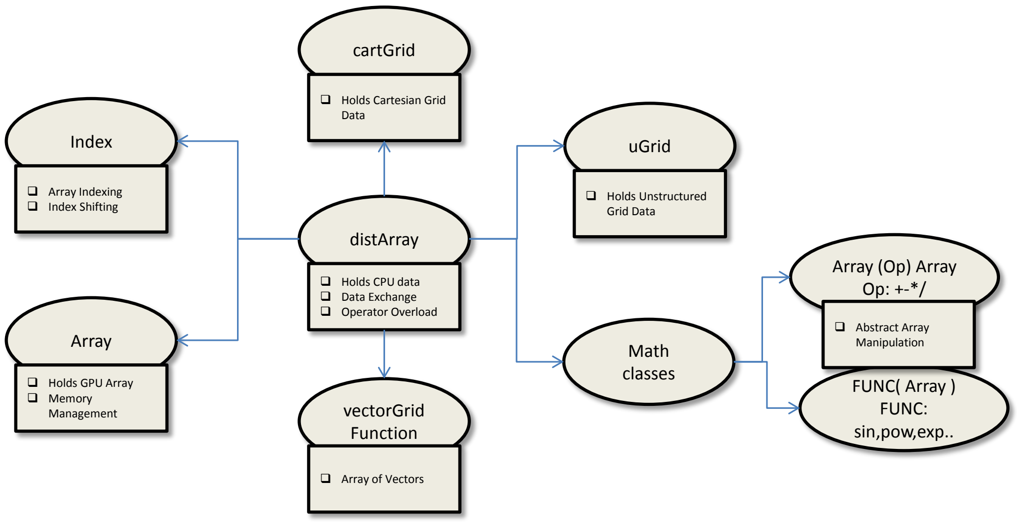


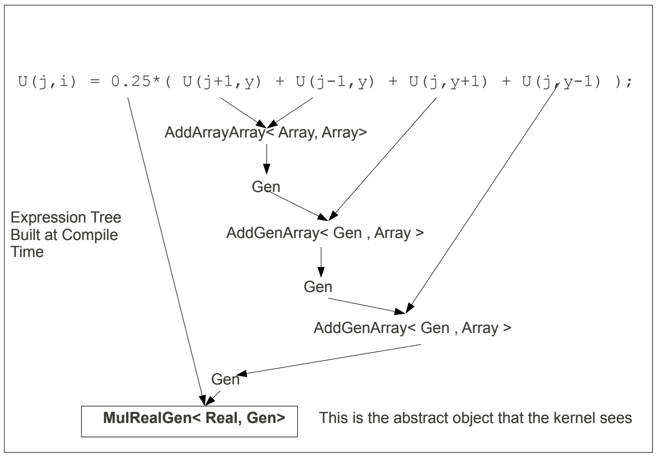
Figure 1. An Overall View of the CU++ET Classes

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. The class *Index* is one of the simplest of all the classes, and it mimics the *Index* class of the package $A++$[13], a serial array class used for vector arithmetic. This class is used to store the base and bound of a given array. For example, 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 listing 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. 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++ET 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 it's *type* is unknown when the code is written. During compile time, all arithmetic expressions in the code are bundled into abstract objects which are un-rolled at run time inside the GPU kernel. Figure (2) shows how this is achieved for the expression in listing 2. Each time when two *distArrays* are required to be added, it returns an abstract object of type *Gen* at compile time. Inside each abstract object's type definition, we overload the operator [ ] to be able to point to the desired array location. This operation is performed until the compiler hits the = symbol. We then write one kernel which will accept the abstract object, un-roll the individual components of this abstract object and perform the required operation. All vector expressions in this method call only one kernel, thus avoiding the necessity to write many kernels.

Listing 4. **Indexing for Unstructured Data**

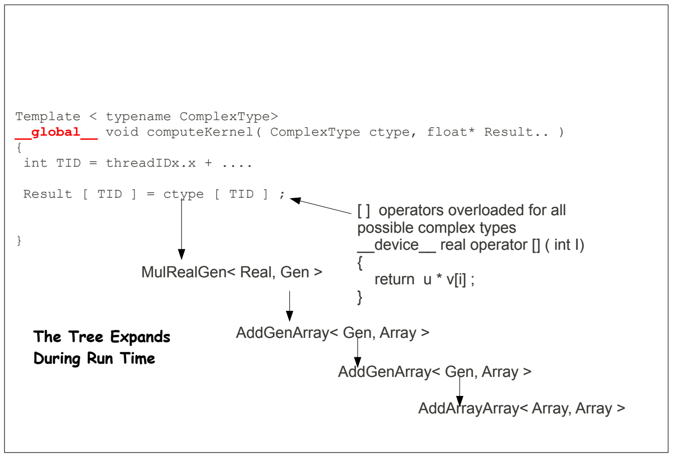
```
// Declare an array to hold the solution
distArray Q(number_of_nodes);

// 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));
```



```
U(j,i) = 0.25*( U(j+1,y) + U(j-1,y) + U(j,y+1) + U(j,y-1) );
```

AddArrayArray< Array, Array>
Gen
Expression Tree Built at Compile Time
AddGenArray< Gen , Array >
Gen
AddGenArray< Gen , Array >
Gen
MulRealGen< Real, Gen>
This is the abstract object that the kernel sees

(a)



```
Template < typename ComplexType>
__global__ void computeKernel( ComplexType ctype, float* Result.. )
{
 int TID = threadIDx.x + ....

 Result [ TID ] = ctype [ TID ] ;

}
```

```
[ ]  operators overloaded for all
possible complex types
__device__ real operator [] ( int I)
{
   return  u * v[i] ;
}
```

MulRealGen< Real, Gen >
The Tree Expands During Run Time
AddGenArray< Gen, Array >
AddGenArray< Gen, Array >
AddArrayArray< Array, Array >

(b)

Figure 2. (a) Building the Abstract Object during Compilation (b) Run time un-rolling of the Abstract Object

IV. Computational Modeling

IV.A. Compressible Flow

To begin with, the current implementation is tested by solving the Three-Dimensional Euler Equations in a box, following the ARC3D framework. The governing equations for 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:

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

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.[14].

IV.B. Incompressible Flow

We also demonstrate the application of this framework towards solving unsteady incompressible flow on unstructured grids. The compressible version has been discussed previously in Soni et al.[12] The equations governing unsteady incompressible flow with moving grid terms are given by:

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

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

where U is the velocity vector, P is the pressure normalized by density, and U_G is a vector of grid speeds. We use the Pressure-Poisson formulation (PPE) of Henshaw[15], where the divergence constraint Eq.(12) is replaced by a Pressure-Poisson equation by taking the divergence of the momentum equation.

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

For the puporses of discretization, Eq.(11) is written in conservative form for each node i and discretized in a finite-volume framework (Fig 3) as follows:

$$V_i \frac{\partial U_i}{\partial t} + \sum_k F.ndS_k = \nu \sum_k \nabla U.ndS_k \tag{14}$$

where k represents the dual face index, and 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:

$$F_k = \frac{1}{2}(F_{e1} + F_{e2}) \tag{15}$$

$$\nabla U_k = \nabla \bar{U_k} - \left(\nabla \bar{U_k} \cdot \delta_{12} - \frac{U_{e1} - U_{e2}}{|x_{e1} - x_{e2}|} \right) \delta_{12} \tag{16}$$

where

$$\delta_{12} = \frac{x_{e1} - x_{e2}}{|x_{e1} - x_{e2}|} \tag{17}$$

$\nabla \bar{U_k}$ 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[16]. 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

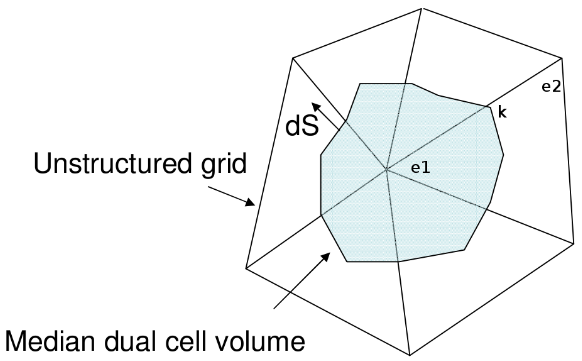


Figure 3. A Portion of the Unstructured Grid showing the Dual Cell

treated explicitly, and the viscous terms implicitly as described in Henshaw[15].

IV.C. GPU Implementation

Implementation in the case of a structured grid follows closely the listing (2),(3). For example, to compute the derivatives $\frac{\partial E}{\partial x}$ using Eq.(4),(5), and Eq.(8), listing (5) shows how this is achieved.

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)) ;
 }
```

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 GPU kernel, which is then unrolled into individual components at run time. For unstructured grids however, since the algorithm is not so straightforward, not all parts of the code are written using the CU++ET format, although we retain the same data structures like *distArray*. 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++ET framework, as they are easily translated to generic kernels.

V. Results and Discussions

V.A. Compressible Flow Single GPU Computations

The inviscid convection of a vortex is used as a test problem to validate the ARC3D-GPU solver on the Fermi (Tesla C2050) using single precision arithmetic. 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:

$$\vec{U}(\vec{r}) = \frac{\Gamma}{2\pi h}(1 - e^{-\sigma h^2})\hat{e}_\theta \tag{18}$$

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 orthogonal to the plane containing $\vec{r}$ and the axis of the vortex. The vortex is initially placed in a box $(0 \le x \le 6, 0 \le y \le 6, 0 \le z \le 0.6)$at a location of (3,3) as given in figure 4. Using a grid size of $200 \times 200 \times 150$ and a sixth order spatial differencing scheme for the

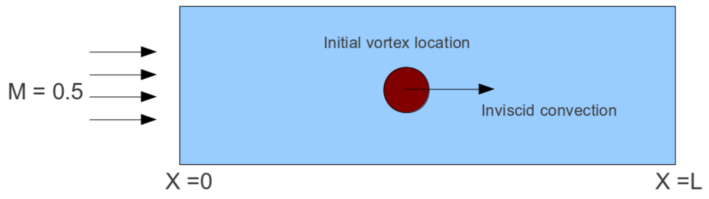


Figure 4. Problem setup for the Lamb vortex propagation on a single GPU

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 5 shows the contours of density at $t = 0$, and after 200 time steps.

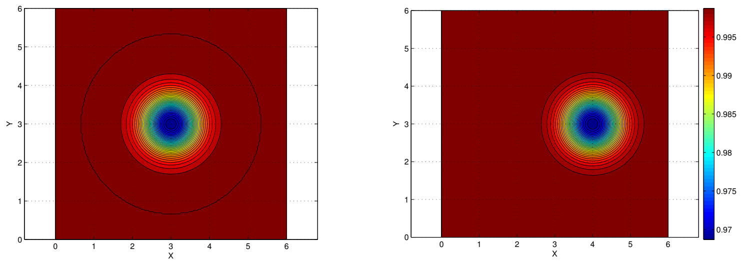


Figure 5. Density contours at (a) t = 0 and (b) t = 200 time steps

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

Table 1. Timing Comparisons for the Inviscid Vortex Convection Problem

Grid Points	Solver	CPU time per Time Step (s)
6 Million	ARC3D(FORTRAN)	50
6 Million	CU++ET	0.7

V.B. Compressible Flow Multi-GPU Computations using MPI

When there are multiple GPUs present on a system, it is possible to use all of the GPUs by dividing the workload using the Message Passing Interface (MPI) standard. Each GPU comes with a feature index called the compute capability. For GPUs having compute capability 1.x (Eg. Tesla C1060), one can execute one *kernel* at any given instant of time on a single GPU, and for recent GPUs such as Tesla C2050, and C2070, up to 16 kernels can be executed concurrently. This implies, multiple MPI processes can map a single GPU simultaneously. However, we do not use this property at the moment, and map one GPU to one MPI process. We consider the same test case of an inviscid vortex convection, but on a bigger domain. Figure 6 describes the partition of the domain on two and six processes respectively. The vortex is initially placed in a box

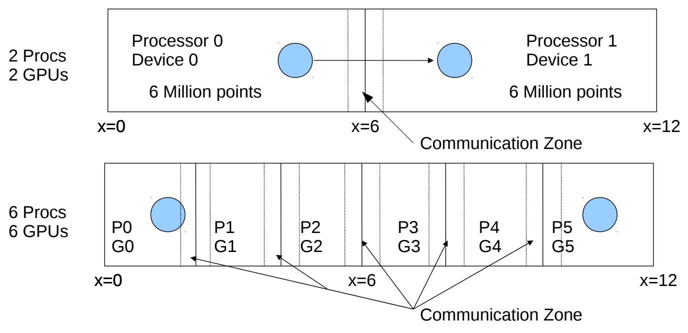


Figure 6. Problem setup for the Lamb vortex propagation using Multiple GPUs

($0 \leq x \leq 12, 0 \leq y \leq 6, 0 \leq z \leq 0.6$)at a location (4,3). The grid size for this problem is $400 \times 200 \times 150$.

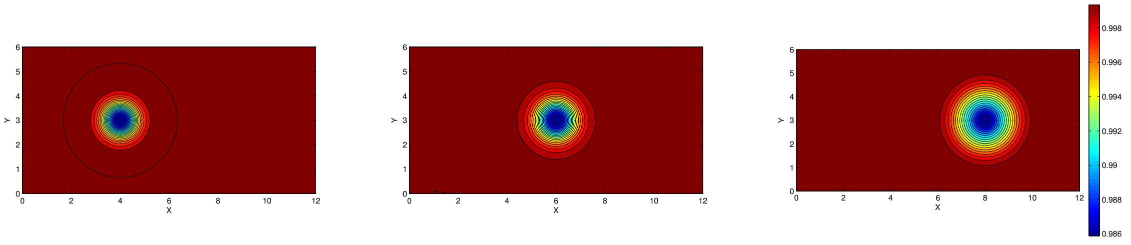


Figure 7. Density contours at time (a) t = 0, (b) t = 400 Time Steps and (c) 800 Time Steps using Multiple GPUs

Each partitioned chunk of the domain is mapped to a GPU device. 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 6^{th} 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. 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 data is copied back and forth between the CPU and GPU, hence perfect scalability might not be obtained. Figure 7 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. Table 2 shows a comparison of the CPU time spent with that of the previous approaches. It can be seen that, on a grid of 12 Million points with minimal overhead in the communication, we are able to execute the code in almost the same time as that of a grid with 6 Million points using two processors. However, due to the communication bottleneck in the six process case, perfect scalability is not obtained. With CUDA 4.0's GPU peer to peer memory access on a single node, this problem can partly be avoided[7].

Table 2. Timing Comparisons for the Inviscid Vortex Convection Problem with and without MPI

Grid Points	Solver	CPU time per Time Step (s)
6 Million	ARC3D(FORTRAN)	50
6 Million	CU++ET using one GPU	0.70
12 Million	CU++ET using one GPU	1.37
12 Million	CU++ET + MPI using two GPUs	0.80
12 Million	CU++ET + MPI using six GPUs	0.39

V.C. Incompressible Flow Computations

To validate the incompressible flow solver on unstructured grids, we consider two cases of a plunging NACA 0012, 0014 airfoil at Reynolds number 500, and 10000 respectively. For Re=500, current computations are compared with flow visualization results of Jones et al.[17], and for Re=10000, with the overset grid computations of Tuncer and Kaya[18]. A sinusoidal motion of the form $h = h_0 sin(\omega t)$ is prescribed for the airfoil. Table (3) shows various parameters for the two test cases. Solutions are computed for six

Table 3. 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

cycles of oscillation, and the corresponding vorticity contours for NACA0012 airfoil are shown in figure(8) in comparison with the flow visualization results of Jones et al.[17]. A satisfactory comparison is obtained in terms of the wake deflection. Computed drag coefficients are compared with those of Tuncer and Kaya[18], and are shown in figure(9). All computations using the unstructured grid framework were performed in double

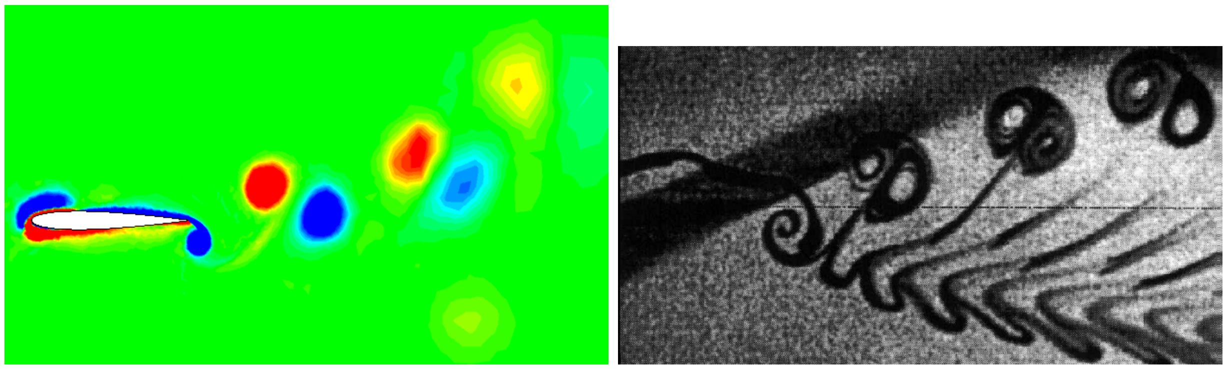


Figure 8. A Comparison of Flow Structures Behind a Plunging Airfoil between GPUINS Computation (left) and Experiments from Jones et al.[17] (right)

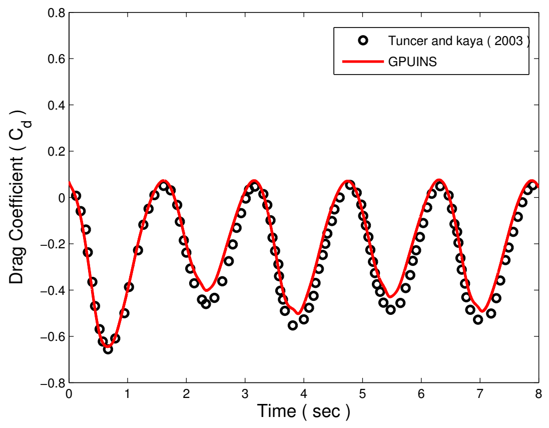


Figure 9. Time history of Drag Coefficient for a Plunging NACA0014 Airfoil at Re 10000

precision as opposed to single precision in the previous section. Over one time step, the serial version of the same code consumed 6.6s of CPU time, whereas the GPU code took 1.2s, resulting in 5.5x speed-up. Noting that GPU performance increases with the number of grid points[12], it is anticipated that three-dimensional problems will benefit to a greater extent, similar to the results from the previous section.

VI. 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 lies in the fact that *kernel* generation for vector operations with proper indexing is automatic, and is achieved at compile time using C++ expression templates. 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 CPU version of the same code. Using multiple GPU units, 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 4.0's peer to peer memory access, where one obviates the necessity to transfer data between CPU and GPU on a single 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++ET framework, and has proven to reproduce some of the results from available literature. In-depth validation, and integration of the above solvers using an overset grid framework is planned in the near future.

Acknowledgments

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

References

[1]Cohen, J.M., and Molemaker, M.J., *A Fast Double Precision Code using CUDA* , Proceedings of Parallel CFD, Moffett Field, CA, 2009.

[2]General-Purpose Computation on Graphics Hardware, *http : //gpgpu.org*

[3]Hagen, T.R., Lie, K-.,A and Natvig, J.R., *Solving the Euler Equations on Graphics Processing Units*, Lecture Notes in Computer Science, 3994, pp. 220-227, 2006.

[4]Elsen, E., LeGresley, P., and Darve, E., *Large Calculation of the Flow over a Hypersonic Vehicle using a GPU*, Journal of Computational Physics, Vol. 227, No. 24, pp. 10148-10161, 2008.

[5]Brandvik, T., and Pullan, G., *Acceleration of a 3D Euler Solver using Commodity Graphics Hardware*, 46^{th} AIAA Aerospace Sciences Meeting and Exhibit, AIAA-2008-0607, Reno, NV, 2008.

[6]Buck, I., *Data Parallel Computing on Graphics Hardware*, Graphics Hardware, 2003.

[7]NVIDIA CUDA C programming Guide 4.0, $http://developer.nvidia.com/cuda-toolkit-40$

[8]Phillips, E.H., Zhang, Y., Davis, R.L., and Owens, J.D., *Rapid Aerodynamic Performance Prediction on a Cluster of Graphics Processing Units*, 47^{th} Aerospace Sciences Meeting and Exhibit, AIAA-2009-0565, Orlando, FL, 2009.

[9]Vandevoorde, D., and Josuttis, N., *C++ Templates: The Complete Guide*, Pearson Education Inc, 2003.

[10]Pulliam, T. H. , *Euler and Thin Layer Navier Stokes Codes : ARC2D, ARC3D*, Computational Fluid Dynamics, University of Tennessee Space Institute, UTSI E02-4005-023-84, 1984.

[11]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., and Strawn, R., *Application of the Helios Computational Platform to Rotorcraft Flowfields*, 48^{th} AIAA Aerospace Sciences Meeting and Exhibit, AIAA-2010-1230, Orlando, FL, 2010.

[12]Soni, K., Chandar, D.D.J., and Sitaraman, J., *Development of an Overset Grid Computational Fluid Dynamics Solver on Graphical Processing Units*, 49^{th} AIAA Aerospace Sciences Meeting and Exhibit, AIAA-2011-1268, Orlando, FL.

[13]Quinlan, D., *A++P++ Manual*, Lawrence Livermore National Laboratory, UCRL Report No: UCRL-MA-136511, 2000.

[14]Kennedy, Chistopher A., Carpenter, Mark H., and Lewis, R. Michael., *Low-Storage, Explicit Runge-Kutta Schemes for the Compressible Navier-Stokes Equations*, NASA/CR 1999-209349, 1999.

[15]Henshaw, W.D., *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.

[16]Crumpton, P.I., Moinier, P., and Giles, M.B., *An Unstructured Algorithm for High Reynolds Number Flows on Highly Stretched Grids*, Numerical Methods in Laminar and Turbulent Flow, pp.561-572, Pineridge Press, 1997.

[17]Jones, K.D., Dohring, C.M., and Platzer, M.F., *Experimental and Computational Investigation of the Knoller-Betz Effect*, AIAA Journal, Vol. 36, No. 7, pp.1240-1246, 1998.

[18]Tuncer, I.H., and Kaya, M., *Thrust Generation Caused by Flapping Airfoils in a Biplane Configuration*, Journal of Aircraft, Vol. 40, pp.509-515, 2003.