# Hydrodynamic Computation with Hybrid Programming on CPU-GPU Clusters

Tingxing Dong*, Veselin Dobrev†, Tzanio Kolev†, Robert Rieben†,
Stanimire Tomov*, Jack Dongarra*

*Innovative Computing Laboratory, University of Tennessee, Knoxville
†Lawrence Livermore National Laboratory
*tdong, tomov, dongarra@eecs.utk.edu
†dobrev1,kolev1,rieben1@llnl.gov

## ABSTRACT

The explosion of parallelism and heterogeneity in today's computer architectures has created opportunities as well as challenges for redesigning legacy numerical software to harness the power of new hardware. In this paper we address the main challenges in redesigning BLAST – a numerical library that solves the equations of compressible hydrodynamics using high order finite element methods (FEM) in a moving Lagrangian frame – to support CPU-GPU clusters. We use a hybrid MPI + OpenMP + CUDA programming model that includes two layers: domain decomposed MPI parallelization and OpenMP + CUDA acceleration in a given domain. To optimize the code, we implemented custom linear algebra kernels and introduced an auto-tuning technique to deal with heterogeneity and load balancing at runtime. Our tests show that 12 Intel Xeon cores and two M2050 GPUs deliver a 24× speedup compared to a single core, and a 2.5× speedup compared to 12 MPI tasks in one node. Further, we achieve perfect weak scaling, demonstrated on a cluster with up to 64 GPUs in 32 nodes. Our choice of programming model and proposed solutions, as related to parallelism and load balancing, specifically targets high order FEM discretizations, and can be used equally successfully for applications beyond hydrodynamics. A major accomplishment is that we further establish the appeal of high order FEMs, which despite their better approximation properties, are often avoided due to their high computational cost. GPUs, as we show, have the potential to make them the method of choice, as the increased computational cost is also localized, e.g., cast as Level 3 BLAS, and thus can be done very efficiently (close to "free" relative to the usual overheads inherent in sparse computations).

## General Terms

Algorithms, Performance, Experimentation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, LLNL-CONF-607852 ...$5.00.

## Keywords

CFD, compressible hydrodynamics, hybrid programming, CUDA, OpenMP, finite element methods

## 1. INTRODUCTION

Currently, more and more computing clusters are equipped with different types of computational units. In particular, systems hosting both multicore CPUs and accelerators like GPUs are widely adopted in high-performance computing. The popularity of such heterogeneous architectures is largely due to the need for both high performance and power efficiency. GPUs are attractive due to their high floating-point operation capability and energy efficiency advantage over CPUs for highly data-parallel computations, while CPUs have advantage in sequential or just difficult to data-parallelize computations. This creates an opportunity for porting complex applications to hybrid architectures because by combining GPUs and CPUs one can design and schedule the computation in a way that exploits both the GPU and CPU strengths while avoiding their weaknesses. However, harnessing this potential is a challenge for developers. In this paper, we use a heterogeneous programming model to accelerate a hydrodynamic code named BLAST on heterogeneous CPU-GPU clusters.

BLAST is a research code developed at the Lawrence Livermore National Laboratory (LLNL) to solve the equations of compressible hydrodynamics using high order finite element methods (FEM). It supports arbitrary order space and time discretizations [1]. Like other computational fluid dynamics (CFD) applications, BLAST makes use of explicit time evolving iterations of computationally intensive kernels. The most floating point intensive part of BLAST is the construction of so called "corner force" matrices which can take up to 55%-80% of the total run time depending on the order of the finite element method and the spatial dimension of the problem. To reveal more refined physical features, high order numerical methods ($p$-refinement) and/or high resolution meshes ($h$-refinement) are introduced which inevitably lead to larger floating point operations and longer run times. For a given number of degrees of freedom, high order methods are more computationally intensive than low order methods since they couple each degree of freedom with more and more of its neighbors on the computational mesh. This means that high order methods perform many more floating point operations per memory access than low order

methods. The corner force kernel in BLAST is purely local and FLOP-intensive with relatively small I/O overhead. Our assumption is that these characteristics make BLAST suitable to accelerate on GPUs. We adopt a hybrid programming approach to implement such an acceleration on GPU clusters.

In recent years, MPI-GPU has been widely adopted to speedup applications. D.A.Jacobsen implemented an incompressible flow solver with MPI-CUDA on GPU clusters [2]. J.C.Thibault implemented incompressible flow computations with a Pthreads-CUDA approach [3]. N.Maruyama used MPI-CUDA for stencil computations [4]. J.Holewinski generalized a scheme for stencil computations on both NVIDIA and AMD GPUs [5]. However, most papers focus mainly on incompressible CFD benchmarks based on structured grids compatible with finite difference methods (FDM) which are inherently block structured in nature. Here, we consider the case of compressible CFD problems based on unstructured finite element methods (FEM) requiring the solution of both dense and sparse linear algebra problems.

In most previous MPI-CUDA applications, one CPU core takes control of only one GPU card in each MPI task, while the remaining CPU cores are ignored [6]. Because CPU cores are often much more than the number of GPUs in one node, this programming practice indicates that hardware resources are not fully exploited (see Section 4). To maximize usage of both multicore and GPUs, we use a hybrid MPI + OpenMP + CUDA programming model. In our model, the top level is still MPI based for inter-node communication. In the second level, we use a combination of CUDA + OpenMP to accelerate the corner force calculation using multiple cores with OpenMP and multiple NVIDIA GPUs with CUDA. To our knowledge, this is the first time that this complete model has been applied in a finite element Lagrangian hydrodynamics application.

## 2. THE BLAST ALGORITHM

The BLAST C++ code uses high order finite element methods in a moving Lagrangian frame to solve the Euler equations of compressible hydrodynamics. It supports 2D (triangles, quads) and 3D (tets, hexes) unstructured curvilinear meshes.

On a semi-discrete level, the conservation laws of Lagrangian hydrodynamics can be written as

$$\text{Momentum Conservation:} \quad \mathbf{M}_V \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = -\mathbf{F}\cdot\mathbf{1}, \qquad (1)$$

$$\text{Energy Conservation:} \quad \frac{\mathrm{d}\mathbf{e}}{\mathrm{d}t} = \mathbf{M}_{\mathcal{E}}^{-1}\mathbf{F}^T\cdot\mathbf{v}\,, \qquad (2)$$

$$\text{Equation of Motion:} \quad \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \mathbf{v}, \qquad (3)$$

where $\mathbf{v}$, $\mathbf{e}$, and $\mathbf{x}$ are the unknown velocity, specific internal energy, and grid position respectively. The kinematic mass matrix $\mathbf{M}_V$ is the density weighted inner product of *continuous* kinematic basis functions and is therefore global, symmetric and sparse. We solve the linear system of (1) using a preconditioned conjugate gradient (PCG) iterative method at each time step. The thermodynamic mass matrix $\mathbf{M}_{\mathcal{E}}$ is the density weighted inner product of *discontinuous* thermodynamic basis functions and is therefore symmetric and block diagonal, with each block consisting of a local dense matrix. We solve the linear system of (2) by pre-computing the inverse of each local dense matrix at the beginning of a simulation and applying it at each time step using dense linear algebra routines. The rectangular matrix $\mathbf{F}$, called the generalized *force matrix*, depends on the hydrodynamic state $(\mathbf{v}, \mathbf{e}, \mathbf{x})$ and needs to be evaluated at every time step.

The most computationally intensive part of the above algorithm is the evaluation of the matrix $\mathbf{F}$, which can be assembled from the generalized *corner force matrices* $\{\mathbf{F}_z\}$ computed in every zone (or element) of the computational mesh. Evaluating $\mathbf{F}_z$ is a locally FLOP-intensive process based on transforming each zone back to the reference element where we apply a quadrature rule with points $\{\hat{q}_k\}$ and weights $\{\alpha_k\}$:

$$(\mathbf{F}_z)_{ij} = \int_{\Omega_z(t)} (\sigma : \nabla\vec{w}_i)\,\phi_j$$

$$\approx \sum_k \alpha_k \hat{\sigma}(\hat{\vec{q}}_k) : \mathbf{J}_z^{-1}(\hat{\vec{q}}_k)\hat{\nabla}\hat{\vec{w}}_i(\hat{\vec{q}}_k)\,\hat{\phi}_j(\hat{\vec{q}}_k)|\mathbf{J}_z(\hat{\vec{q}}_k)|. \qquad (4)$$

The generalized force matrix $\mathbf{F}$ is constructed by two loops: an outer loop over zones (for each $z$) in the domain and an inner loop over the quadrature points (for each $k$) in each zone. Each zone and quadrature point computes a component of the corner forces associated with it independently.

A local corner force matrix $\mathbf{F}_z$ can be written as

$$\mathbf{F}_z = \mathbf{A}_z\mathbf{B}^{\mathbf{T}},$$

with

$$(\mathbf{A}_z)_{ik} = \alpha_k \hat{\sigma}(\hat{\vec{q}}_k) : \mathbf{J}_z^{-1}(\hat{\vec{q}}_k)\hat{\nabla}\hat{\vec{w}}_i(\hat{\vec{q}}_k)\,|\mathbf{J}_z(\hat{\vec{q}}_k)| \qquad (5)$$

and

$$(\mathbf{B})_{jk} = \hat{\phi}_j(\hat{\vec{q}}_k)\,. \qquad (6)$$

The matrix $\mathbf{B}$ contains the values of the thermodynamic basis functions sampled at quadrature points on the reference element $\hat{\phi}_j(\hat{\vec{q}}_k)$ and is of dimension number of thermodynamic basis functions by number of quadrature points. The values stored in the matrix $\mathbf{B}$ are constant in time. The matrix $\mathbf{A}_z$ contains the values of the gradient of the kinematic basis functions sampled at quadrature points on the reference element $\hat{\nabla}\hat{\vec{w}}_i(\hat{\vec{q}}_k)$ and is of dimension number of kinematic basis functions by number of quadrature points. This matrix also contains terms which depend on the geometry of the current zone, $z$, as well as the stress values $\hat{\sigma}(\hat{\vec{q}}_k)$ which require evaluation at each time step and involve significant amounts of computation including singular value decomposition (SVD), eigenvalue, eigenvector, equation of state (EOS) evaluations, etc. at each quadrature point (see [1] for more details).

A high order finite element solution is specified by choosing the order of the kinematic and thermodynamic bases. In practice, we choose the order of the thermodynamic basis to be one less than the kinematic basis, where a particular method is designated as $Q_k$-$Q_{k-1}$, $k \geq 1$, corresponding to a continuous kinematic basis in the space $Q_k$ and a discontinuous thermodynamic basis in the space $Q_{k-1}$. High order methods (as illustrated in Figure 1) can lead to better numerical approximations at the cost of more basis functions and quadrature points in the evaluation of (2). By increasing the order of the finite element method, $k$, used in a given simulation, we can arbitrarily increase the floating point intensity of the corner force kernel of (2) as well as the overall algorithm of (1) - (3).

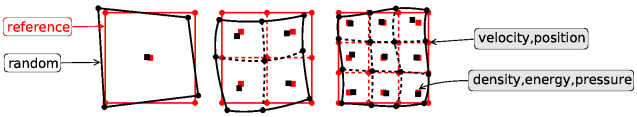


**Figure 1: Schematic depiction of bilinear ($Q_1$-$Q_0$), biquadratic ($Q_2$-$Q_1$), and bicubic ($Q_3$-$Q_2$) zones.**

Here we summarize the basic steps of the overall BLAST MPI-based parallel algorithm:

1. Read mesh, material properties and input parameters;
2. Partition domain across MPI tasks and refine mesh in parallel if necessary;
3. Compute initial time step;
4. Loop over zones in the subdomain of each MPI task:
   (a) Loop over quadrature points in each zone;
   (b) Compute corner force associated with each quadrature point and update time step;
5. Assemble zone contribution to global linear system;
6. Solve global linear system for new accelerations;
7. Integrate accelerations in time to get velocities and new mesh positions;
8. Update internal energies due to hydrodynamic motion;
9. Go to 4 if final time is not yet reached, otherwise exit.

Step 4 is associated with the corner force calculation of (2). It takes at least 55% of the total time and is therefore a computational hot spot where we focus our effort. Step 6 requires solving the linear equation (using a simple PCG solver) of (1) which takes about 25% of total run time. In Table 1 we show timing data for each of these kernels for various high order methods in 2D and 3D. In general, the computational work of each of these kernels increases as the order of the method $k$ is increased and also substantially increases for 3D computations.

| Method | Corner Force | CG Solver | Total time |
|---|---|---|---|
| 2D: $Q_4$-$Q_3$ | 198.6 | 53.6 | 262.7 |
| 2D: $Q_3$-$Q_2$ | 72.6 | 26.2 | 103.7 |
| 3D: $Q_2$-$Q_1$ | 90.0 | 56.7 | 164.0 |

**Table 1: Profile of BLAST: The corner force kernel consumes 55%-75% of total time. The CG solver takes 20%-34%. Measurements are based on three hundred time step iterations. Time is in seconds.**

# 3. HYBRID PROGRAMMING MODEL

Multi-GPU communication relies on CPU-GPU communication on a single node and CPU-CPU communication across nodes. Therefore, a multi-GPU implementation requires CUDA to interact with other CPU programming models like MPI, OpenMP or Pthreads. Our implementation is composed of the following two layers of parallelism:

- MPI-based parallel domain-partitioning and communication between CPUs
- CUDA and OpenMP based parallel corner force calculation inside each MPI task

To facilitate development and code maintainability, we take a modular approach. Both CUDA and OpenMP can be independently enabled in BLAST. In fact, the work described in this paper represents a combination of research efforts that started with a CUDA implementation, continued with an OpenMP module, and completed with the merge of these two approaches. In the following sections, we detail the implementation.

## 3.1 CUDA Implementation

Since the corner force matrix $\mathbf{F}$ is large and memory transfer between the host CPU and the GPU via PCI-E is relatively slow, we implement the entire right hand sides of the momentum (1) and energy (2) equations on the GPU to avoid transferring the full matrix $\mathbf{F}$, and instead transfer only its action via input/output vectors. In the CUDA programming guide [7], the term host is used to refer to CPU and device to GPU. Hereafter in this paper, we follow this practice.

### *3.1.1 CUDA Kernels*

The CUDA implementation is composed of the following set of kernels:

1. **Kernel 1** loops over quadrature points of all zones. This kernel computes $\mathbf{A}_z$ from (5) based on the current state data $(\mathbf{v}, \mathbf{e}, \mathbf{x})$ which is transferred from the host before the kernel is launched. Each thread works on one quadrature point and computes a column of the matrix $\mathbf{A}_z$. The number of quadrature points depends on the order of the finite element method, $k$, but is usually far below 1024, the maximum size of thread blocks under CUDA computing capability 2.x. Therefore, multiple zones can fit in the same thread block. The number of zones residing on each thread block is tunable. This kernel requires a significant amount of floating point operations, as each thread involves Level-3 BLAS routines like SVD, eigenvalue and eigenvector calculations for a $2\times2$ (for 2D problems) or $3\times3$ (for 3D problems) matrix. Because there is no way for each thread to call standard linear algebra library routines like MAGMA [8] inside of CUDA kernels, we hand code these routines.
2. **Kernel 2** loops over zones. One thread block works on one zone. Each zone (thread block) does a matrix-matrix transpose multiplication $\mathbf{F}_z = \mathbf{A}_z\mathbf{B}^\mathbf{T}$ (DGEMM). Therefore, this kernel can be expressed as a batched DGEMM, with the number of batches be number of zones. Because the matrix $\mathbf{B}$ is constant in time as described in Section 2, it can be stored in read only memory. Consistent with CUBLAS [9] and MAGMA, the matrices are stored in column-major order. To maximize memory coalescing, each thread in a thread block computes one row of the matrix $\mathbf{F}$ as shown in Figure 2.
3. **Kernel 3** computes $\mathbf{F}\cdot\mathbf{1}$ from (1) and either returns the result to the CPU or remains on the GPU depending on whether or not our custom CUDA-PCG solver is

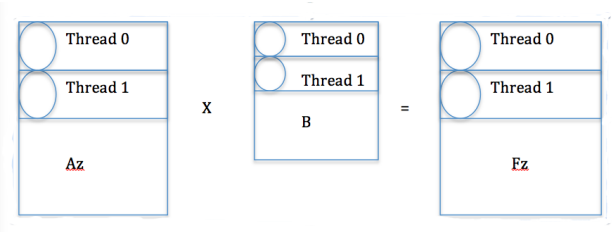


**Figure 2: Each thread block performs the operation $\mathbf{F}_z = \mathbf{A}_z\mathbf{B}^\mathbf{T}$. Matrices are stored in column-major order. Threads read and write row-wise to ensure memory coalescing. Reading $\mathbf{A}_z$ and B can be accelerated via shared memory and constant memory respectively.**

enabled. In this kernel, each thread block does a small matrix-vector multiplication (DGEMV) and computes part of a big vector. All thread blocks assemble the result vector. This kernel can be expressed as batched DGEMV.

4. **Kernel 4** is not a single kernel function in CUDA. It is a custom conjugate gradient solver for (1) with a diagonal preconditioner (PCG) [12]. It is constructed with CUBLAS/CUSPARSE routines [10]. When enabled, this custom CUDA solver performs step 6 from Section 2 and is outside of the corner force calculation. Currently, it only runs on a single GPU (see discussion in Section 3.4. The result vector $\frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t}$ is transferred back to the CPU and used in next time step if time is not out.

5. **Kernel 5** computes $\mathbf{F}^T \cdot \mathbf{v}$ from (2). Every thread block does a matrix transpose vector multiplication (DGEMV) similar to kernel 3. Again, this kernel can be expressed as batched DGEMV.

6. **Kernel 6** is a sparse (CSR) matrix multiplication to compute $\mathbf{M}_\mathcal{E}^{-1}(\mathbf{F}^T \cdot \mathbf{v})$ by calling a CUSPARSE SpMV routine [10]. The reason for calling SpMV routine instead of using a PCG solver as in kernel 4 is that the matrix $M_\mathcal{E}$ is block diagonal as described in Section 2. The inverse of $M_\mathcal{E}$ is only computed once at the initialization stage. The result vector $\frac{\mathrm{d}\mathbf{e}}{\mathrm{d}t}$ is transferred back to the CPU and used in next time step.

### 3.1.2 CUDA Memory Transfer Overhead

Input vectors are transferred from the host to the device before kernel 1, and output vectors are transferred back from the device to the host after kernels 3,4 and 6. Because the data size is relatively small (i.e. vectors instead of matrices) and the floating point intensity of the kernels is quite large, the memory transfer overhead is minor compared to the kernel running time. The CUDA profiler confirms it as shown in Figure 3.

### 3.1.3 CUDA Code Optimization

We use the CUDA profiler to identify performance bottlenecks of our CUDA kernels. Various techniques were used to optimize kernel 1. We use auto tuning to find the optimal number of zones for each thread block in kernel 1 as shown in Table 5. Details of auto tuning will be discussed in Section 3.3.

Step by step optimization of kernel 2. At the beginning, we expected kernel 2 to be faster than kernel 1 which has more FLOPs than kernel 1. However, the CUDA profiler showed that our first version kernel 2.1 dominated GPU run time. The reason is we did not make good use of bandwidth of GPU. In kernel 2.1, the matrices $\mathbf{A}_z$ and $\mathbf{B}$ were loaded directly from global memory. In kernel 2.2, we take advantage of the memory hierarchy. Shared memory is used to accelerate reading of $\mathbf{A}_z$ and constant memory is used to accelerate reading of $\mathbf{B}$, because $\mathbf{A}_z$ is local per thread block and $\mathbf{B}$ is globally shared by all thread blocks. Compared to kernel 2.1, kernel 2.2 is a substantial improvement, but still not satisfactory. An alternative implementation is to call `cublasDgemmbatched` [9] as shown in Figure 4. However, `cublasDgemmbatched` is general purpose since it assumes each $\mathbf{A}_z$ multiplies a different $\mathbf{B}$. Our custom DGEMM version of kernel 2 is application specific and we believe there is still space to improve based on CUBLAS. As a further optimization, kernel 2.3 uses blocking techniques. Blocking is the process of dividing a large matrix into smaller matrices to solve. Blocking is widely adopted in LAPACK. The main purpose of blocking is to increase data locality and thus improve cache performance. On GPUs, blocking can deliver a second benefit: reducing the amount of shared memory requirement for each thread block and allow more thread blocks to reside on streaming multiprocessors and thus enhance the parallelism. Blocking can be done in different patterns. We found that accessing columns in blocks by 1D dimension proved to be most effective. However, the number of rows of $\mathbf{A}_z$ is not always multiple of a warp size (32). Loading one column from global memory to shared memory(as shown in Figure 2) may be not coalesced, since consecutive warp may not start from aligned address. To avoid this potential penalty, we pad column vectors of $\mathbf{A}_z$ with zeros when the row size is not divisible by 32 in kernel 2.4. Surprisingly, this benefit turns out to be very small as shown in Figure 4. The reason is starting from computing capability 2.x devices, the alignment requirement is not as restricted as before [7]. Considering padding will incur some storage overhead, thought it is small, we favor kernel 2.3 in our code. Finally, we are able to achieve 70-80 Gflops on one M2050, about 1.3-2 times faster than CUBLAS routines.

Optimization of kernel 1. Unlike kernel 2, it is hard to explore shared memory in kernel 1, because threads seldom share data. Instead, each thread works on an individual workspace allocated in global memory. This can be done by allocating a big array and splitting a chunk to each thread. The workspace is allocated just once and can be rewritten each time kernel 1 is called. SVD, eigenvalue and eigenvector calculations are all performed in the workspace.

Because neighboring threads (in one warp) not only execute together but also access data consecutively in workspace, good data locality is achieved both in time and space. Under this circumstance, we favor larger L1 cache to cache data, that is 48KB L1 cache vs 16KB shared memory, in kernel 1. While kernel 2 needs substantial shared memory, we configure to favor larger shared memory, that is 48KB of shared memory vs. 16KB of L1 cache.

Table 2 shows the performance of kernel 1 under two configurations. From the table, we can see that a configuration of 48KB L1 cache is clearly better than the other, especially

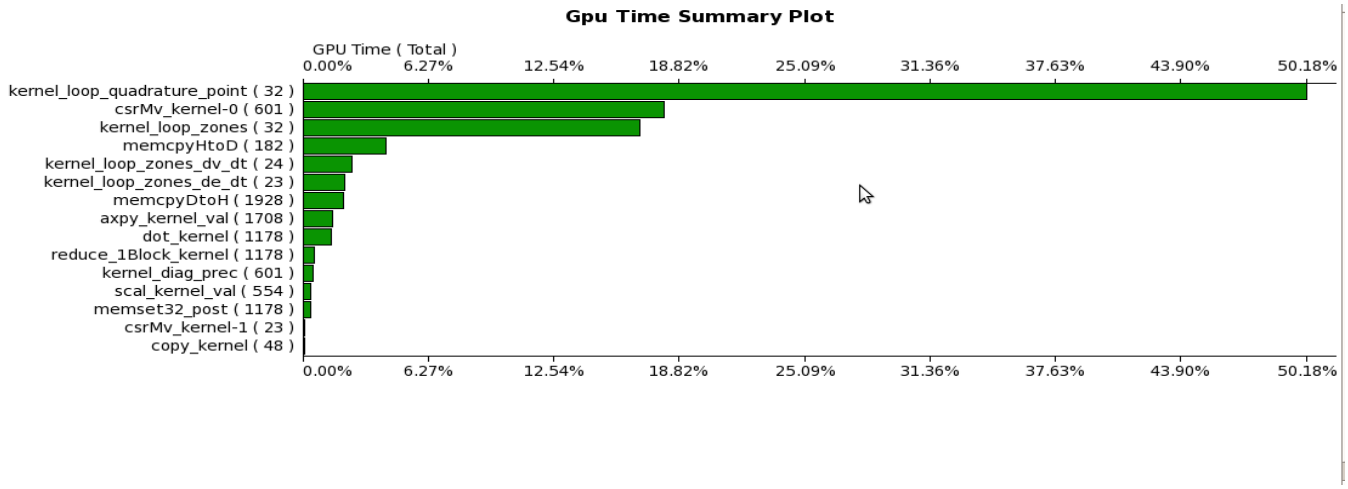


**Figure 3: CUDA Profiler shows a high ratio of kernel time to memory transfer in the GPU implementation. In the figure, loop_quadrature_point corresponds to kernel 1. Loop_zone corresponds to kernel 2. loop_zone_dv_dt and de_dt corresponds to kernel 3 and 5, respectively. csrMv_kernel-1 is kernel 6. Other routines are parts of kernel 4 (PCG solver). Notice csrMv_kernel-0 is in PCG solver. Numbers in bracket are times the routine is called during the profiling period.**

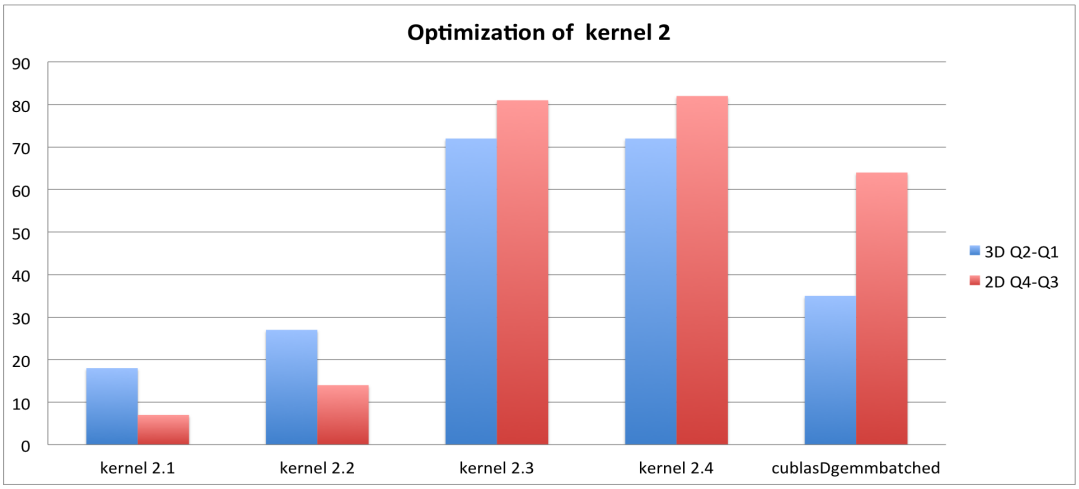


**Figure 4: Custom kernel 2.x and `cublasDgemmbatched` implementation of batched $A_z B^T$ on M2050. Only global memory is used in kernel 2.1. Kernel 2.2 load the entire matrix $A_z$ into shared memory and uses constant memory to read B. Blocking technique is used in kernel 2.3. Padding is added in kernel 2.4. Two test cases are performed. The left blue one is a 3D Q2-Q1 case with $A_z$ 81*64 and B 8*64, and the right red one is 2D Q4-Q3 with $A_z$ 50*64 and B 16*64. Number of batches is 4096 and 5376 respectively.**

for higher order methods with more quadrature points.

| Method | Config 1 | Config 2 |
|---|---|---|
| $Q_2$-$Q_1$ | 0.32 | 0.34 |
| $Q_3$-$Q_2$ | 0.90 | 1.05 |
| $Q_4$-$Q_3$ | 2.14 | 3.13 |

**Table 2: Performance of kernel 1 under two configurations. Config 1 is with 48KB L1 cache and less shared memory and Config 2 is with 16KB L1 cache but more shared memory. Time is in milliseconds**

There is a global reduction related to kernel 1 to find the minimum time step (Step 4.b from Section 2) because of the CFL time step condition. However, a reduction operation on the GPU is expensive and requires more than one kernel (without using atomic operation), because every thread needs to cooperate to reduce to one block and then every block cooperates to reduce again. We therefore implement this reduction on the CPU.

Kernel 3 and 5 can be translated into batched DGEMV. Unfortunately, CUBLAS does not provide batched DGEMV until the latest version 5.0. An alternative is to call a standard DGEMV routine number of zones times. Yet, the performance is very bad, as the number of zones is huge and the overhead of function calls overwhelmed any potential benefits. Finally, we program from scratch instead of calling any library. In our custom kernel, each thread block (zone) does

| Problem | Serial | 1 OMP Thread | Overhead |
|---|---|---|---|
| 3D: Sedov | 51.5 | 51.9 | 0.7% |
| 2D: Triple-pt | 17.7 | 17.9 | 1.1% |

**Table 3: Overhead of memory allocation and de-allocation in an OpenMP thread. Measurements are based on three hundred iterations. Time is in seconds**

a DGEMV operation. As shown in CUDA profiler, the two kernels are very small compared to kernel 1 and 2. This is not surprising, since DGEMV is a BLAS Level 2 routine.

Yet, to build our PCG solvers of kernel 4, we still call routines in CUBLAS and CUSPARSE as shown in Figure 3. We assume libraries to be stable. Kernel 6 is a sparse (CSR) matrix multiplication routine in CUSPARSE. This SpMV routine is also needed in kernle 4. From Figure 3 We can see the performance of SpMV is critical to the PCG, since it is the biggest one of PCG.

## 3.2 CUDA + OpenMP Implementation

Because a GPU runs asynchronously with the CPU, control can return to a host thread prior to the GPU completing work. It is equally important to make sure that the CPU cores are not idle while the GPU is working, especially considering that the number of cores is far more than the number of GPU cards in a typical heterogeneous cluster. In terms of the corner force calculation, since each zone computes independently, we split the work load by zones. A portion of the zones on a given domain are distributed to CPU cores instead of the entire set of zones going to the GPU.

After the launch of CUDA kernels, the host thread will spawn OpenMP threads and distribute the remaining zones among the threads. Each thread allocates private working space and executes like normal serial code. There is no synchronization between threads unless they exit the parallel region. Upon exit, the memory associated with each thread will be freed. Because the corner force routine is called repeatedly, there is additional overhead of memory allocation and de-allocation introduced by OpenMP compared to the serial code. Table 3 shows this overhead is small by comparing 1 OpenMP thread to the serial code.

A synchronization between the CPU and the GPU is required to complete the corner force calculation because there is data dependency in the following code. A key question is how to find an optimal ratio of work load between CPU and GPU, so none of them becomes idle while waiting for the other. We propose auto tuning to find this ratio.

## 3.3 Auto tuning

Like other CFD applications, BLAST makes use of explicit time evolving iterations. Our auto tuning technique is able to find the optimal load balancing parameters in the code by taking advantage of the time stepping in BLAST.

We take the auto balance of CPU and GPU as an example to describe the idea of auto tuning. First, the work load (zones) is distributed among CPU and GPU in an arbitrary ratio, usually half to half. In each iteration, GPU kernels and CPU threads are timed separately. After each sampling period, if the execution time ratio is outside the interval we set, say $[0.9 - 1.1]$, a scheduler simply reassigns zones (say 10% of the zones, we call this the 10% offset) to the one that finished faster.

After a few sampling periods, the scheduler will finally converge to an optimal ratio. Our tests shows that the convergence only takes a few time periods as shown in Table 4. Sometimes, the ratio fluctuates and does not converge. In that case, reducing the offset or relaxing the interval will help to speedup the convergence rate.

| Problem | Optimal ratio | Convergence period |
|---|---|---|
| 3D: Sedov | 0.45 | 3 |
| 2D: Sedov | 0.75 | 14 |
| 2D: Triple-pt | 0.77 | 12 |

**Table 4: The optimal ratio refers to the percentage of all zones assigned on the GPU. The starting tentative ratio is 0.5, and the target execution time ratio interval is $[0.9 - 1.1]$. One sampling period consists of forty iterations.**

Considering load balance is vulnerable to almost every outside factor including the number of CPU cores to GPUs, the order of numerical method, domain size and dimension of testing case, auto balance is a convenient and robust practical tool. In particular, when the code is ported to to another platform, the changes will be detected and load will be rebalanced automatically. Furthermore, auto tuning is independent of the MPI load balance layer, since it is called inside each MPI task. Finally, our rebalancing technique is built upon small number of timing functions, so its overhead is minor.

In kernel 1, we use the same idea to tune the number of zones per thread block, as shown in Table 5 . In fact, this methodology can be extended to other adjustable parameters.

## 3.4 MPI Level Parallelism

The MPI level parallelism in BLAST is based on MFEM which is a modular C++ finite element library [11]. At the initialization stage (Step 2 in Section 2), MFEM takes care of the domain splitting and parallel mesh refinement as shown in Figure 5. Each MPI task is assigned a subdomain consisting of a number of elements (zones). Finite element degrees of freedom (DOFs) shared by multiple MPI tasks are grouped by the set (group) of tasks sharing them and each group is assigned to one of the tasks in the group (the master), see Figure 6. This results in a non-overlapping decomposition of the global vectors and matrices and typical FEM and linear algebra operations such as matrix assembly and matrix-vector product require communications only within the task groups.

After computing the corner forces, a few other MPI calls are needed to handle the translation between local finite element forms and global matrix / vector forms in MFEM (Step 5 in Section 2). An MPI reduction is used to find the global minimum time step.

| Zones per Block | Block Size | Time |
|---|---|---|
| 2 | 32 | 0.48 |
| 4 | 64 | 0.52 |
| 6 | 96 | 0.50 |

**Table 5: Tuning the number of zones per thread block in kernel 1. Kernel time is in milliseconds.**

Because computing the corner forces can be done locally, the MPI level and the CUDA/OpenMP parallel corner force level are independent. Each module can be enabled or disabled independently. However, the kinematic mass matrix $\mathbf{M}_V$ in (1) is global and needs communication across processors, because the kinematic basis is continuous and components from different zones overlap. The modification of MFEM's PCG implementation needed to enable the CUDA-PCG solver to work on multi-GPUs, is beyond the scope of the present work. Therefore, we only consider the CUDA-PCG solver for (1) on a single GPU.

**Figure 5: Parallel mesh splitting and parallel mesh refinement**

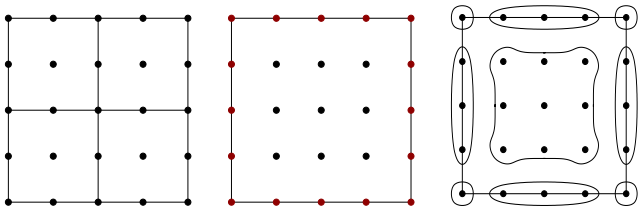

**Figure 6: Zones assigned to one MPI task and associated $Q_2$ DOFs (left); the DOFs at the boundary of this subdomain are shared with neighboring tasks (middle); groups of DOFs, including the local group of internal DOFs (right).**

## 4. TESTING CLUSTER

We target our hybrid implementation for the Edge cluster installed at LLNL which has 2 Intel 6 Core Xeon CPUs and 2 NVIDIA M2050 GPUs per node, with further details provided in Table 6. This is a typical GPU cluster architecture, since most GPU clusters are equipped with dozens of CPU cores but only a few high-end GPUs. Another GPU-accelerated cluster, open to the science community, is the NSF Keeneland system installed at ORNL [13]. The Keeneland Full Scale (KFS) system is a 264-node cluster based on HP SL250 servers. Each node has 32 GB of host memory, two Intel Sandy Bridge CPU's, three NVIDIA M2090 GPUs, and a Mellanox FDR InfiniBand interconnect.

| CPU | GPU | Mem/Node | Switch | Nodes |
|---|---|---|---|---|
| 6CoreX5660 | M2050 | 96GB | IB QDR | 216 |

**Table 6: Overview of the LLNL Edge cluster.**

Although a Fermi GPU can be shared by multiple MPI processes (in shared modes), multiple MPI tasks will cause false serialization across these tasks, because CUDA only allows one context to be active at one time [16]. It is similar to the restriction of one process running on one CPU core at one time. Normally this false sharing is not what users wish, since users want dedication instead of sharing. In most cases, the intent is to allow one process/thread to take control of only one GPU. In fact, most cluster administrators set GPUs on exclusive process mode to prevent sharing. This mode is set on the Edge cluster at LLNL.

In our setting, one GPU is dedicated to one MPI task. OpenMP is adopted to harness 6 CPU cores and the main OpenMP thread is used to schedule loads between the CPU and GPU, while a top MPI layer is used for inter-CPUs/GPUs communications, as shown in Figure 7.

| Problem | Method | MFEM PCG | CUDA PCG |
|---|---|---|---|
| 2D: Triple-pt | $Q_3$-$Q_2$ | 90.18 | 20.8 |
| 3D: Sedov | $Q_2$-$Q_1$ | 27.81 | 10.55 |

**Table 8: Time of CUDA-PCG compared to MFEM-PCG based 1000 iterations. Time is in seconds. The overhead of memory transfer between host and device is included in CUDA-PCG.**

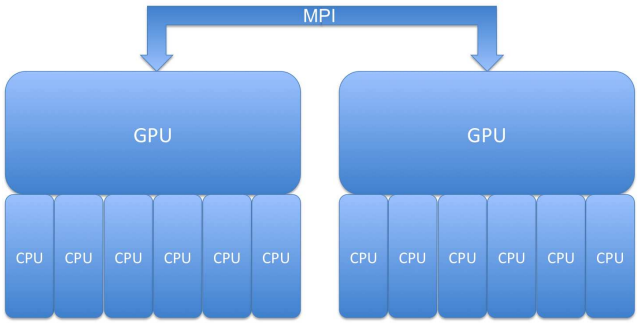


**Figure 7: MPI + CUDA + OpenMP hierarchy.**

## 5. TESTING RESULTS AND DISCUSSION

For our test cases we consider the 2D triple point problem using a $Q_3$-$Q_2$ method and the 3D Sedov blast wave problem using a $Q_2$-$Q_1$ method (see [1] for further details on the nature of these benchmarks). In all cases we use double precision. The Intel compiler and NVCC compiler under CUDA v4.2 are used for the CPU and GPU codes respectively.

### 5.1 Validation of CUDA code

We get consistent results on the CPU and the GPU. Both the CPU and the GPU code preserved the total energy of each calculation to machine precision, as shown in Table 7.

The density distributions of the 2D triple point problem in the CPU and CPU + GPU codes are shown in Figure 9 and Figure 10 respectively.

### 5.2 CUDA-only Acceleration of PCG

In Table 8 we compare our custom CUDA-PCG solver to our standard MFEM-PCG solver. Note that the CUDA-PCG solver outperforms the CPU based MFEM-PCG solver and achieved a maximum 4.3x speedup.

### 5.3 MPI + OpenMP + CUDA Acceleration of Corner Force

| Platform | Final Time | Kinetic | Internal | Total | Total Change |
|---|---|---|---|---|---|
| CPU | 0.6 | 5.0423596813598e-01 | 9.5457640318651e+00 | 1.005000000001e+01 | -9.2192919964873e-13 |
| GPU | 0.6 | 5.0418618040297e-01 | 9.5458138195986e+00 | 1.005000000002e+01 | -4.9382720135327e-13 |

**Table 7: Results of CPU and GPU for 2D triple-pt problem using a $Q_3$-$Q_2$ method; the total energy includes kinetic energy and internal energy. Both CPU and GPU results preserve the total energy to machine precision.**

Table 9 shows the performance of the corner force routine parallelized by different methods. From the table, we can see that one GPU can outperform the 6 core CPU in certain cases. Auto balancing distributes the ratio of CPU to GPU workload roughly according to the inverse of their running time. Speedups compared to the serial code are also shown in Figure 8. Here we see that 2 GPU & 12 Cores achieved a 24x and 2.5x speedup compared to 1 core and 12 cores, respectively for the triple-pt problem. In these tests, the sub-optimal kernel 2.2 was used. As discussed earlier, if the more advanced kernel 2.3 is used, the speedup will be even greater.

Table 10 shows that we get a perfect weak scaling of the triple-pt problem using a $Q_4$-$Q_3$ method with up to 64 GPUs.

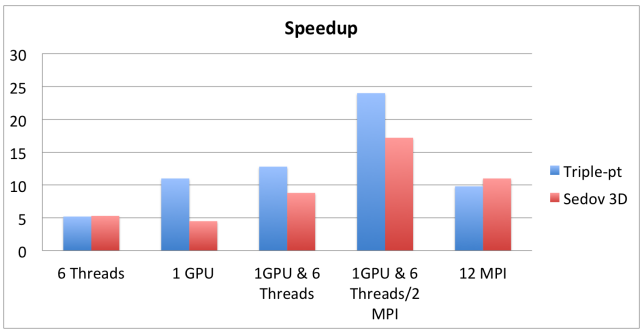


**Figure 8: Speedup of corner force compared to serial code**

## 6. LIMITATIONS

Currently, our custom PCG solver only works on a single GPU. Because the PCG Solver takes more than 20% percent and is not parallelized on multi-GPU, the overall speedup will be limited by Amdahl's law.

Unfortunately, there are a few CUDA math libraries that BLAST can directly use at present. In particular, BLAST can use a third party MFEM CPU code, but no CUDA-based counterparts. Until now, only a small number of CUDA math libraries are available. For example, CUSPARSE and CUSP deal with sparse matrix [14], and CUBLAS, MAGMA, and CULA deal with dense linear algebra [8] [15]. MAGMA is able to support multi-GPU in a single node right now, but CUSP, CUSPARSE, and CUBLAS only support single GPU. Although there is still some lack of CUDA software in the area of hydrodynamics computations, and others as well, the wide acceptance of GPUs for scientific computing is influencing developers to quickly close those gaps.

## 7. CONCLUSIONS AND FUTURE WORK

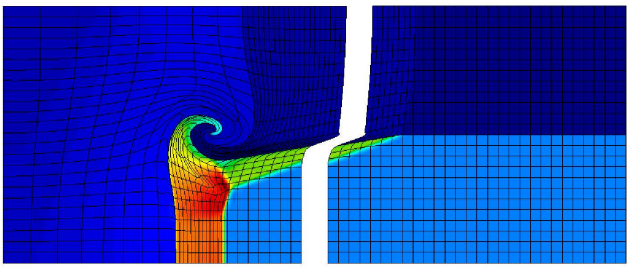

**Figure 9: Domain decomposition using 2 MPI tasks, each with 1 M2050 and 6 Xeon Cores.**

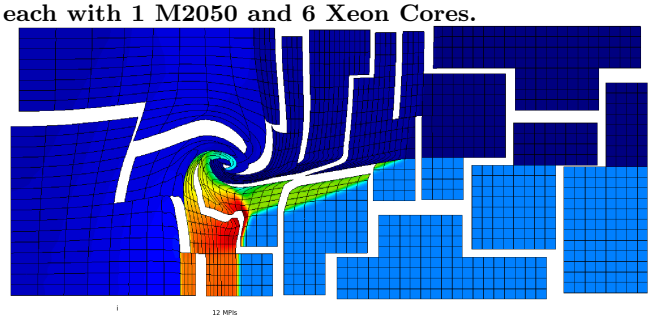

**Figure 10: Domain decomposition using 12 MPI CPU tasks.**

The BLAST code uses high order finite element methods to solve compressible hydrodynamics problems on a moving Lagrangian mesh and requires the computation and assembling of corner force matrices and the solution of both sparse and dense linear algebra problems. In this paper, we presented a hybrid programming model and a port of the BLAST code on a GPU/CPU cluster. OpenMP/CUDA was used to accelerate the corner force computation and MPI is used for communication. We use existing CUDA linear algebra library routines to construct our solvers and implement custom routines to optimize our code where the libraries are unable to achieve optimal performance. An auto tuning technique was introduced to maintain load balancing between multi-core CPU and GPU, and to tune CUDA kernels. The CPU/GPU results demonstrate consistency with the CPU only results and achieved a good speedup. Although GPU computing is still an area of research and key libraries are still lacking, our results demonstrate the potential of hybrid GPU computing.

In the future, we plan to move our code on Kepler platform not just because Kepler is a new GPU, but due to a key feature called hyper-Q introduced on Kepler [16]. Hyper-Q allows multiple CPU cores to launch work on a single GPU simultaneously, even each CPU core from different MPI processes. With this feature, the OpenMP implementation in our code can be removed without concerning about the idle

| Problem | Method | Serial | 6 Threads | 1 GPU | 1 GPU + 6 threads | 2 MPI x (1 GPU + 6 threads) | 12 MPI | GPU workload |
|---|---|---|---|---|---|---|---|---|
| 2D: Triple-pt | $Q_3$-$Q_2$ | 68 | 13.1 | 6.15 | 5.3 | 2.8 | 6.9 | 0.77 |
| 3D: Sedov | $Q_2$-$Q_1$ | 407 | 77 | 89 | 45.8 | 23.6 | 36.8 | 0.45 |

**Table 9: Performance of corner force routine with different parallel methods: Serial, GPU, OpenMP, MPI, GPU & OpenMP and GPU & OpenMP & MPI. Time is in seconds.**

| Number of GPUs | 1 | 4 | 16 | 64 |
|---|---|---|---|---|
| Mesh Size | 160*160 | 320*320 | 640*640 | 1280*1280 |
| Corner Force Time | 11.6 | 11.73 | 11.34 | 11.71 |

**Table 10: Perfect weak scaling of the 2D triple-pt problem using a $Q_4$-$Q_3$ method on multiple GPUs. Each GPU works on a fixed number of zones. Time is in seconds based on one hundred iterations.**

of CPU cores.

# 8. REFERENCES


[1] V.A.Dobrev, Tz.V.Kolev, R.N.Rieben. *High order curvilinear finite element methods for Lagrangian hydrodynamics*, SIAM J. Sci. Comp., 34(5), 2012, 606-641.

[2] D.A.Jacobsen, J.C.Thibault, I.Senocak. *An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters*, 48th AIAA Aerospace Sciences Meeting and Exhibit, 2010.

[3] J.C.Thibault, I.Senocak. *Accelerating Incompressible Flow Computations with a Pthreads-CUDA Implementation on Small-Footprint Multi-GPU Platforms*, The Journal of Supercomputing, 59(2), 693-719.

[4] N.Maruyama, T.Nomura, K.Sato, S.Matsuoka. *An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers*, SC11, 2011.

[5] J.Holewinski, L.Pouchet, P.Sandayappan. *High-performance code generation for stencil computations on GPU architectures*, ICS'12, Proceedings of the 26th ACM international conference on Supercomputing.

[6] L.Wang, W.Jia, X.Chi, Y.Wu, W.Gao, L.Wang. *Large Scale Plane Wave Pseudopotential Density Functional Theory Calculations on GPU Clusters*, SC11, 2011.

[7] NVIDIA CUDA C Programming Guide v4.2, http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation.

[8] MAGMA: http://icl.cs.utk.edu/magma/

[9] CUBLAS User Guide, http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation.

[10] CUSPARSE User Guide, http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation.

[11] MFEM: http://mfem.googlecode.com/

[12] M.Naumov, *Incomplete-LU and Cholesky Preconditioned Iterative Methods Using CUSPARSE and CUBLAS*, June 21, 2011.

[13] Keeneland: http://keeneland.gatech.edu/

[14] N.Bell, M.Garland. *Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations*, http://cusp-library.googlecode.com, 2012, version 0.3.0.

[15] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, E. J. Kelmelis. *CULA: Hybrid GPU Accelerated Linear Algebra Routines*, SPIE Defense and Security Symposium (DSS), April, 2010.

[16] Whitepaper: NVIDIA Next Generation CUDA Compute Architecture: Kepler GK110. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf