2014 IEEE 28th International Parallel & Distributed Processing Symposium

A Step towards Energy Efficient Computing: Redesigning A Hydrodynamic Application on CPU-GPU

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

Power and energy consumption are becoming an increasing concern in high performance computing. Compared to multi-core CPUs, GPUs have a much better performance per watt. In this paper we discuss efforts to redesign the most computation intensive parts of BLAST, an application that solves the equations for compressible hydrodynamics with high order finite elements, using GPUs [10, 1]. In order to exploit the hardware parallelism of GPUs and achieve high performance, we implemented custom linear algebra kernels. We intensively optimized our CUDA kernels by exploiting the memory hierarchy, which exceed the vendor's library routines substantially in performance. We proposed an autotuning technique to adapt our CUDA kernels to the orders of the finite element method. Compared to a previous base implementation, our redesign and optimization lowered the energy consumption of the GPU in two aspects: 60% less time to solution and 10% less power required. Compared to the CPU-only solution, our GPU accelerated BLAST obtained a 2.5× overall speedup and 1.42× energy efficiency (greenup) using $4th$ order (Q_4) finite elements, and a 1.9× speedup and 1.27× greenup using $2nd$ order (Q_2) finite elements.

Keywords

GPU, hydrodynamics, Power, Energy, FEM

1. INTRODUCTION

High performance computing is increasingly becoming power and energy constrained. The average power of TOP 10 supercomputers climbed from 3.2MW in 2010 to 6.6MW in 2013, which is enough to power small towns [2]. DOE has recently set a goal of 20MW for exascale systems, which means 50 GFLOPS per watt; though the current No.1 supercomputer Tianhe-2 has already reached 17MW at 0.03 EFLOPS. Limited by the power budget, more and more computing clusters seek to install accelerators, such as GPUs, due to their high floating-point operation capability and energy efficiency advantage over CPUs, as in Figure 1. The trend of accelerated supercomputers is indicated in the latest June 2013 ranking of the TOP 500 and the Green 500. In the TOP 500, 51 are powered by GPUs [2]. Although accelerated systems make up only 10% of the systems, they accomplish 33% of the total computations. In the Green 500, the most efficient systems powered by K20 surpassed 3 GFLOPS per watt, up from 2 GFLOPS per watt in the June 2012 ranking [3].

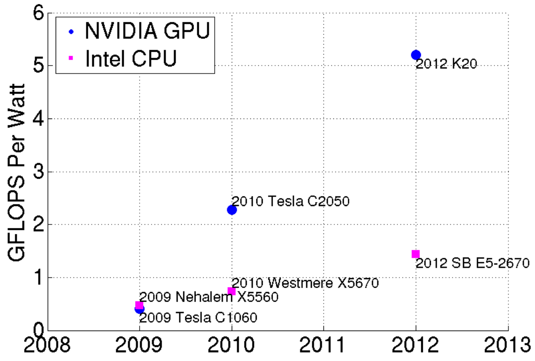

Figure 1: Performance per watt of NVIDIA GPUs versus Intel CPUs in double precision.

Energy consumption can be expressed as

$$Energy = Power \cdot Time$$

Reducing either factor on the right side of the equation will help to reduce energy consumption. Reducing processor power can involve either hardware redesign or software redesign. Hardware redesign includes changing hardware threading, power gating or decreasing frequency [4, 5]. A more power efficient implementation of the same code is a software aspect. Figure 1 shows the performance per watt of NVIDIA GPUs and that of Intel CPUs in double precision floating-point operations, where we use the theoretical peak performance as the FLOPS and TDP (Thermal Design Power) as watts.

Most users might be tempted to use TDP (Thermal Design Power) as the power of their application. However, TDP is only an engineering term and can be vastly different from the actual power used by applications [7]. A challenge for most users is that it is impractical to attach a power meter to monitor the power usage of their application. Fortunately, for Intel Sandy Bridge and NVIDIA Kepler users, monitoring and management of power is assisted by using RAPL and NVML, respectively [8, 9]. With these tools, computing can be more power aware for users.

In this paper, we redesign a hydrodynamic code on CPU-GPU clusters. BLAST is a software package that solves the

1530-2075 2014

DOI 10.1109/IPDPS.2014.103

equations of compressible hydrodynamics using high order finite element (FE) methods [10, 1]. High order numerical methods (p-refinement) and/or high resolution meshes (h-refinement) are introduced, to reveal more refined physical features as shown in Figure 2. However, for a given number of degrees of freedom, high order methods are more computationally intensive than low order methods, as they couple more degrees of freedom on the mesh. Therefore, the higher the order of the method, the more FLOPs per memory access. The most floating point intensive part of BLAST is the "corner force" part which can take up to 55%-80% of the total run time, increasing with the order of the methods, but accounts for only 10% of the code. In our CPU-GPU solution, the FLOP intensive parts, including corner force computation are accelerated on the GPU, while other parts are still on the CPU. In order to exploit the hardware parallelism of GPUs and achieve high performance, we redesigned the CPU code with CUDA. Most of the target CPU code are processed into GPU-efficient parallel batched matrix operations, which are expressed by linear algebra routines(kernels).

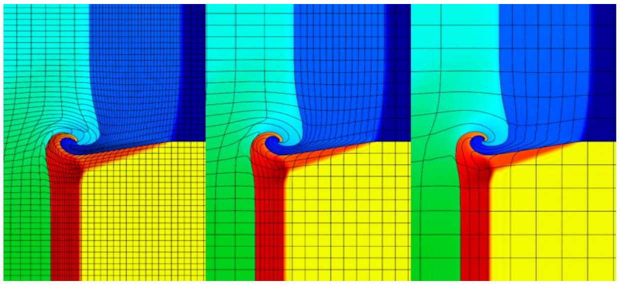

Figure 2: Shock triple-point benchmark using Q_8-Q_7, Q_4-Q_3 and Q_2-Q_1 finite elements from left to right.

Our contributions can be summarized as follows.

1) We reestablish the appeal of high order methods on GPUs. Compared to low order methods, the speedup and energy efficiency (greenup) of high order finite elements are greater.

2) We design custom linear algebra kernels on the GPU and intensively optimize them. Our optimization results in less time and less power. The optimized kernels achieve a substantial improvement in performance compared to the widely used vendor's library routines.

3) We analyze the power and energy consumption of a real application on CPUs and GPUs. Compared to the CPU-only solution, we show that our hybrid solution obtains both speedup and greenup.

4) We use CUDA and OpenMP inside MPI in order to exploit all of the computing resources of multi-core CPU with Fermi GPUs. To our best knowledge, this is the first time to apply all three programming methods in hydrodynamics.

5) We present a good weak scaling on the current NO.2 supercomputer ORNL Titan to 4096 computing nodes.

2. THE BLAST ALGORITHM

The BLAST C++ code uses high order finite elements 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:

$$\text{Momentum Conservation:} \quad \mathbf{M}_{\mathcal{V}} \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = -\mathbf{F} \cdot \mathbf{1}, \tag{1}$$

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

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

where $\mathbf{v}$, $\mathbf{e}$, and $\mathbf{x}$ are the unknown velocity, specific internal energy, and grid position, respectively. $\mathbf{1}$ is a vector with each element 1. The kinematic mass matrix $\mathbf{M}_{\mathcal{V}}$ 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 sparse 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.

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 $\{\alpha_k\}$:

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

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

where, J_z is the Jacobian matrix, and the hat symbol indicates the quantity is on the reference zone. Other quantities will be explained shortly. In the CPU code, $\mathbf{F}$ 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

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

and

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

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. Finite element zones are defined by a parametric mapping Φ_z from a reference zone. The Jacobian matrix $J_z = \hat{\nabla}\Phi_z$ is non-singular and varies inside each zone. The determinant $|J_z|$ can be viewed as local volume. The total stress tensor $\hat{\sigma}(\hat{\vec{q}}_k)$ requires evaluation at each time step and involves 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 finite element solution is specified by 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 3) 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, we can arbitrarily increase the floating point intensity of the corner force kernel of (2) as well as the overall algorithm of (1) - (3).

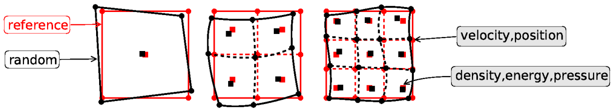

Figure 3: Schematic depiction of bilinear (Q_1-Q_0), biquadratic (Q_2-Q_1), and bicubic (Q_3-Q_2) zones.

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;
3) Compute initial time step;
4) Loop over zones in the sub-domain of each MPI task:
(4.1)Loop over quadrature points in each zone;
(4.2)Compute corner force associated with each quadrature point and update time step;
5) find minimum time step and assemble zone contribution to global linear system;
6) Solve global linear system for new accelerations;
7) Update velocities, positions and internal energy;
8) Go to 4 if final time is not yet reached, otherwise exit.

Step 4 is associated with the corner force calculation of (2) which is a computational hot spot where we focus our effort. Step 6 solves the linear equation (using a simple PCG solver) of (1). Table 1 shows timing data for various high order methods in 2D and 3D. The computational cost of both the corner force and CG solver increase as the order of the method k and dimension are increased, though the cost of the corner force calculation grows faster than that of the CG solver.

3. HYBRID PROGRAMMING MODEL

Multi-GPU communication across nodes relies on CPU-GPU communication on a single node and CPU-CPU communication across nodes. Our implementation is composed of the following two layers of parallelism: (1) MPI-based parallel domain-partitioning and communication between CPU; (2) CUDA based parallel corner force calculation inside each MPI task.

Table 1: Profile of BLAST on Xeon CPU: The corner force kernel consumes $55\% - 75\%$ of total time. The CG solver takes $20\% - 34\%$.

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

3.1 CUDA Implementation

We implemented the momentum (1) and energy (2) equations on the GPU. In the CUDA programming guide [11], the term host is used to refer to CPU and device to GPU. Hereafter in this paper, we follow this practice. The CUDA implementation is composed of the following set of kernels:

3.1.1 CUDA Code Redesign

The CPU code loops over the points in each zone and performs operations on the variables, most of which are represented as matrices. The right of Figure 6 shows our base CUDA implementation. `kernel_loop_quadrature_point` is a kernel to unroll $\mathbf{A}_z$ which loops over quadrature points. The kernel on Fermi is faster than a six core Westmere X5660 CPU. Yet, it is still inefficient and dominated most of the GPU time. We replaced it with six new designed kernels **1-6**. The formulation of these CUDA kernels are based on two considerations. First, the kernels can be reused. Second, they can be translated into linear algebra routines whose interfaces are very similar to LAPACK's. Except kernel 1-2, the other kernels are all based on a LAPACK interface and are of general purpose. Thus, it is easy for developers to maintain the code and for others to reuse them. A major change from the CPU code to our newly designed CUDA code is that loops become batch-processed. Thus the challenge is to write GPU-efficient massively parallel batched matrix operations.

The purpose of kernel 1-6 is to compute $\mathbf{A}_z$ in (5).

Kernel 1,2 are used in evaluations of $\hat{\sigma}(\hat{\vec{q}}_k)$, and in computing the adjugate of $\mathbf{J}_z$. Independent operations are performed on each quadrature point (thread). Each thread implements routines for computing SVDs and eigenvalues for $DIM \times DIM$ matrices.

Kernel 3,4 evaluate $\hat{\nabla}\hat{\vec{v}}(\hat{\vec{q}}_k)$, $\mathbf{J}_z(\hat{\vec{q}}_k)$.

Kernel 5,6 Auxiliary kernels batched DGEMM, where all matrices are $DIM \times DIM$. These kernels multiply Jacobian matrices $\mathbf{J}_z$, gradient of basis functions $\hat{\nabla}\hat{\vec{w}}_i$ and stress tensor values $\hat{\sigma}$ together.

Kernel 7 One thread block works on one zone. Each thread block does a matrix-matrix transpose multiplication $\mathbf{F}_z = \mathbf{A}_z\mathbf{B}^\mathbf{T}$, where $\mathbf{A}_z$ is the output of the last kernel. Therefore, this kernel can be also expressed as a batched DGEMM, with the number of batches being the number of zones.

Kernel 8 and **Kernel 10** compute $-\mathbf{F} \cdot \mathbf{1}$ from (1) and $\mathbf{F}^T \cdot \mathbf{v}$ from (2), respectively. Each thread block does a matrix-vector multiplication (DGEMV) and computes part of a big vector. All thread blocks assemble the result vector. The two kernels can be expressed as batched DGEMV.

Kernel 9 is a custom conjugate gradient solver for (1) with a diagonal preconditioner (PCG) [16]. It is constructed with CUBLAS/CUSPARSE routines [14].

Kernel 11 is a sparse (CSR) matrix multiplication by calling a CUSPARSE SpMV routine [14]. The reason for calling SpMV routine instead of using a CUDA-PCG solver as in kernel 9 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.

A summary of the kernels is given in Table 2.

Table 2: Implementation on Kepler. Kernel 9 is a set of kernels instead of one single kernel.

No.	Kernel Name	Purpose
1	kernel_CalcAjugate_det	SVD,Eigval,Adjugate
2	kernel_loop_grad_v	EoS, $\hat{\sigma}(\vec{\hat{q}}_k)$
3	kernel_PzVz_Phi_F_Batched	$\hat{\nabla}\vec{\hat{v}}(\vec{\hat{q}}_k)$, $\mathbf{J}_z(\vec{\hat{q}}_k)$
4	kernel_Phi_sigma_hat_z	$\hat{\sigma}(\vec{\hat{q}}_k)$
5	kernel_NN_dgemmBatched	Auxiliary
6	kernel_NT_dgemmBatched	Auxiliary
7	kernel_loop_zones	$\mathbf{A}_z\mathbf{B}^\mathbf{T}$
8	kernel_loop_zones_dv_dt	$-\mathbf{F}\cdot\mathbf{1}$
10	kernel_dgemvt	$\mathbf{F}^T\cdot\mathbf{v}$
9	CUDA_PCG	Solve linear system(1)
11	SpMV	Solve linear system(2)

3.1.2 Memory Transfer and CPU Work

Input vectors $(\mathbf{v}, \mathbf{e}, \mathbf{x})$ are transferred from the host to the device before kernel 1, and output vectors $\frac{\mathbf{de}}{dt}$ are transferred back from the device to the host after kernel 11. Whether the vector $\frac{\mathbf{dv}}{dt}$ after kernel 9 or the vector $-\mathbf{F}\cdot\mathbf{1}$ after kernel 8 is transferred to the host depends on turning on/off the CUDA-PCG solver. The time integration of the output right-hand-side vectors in the momentum (1) and energy (2) equations, together with the motion (3) equation are still done on CPU to get new $(\mathbf{v}, \mathbf{e}, \mathbf{x})$ states.

Because of kernel 8,10, the two DGEMV kernels, we avoid transferring the full matrix $\mathbf{F}$ which has large number of nonzeros due to its high-order nature. This leads to significant reduction in the amount of data transferred between the CPU and GPU via the relatively slow PCI-E bus.

3.2 Optimization and Autotuning

Data streamed **kernel 1,2**. Each thread maintains a $DIM \times DIM$ workspace for each matrix and a number of scalar variables. In the base implementation, the workspace related to the two kernels will be put in local memory by compiler, even declared as registers. The register spill issue is serious by inspecting the PTX code, especially on Fermi(computing ability 2.0) whose registers are rather limited [11]. After separating the two kernels out, there is no register spill issue related to their workspaces, especially on Kepler(computing capability 3.5) which doubles the number of physical registers per SMX. Registers are critical to the performance of these two kernels, because they involve a large amount of scalar operations related with the workspace. Figure 4 compares the performance of using register array and that of being forced using local memory for workspace on Kepler for a 3D Q_2-Q_1 case. By taking advantage of the more registers available on Kepler, kernel 2 achieved a 4x speedup.

Kernel 5,6: Batched DGEMM of $DIM \times DIM$ matrices. Each thread block performed multiple matrix operations. This avoided an unaligned memory access problem in the case of one thread block reading one matrix size of 4 or 9. Threads inside block are configured flexibly. When reading and writing, threads can be organized as one dimension to access linearly stored data in global memory. While performing multiplication, they are configured as two-dimension to naturally fit the matrix indexing. The number of matrix performed per thread block can be tuned to find an optimal occupancy. We use autotuning(Section 3.2.1) to tune this number. We find 32 delivered the best performance with an occupancy 98.3%. Figure 5 shows the effect of tuning and we are able to achieve 60% of the theoretical peak performance of batched DGEMM on K20. Notice the peak performance of batched is lower than that of the regular DGEMM, since batches of small matrices can not achieve the same GFLOPS as one large matrix. One n^2 matrix performs n^3 operations, but k^2 small $(\frac{n}{k})^2$ matrices only perform $k^2(\frac{n}{k})^3 = \frac{n^3}{k}$ operations with the same input size [13]. The Batched DGEMM flop per element is $\frac{2k * DIM^3}{3k * DIM^2}$ $= \frac{2DIM}{3}$. The bandwidth of K20 is 208GB/s, which means it is able to get 26G data in double precision per second. Since each element will perform 4/3, 2 operations, the theoretical peak performance on K20 is 35, 52 Gflop/s for DIM = 2,3, respectively. `cublasDgemmbatched` has exactly the same purpose but only achieves 1.3Gflop/s.

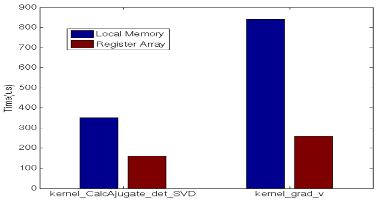

Figure 4: Performance of kernel 1,2 with local memory and register arrays, respectively.

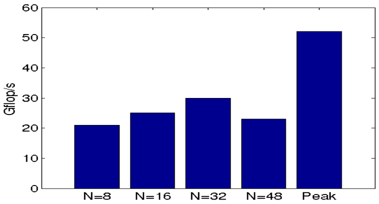

Figure 5: Tuning of kernel 3 which achieved 60% of theoretical peak performance on K20. N is the number of matrices performed in each thread block.

Kernel 3,4 implement custom batched DGEMM $\mathbf{C} = \mathbf{AB}$. Notice here $\mathbf{A}$ and $\mathbf{B}$ are different from that in (5) and (6). They are described in Table 3. In kernel 3,4, since number of quadrature points << zones, the number of matrices $\mathbf{B}$ is much smaller compared to that of $\mathbf{A}$. Therefore we choose to use texture memory to access matrix $\mathbf{B}$ in version 1(v1), because we hope they fit the cache. We always use shared memory to read $\mathbf{A}$. It seems that reading $\mathbf{B}$ via cached texture memory is still not as fast as shared memory as in v2, though shared memory will introduce synchronization overhead. To reuse **A**, each thread block will loop over all of the smaller matrices **B**, because if this thread block only partially loops over **B**, **A** will be picked up in another thread block which loops over the remaining part of **B**. Similar to the optimizations in **kernel 5,6**, to increase occupancy we fit multiple **A** in one thread block. This, in turn, also helps the reuse of **B**, for the same reason as above. Yet, too many matrices will overfill shared memory and reduce the occupancy which offsets the benefits of data reuse. However; we can use auto tuning (see Section 3.2.1) to find the balance point; v3 is the tuned result, as shown in Figure 7.

Table 3: Points refers to the quadrature points. For example, in kernel 3 each quadrature point corresponds to a matrix B and each zone corresponds to a matrix A.

Name	Num of A	Num of B	Num of C
kernel 3	zones	points	zones*points
kernel 4	zones*points	points	zones*points
kernel 7	zones	1	zones

Optimization of **kernel 7**. v1 is a naive implementation. $\mathbf{A}_z$ and **B** were loaded directly from global memory, as shown in Figure 7. In v2, we use shared memory to read $\mathbf{A}_z$, while **B** is read in constant memory, since **B** is globally shared by all thread blocks. v2 is a substantial improvement, but still not satisfactory. As a further optimization, v3 uses blocking technique. 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 GPU, blocking can deliver a second benefit: reducing the amount of shared memory for each thread block and allowing 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, while blocking in rows has little benefits, probably because the data layout is in column major. Again, the optimal blocking size can be found by our autotuning technique. An alternative implementation of these batched DGEMM kernels is to call `cublasDgemmbatched` [13] as shown in Figure 7.

Bandwidth. Because of the "memory wall", most applications are bandwidth bounded nowadays. Here we chose to profile the bandwidth of the base and optimized code on K20. Figure 8 shows the bandwidth of all the 3 level memory, from on-chip memory L1/Shared to off-chip L2 and device memory. All the optimized kernels exceeded the base implementation in bandwidth of L1/Shared and device memory except kernel 3 in device memory, which instead has very high bandwidth in L1/shared memory. Optimized kernels achieved much higher bandwidth in L1/shared memory, because they exploited shared memory and/or registers. Kernels 1,2 have streaming data which is more likely cached by L2. Their bandwidth in L2 tend to be higher compared to other kernels. Because on-chip memory is much faster than off-chip memory, the bandwidth of on-chip memory has a greater impact on performance.

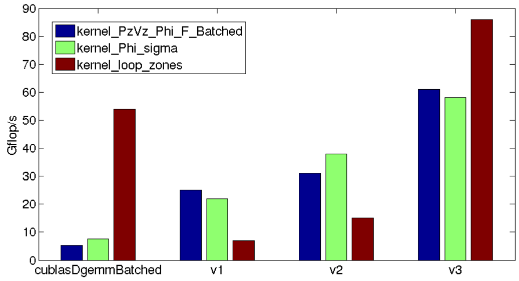

Figure 7: The performance of kernel 3,4,7 on K20. v1 is the straightforward implementation. v3 is the optimized and tuned result.

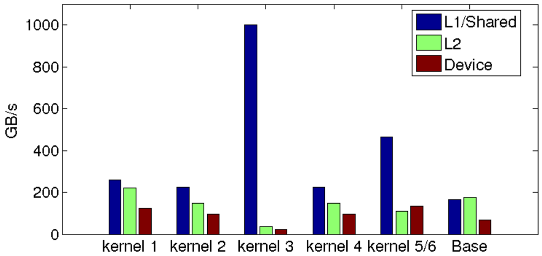

Figure 8: Memory bandwidth of previous (base) and optimized kernels. The theoretical peak bandwidth of device memory of K20 is 208GB/s.

Kernel 8 and 10 can be viewed as batched DGEMV. In our implementation, each thread block (zone) does a DGEMV operation. An implementation involving CUBLAS is to put `cublasDgemv` in streams of number zones, as recommended in the User Guide [13], since there is no batched DGEMV routine in CUBLAS. However, the performance is very poor, as shown in Table 4.In this test, each small matrix is 81 by 8 and each vector is 8. Number of batches (streams) is 4096. Our custom kernel is 90x faster than that of `cublasDgemv`, achieving 50 % of theoretical peak performance of batched DGEMV on C2050.

Table 4: Custom kernel 8 and streamed `cublasDgemv` implement batched DGEMV on one C2050.

	streamed `cublasDgemv`	kernel 8	theoretical
Gflop/s	0.2	18	35.5

We implemented a custom CUDA-PCG solver(**kernel 9**) from scratch. CUDA-PCG contains a SpMV and a dot product routine only where we call CUSPARSE SpMV and cublasDdot as shown in Figure 6. Kernel 11 is a sparse matrix multiplication routine in CUSPARSE. Notice, this SpMV routine is also needed in **kernel 9**. But kernel 11 is only called once per time step. From Figure 6, we can see the performance of SpMV is critical to the CUDA-PCG, since it is the biggest component of CUDA-PCG. The CUDA-PCG solver is outside of corner force. By comparing Figure 6 and timing data in Table 1, we can see the optimized corner force time becomes very small in overall time.

The impact of redesign and optimization is shown in the right side of Figure 6. `kernel_loop_quadrature_point` is replaced by kernel 1-6. Its percentage decreases to 25% from 65% after optimization. The actual time of kernel `csrMv_ci_kernel` (SpMV) does not change, but its percentage increases from 30% to 65%, because the total time is

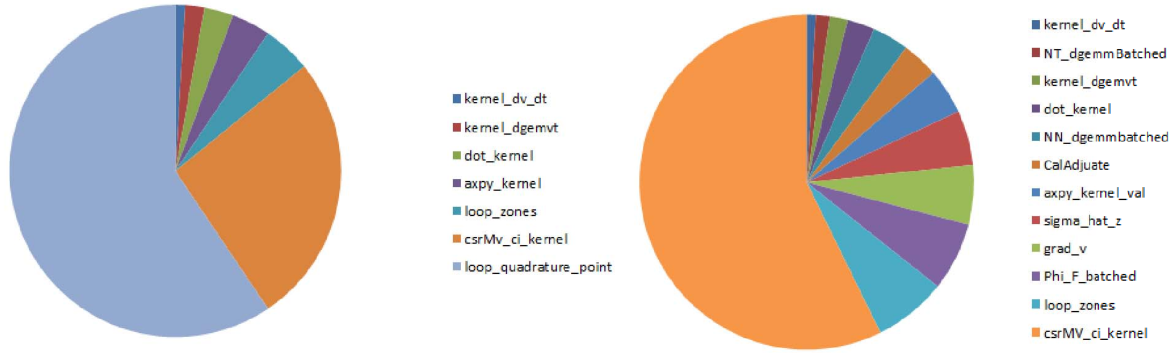

Figure 6: A break down of kernel time of the base implementation (*left*) versus the redesigned and optimized implementation (*right*). The CsrMv_ci_kernel time remains the same in the two implementations. It dominates in the redesign (*right*) due to the increased performance of the other kernels

reduced. Its percentage is big, because it is called many times in one time step. Other kernels will only be called once, except kernel 5 twice.

3.2.1 Autotuning CUDA kernels

The dimensions of the Jacobian matrix J and the total stress tensor $\hat{\sigma}$ depend only on the spatial dimension DIM, while the size of other matrices is a function of the finite element order k. For example, $\hat{\vec{w}}_i(\hat{\vec{q}}_k)$ is 81×64 for Q_2-Q_1 finite elements and 375×512 for Q_4-Q_3 finite elements in 3D. We propose an autotuning technique to adapt our kernels to the order of the method. Tuning can cause a large difference in performance, as shown in the figures above.

Our autotuing is based on the iterative time stepping nature of CFD applications. First, we parametrize every kernel as far as possible. Parameters include the blocking size in kernel 7, number of matrices in kernel 5,6, etc. Second, we set up a range of values for the parameters we want to tune. Artificial values, like those exceeding the shared memory, will be eliminated. Depending on how far we want to tune, we may tighten or loosen the range, because it is easy to end up with ten thousand possibilities with a few parameters. This second step usually takes most efforts. In each sampling period, the scheduler picks up a candidate value and times it. After comparing all the candidates, the scheduler will give an optimal one. In our test, one sampling period consists of forty time steps which will be averaged to eliminate the noise.

3.3 CUDA + OpenMP Implementation of Corner Force

CUDA and OpenMP is used on Kepler K10 and Fermi clusters, where only one MPI process is allowed to connect to the GPU at a time. Multiple MPI processes will be forced into a serialization if the GPU is in shared compute mode. They will be prohibited if it is in exclusive compute mode. This limitation usually requires the number of GPU to match CPU cores, otherwise GPU and CPU utilization will be limited. However, the number of CPU cores is usually far more than that of GPU. Therefore, we adopt OpenMP to deal with multi-core and CUDA to deal with one K10 or Fermi in one MPI process. On Kepler K20, we may turn OpenMP off, because MPI processes from multicore can use a subset of a shared K20 simultaneously due to Hyper-Q (see Section 4.2) without causing the CPU core idle.

In the corner force computation, after the launch of CUDA kernels, control can return to a host thread prior to the GPU completing work. The host thread will spawn OpenMP threads and distribute a portion of the 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. 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. We use auto-balance to find the ratio between CPU and GPU to ensure load balance. The idea of auto-balance of CUDA and OpenMP is the same with autotuning. The scheduler will compare their time to decide to move more or less work to each processor. After a few sampling periods, the scheduler will converge to an optimal ratio. Our tests shows that the convergence only takes a few time periods as shown in Table 5.

Table 5: Ratio refers to the percentage of zones on GPU. Zones are allocated on a six core X5560 CPU and a C2050 GPU.

Problem	Optimal ratio	Convergence periods
2D: Sedov	75%	14
2D: Triple-pt	77%	12

Auto tuning is a convenient and robust tool. When the code is ported on another architecture, the changes will be detected and the load will be rebalanced automatically.

3.4 MPI Level Parallelism

The MPI level parallelism in BLAST is based on MFEM which is a modular C++ finite element library [15]. At the initialization stage (Step 2 in Section 2), MFEM takes care of the domain splitting and parallel mesh refinement as shown in Figure 9. Each MPI task is assigned a sub-domain 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 10. This results in a non-overlapping decomposition of the global vectors and matrices and typical FE 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.

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 $\mathbf{M}_{\mathcal{V}}$ 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-GPU, is beyond the scope of the present work. With the higher order of the methods, CG time will be less significant compared to corner force time. Therefore, we only consider the CUDA-PCG solver for (1) on a single GPU.

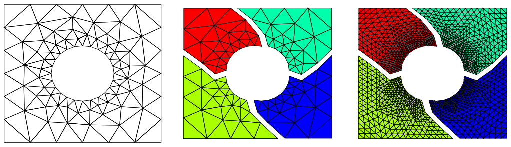

Figure 9: Parallel mesh splitting and refinement

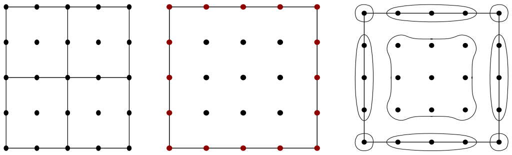

Figure 10: Zones assigned to one MPI task and associated Q_2 DOFs (left); the DOFs at the boundary of this subdomain are shared with neighboring tasks (middle); groups of DOFs, including the local group of internal DOFs (right).

4. TESTING RESULTS AND DISCUSSION

For our test cases we consider the 3D Sedov blast wave problem (see [1] for further details on the nature of these benchmarks). In all cases we use double precision. The gcc compiler and NVCC compiler under CUDA v5.0 are used for the CPU and GPU codes, respectively.

4.1 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 6.

4.2 Performance on A Single Node

Due to the new feature Hyper-Q of Kepler, multiple MPI processes can run on a K20 GPU simultaneously. The K20 GPU is able to set up to 32 work queues between the host and the device. Each MPI process will be assigned to a different hardware work queue, enabling them to run concurrently on the GPU.

In our test, the CPU is a 8 core Sandy Bridge E5-2670 and the GPU is a K20. Unless explicitly noted, we always use them to perform our tests in the following sections. In our configuration, 8 MPI tasks share one K20. Only corner force is accelerated on the GPU. Figure 11 shows the speedup achieved by CPU-GPU over the CPU. We compared two order of methods Q_2-Q_1 and Q_4-Q_3. When the order is higher, the percentage of corner force increases, and GPU acceleration benefits BLAST more. The overall simulation time of Q_4-Q_3 compared to Q_2-Q_1 is 3.2x on the CPU, but only 2x on CPU-GPU.

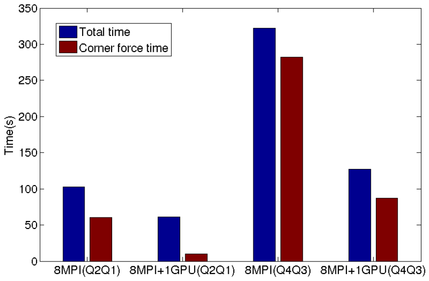

Figure 11: Speedup of CPU-GPU code over the CPU. A 1.9x overall speedup is obtained using Q_2-Q_1 elements; 2.5x using Q_4-Q_3 elements.

4.3 Performance on Distributed Systems: Strong and Weak Scalability

We tested our code on ORNL Titan which has 16 AMD cores and 1 K20m per node. We scaled it up to 4096 computing nodes. 8 nodes is the base line. For a 3D problem, one refinement level will make the domain size 8x bigger. We fixed a domain size of 512 for each computing node, and used 8x more nodes for every refinement. From 8 nodes to 4096 nodes, the time for 5 cycles (steps) increase from 0.85 to 1.83 seconds, see Figure 12. The limiting factor is the MPI global reduction to find the minimum time step after corner force computation and MPI communication in MFEM (Step 5 in Section 2).

We also tested the strong scalability on a small cluster, the SNL Shannon machine. It has 30 computing nodes, with two K20m and two sockets of Intel E5-2670 CPU per node. Figure 13 shows the linear strong scaling on this machine. The domain size is 32^3.

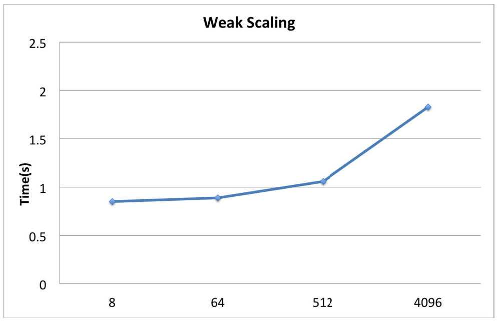

Figure 12: Weak scaling on ORNL's Titan. Time is for 5 time step cycles. The x-axis is the number of nodes.

5. POWER AND ENERGY ANALYSIS

Table 6: Results of CPU and GPU code 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.

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

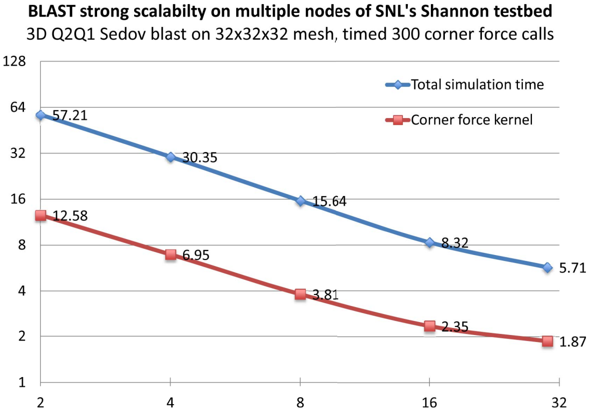

Figure 13: Strong scaling on SNL's Shannon. The x-axis is the number of nodes. The y-axis is run time on a log scale.

Generally, there are two ways to measure power. First is attaching an external power meter to the machine; this is from a hardware aspect. It is accurate, but it can only measure the power of the whole machine. It is not able to profile power usage of individual processors or memory. The other way is estimation from a software aspect. We adopt this way in our measurements.

5.1 Profiling CPU Power with Intel RAPL

From SandyBridge, Intel CPUs support on board power measurement via the Running Average Power Limit (RAPL) interface [8]. The internal circuitry can estimate current energy usage based on a model accessing Model (or sometimes called Machine) Specific Registers(MSRs), with an update frequency on the order of milliseconds. The power model has been validated by Intel [17] to actual energy.

RAPL provides measurement of:

- The total package domain.
- PP0 (Power Plane 0) which refers to the processor cores in a package.
- Memory which includes the directly-attached DRAM.

Figure 14 shows the power of two CPU packages and their DRAM. For comparison purposes, we let one package(processor) busy, while the others we keep idle. The full loaded package power is 95W with DRAM at 15W. The idle power is slightly lower than 20W with DRAM almost at 0. The TDP of the E5-2670 is 115W. Our observation 95W (82%) confirms the AMD reports of the normal range of Average CPU Power (ACP) in [7]. The test is a 3D Q_2-Q_1 case with 8 MPI tasks without GPU.

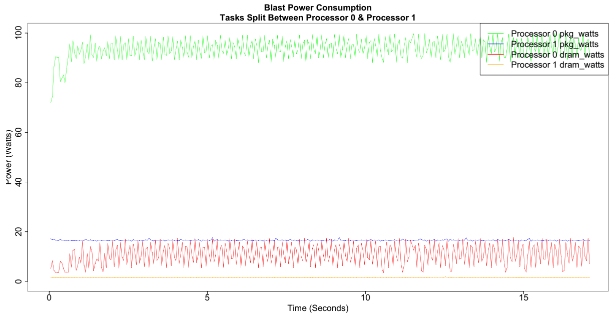

Figure 14: Power of two packages of Sandy Bridge CPU. Package 0 is full loaded. Package 1 is idle.

5.2 Profiling GPU Power with NVML

Recently, NVIDIA GPUs support power management via the NVIDIA Management Library (NVML). NVML provides an API to developers. NVIDIA also provides a high level utility nvidia-smi which calls the same interface. It only reports the entire board power, including GPU and its memory. It has milliwatt resolution within +/- 5 W and is updated per millisecond. Our CUDA kernels time is around several to tens milliseconds for our target problem, so the computation will not be missed by NVML.

Kernels aggregated in the corner force calculation are set as a testing component, and kernels in CUDA_PCG solver is another component. Therefore, kernel aggregated power usage is tested instead of individual kernels, since significant coding is needed to separate a single kernel on GPU while keeping others on CPU. We test the power usage of one K20 GPU in six scenarios. 1,2) Base versus optimized implementation with both corner force and CUDA_PCG solver enabled with 1 MPI task We call it overall in Figure 15. 3) Optimized corner force (Q_2-Q_1) with 1MPI task. 4,5) Optimized corner force (Q_2-Q_1 and Q_4-Q_3) with 8 MPI tasks running on the same GPU. 6) CUDA_PCG (Q_2-Q_1) only with 1 MPI task. The test case is a 3D Sedov problem with the domain size 16^3, which is the maximum size we were able to allocate with Q_4-Q_3 elements because of memory limitation for K20. The GPU is warmed up by a few runs to reduce noise. Our test shows that the startup power is around 50W by launching any kernel. The idle power is 20W if the GPU is doing nothing for a long time. The TDP of K20 is 225W.

From the base versus the optimized in Figure 15 we can see, the optimized code not only runs faster, but also lowers the power cost relative to the base implementation. Since both perform the same FLOPs, their main difference is in how they exploit the memory. The memory power consumes around 25% of total GPU power in various reports [5]. Studies have examined the power consumption of different components of GPU [18, 19]. The power consumption of device memory is much higher than on-chip memory. In the microbenchmarks of [19], the device memory power is 52, while shared memory is 1 with FP and ALU only 0.2(normalized unit). Their difference can be explained by their accessing cost. Accessing on-chip shared memory can only take 1 cycle, while accessing device memory may take 300 cycles [11]. It requires much more energy to drive data across to DRAM and back than to fetch it from on-chip RAM. Because the optimized kernels effectively exploited on-chip memory and significantly improved the bandwidth, as shown in Figure 8, the memory utilization efficiency is improved and power is reduced.

From the corner force 1MPI and 8MPI cases, we can see when the GPU is shared by 8 MPI tasks, its power usage will be higher than 1 MPI (with the same domain size and problem). We did not find any previous report about this situation, but obviously this additional power cost should come from the overhead of Hyper-Q. 1MPI corner force using Q_2-Q_1 elements has not saturated the GPU, therefore its power is low. Compared to using Q_2-Q_1 elements, because the FLOPs and data of using Q_4-Q_3 elements is much greater, its utilization and power consumption of GPU is also higher.

The power usage of the solver (CUDA-PCG) using 1 MPI task is higher than that of the corner force calculation using 1 MPI task as shown in Figure 15. This is because CG(SpMV) is effectively memory bound due to its sparse structure;the it is very hard to achieve memory bandwidth efficiency comparable with dense operations such as those used in the corner force calculation.

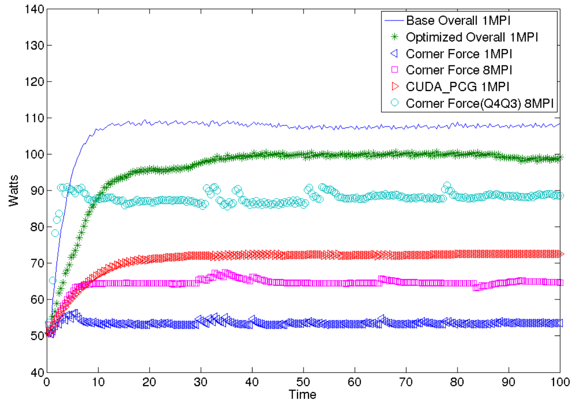

Figure 15: Except denoted, all are with a Q_2-Q_1 method. The stable value of y-axis is more meaningful. The x-axis value is trivial, since the initialization stage on CPU(Section 2) can be long.

The CPU power with corner force accelerated on GPU is shown in Figure 16. Both of the two processors are busy. The total package power is around 75W and PPO at 60W. Their difference is mainly the DRAM power. Compared to Figure 14, CPU power is reduced by 20W. We tested various orders of methods, but did not see any obvious difference.

5.3 Greenup

Similar to the notion of speedup which is usually used to describe the performance boost, we define a notion of greenup to quantify the energy efficiency [28].

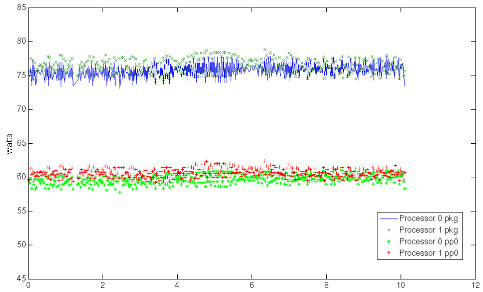

Figure 16: CPU power with GPU accelerating.

$$\begin{aligned}
Greenup &= \frac{CPU_{energy}}{(CPU+GPU)_{energy}} = \\
&= \frac{CPU_{power} \cdot CPU_{time}}{(CPU+GPU)_{power} \cdot (CPU+GPU)_{time}} \\
&= \frac{CPU_{power}}{(CPU+GPU)_{power}} \cdot Speedup \\
&= Powerup \cdot Speedup
\end{aligned}$$

where powerup and speedup are larger than 0. Powerup may be less than 1, since CPU+GPU power may exceed that of CPU only. Yet, the speedup is greater than 1. Therefore the greenup will be larger than 1. Table 7 outlines the greenup, powerup and speedup of BLAST code of Sedov problem. Speedup is from Figure 11. The CPU+GPU power we used in Table 7 is by adding data in Figure 15 and Figure 16 together. The hybrid CPU-GPU solution make BLAST greener. It saved 27% and 42% of energy, respectively for the two methods, compared to CPU only solution. However, The use of GPU is more than energy and speedup. Because the CPU power decreases, the power leakage and failure rate of cores are also reduced. Applications are more fault tolerant and runs faster, since the frequency of checking points can be reduced.

Table 7: The CPU-GPU greenup over CPU for BLAST code in 3D Sedov problems.

Method	Power Efficiency	Speedup	Greenup
Q_2-Q_1	0.67	1.9	1.27
Q_4-Q_3	0.57	2.5	1.42

6. RELATED WORK

Incompressible flow computations on GPUs with stencil computing were studied in [20]. Strategies of assembly of Galerkin methods on GPUs were discussed in [21]. [22] focused on solving the sparse linear system resulting from FE discretizations. A high order discontinuous Galerkin FE method applied in electromagnetic on GPUs was discussed in [23]. L.Wang et al examined the limiting factors of scaling on big CPU-GPU clusters in [24].

There are some reports comparing the power consumption of GEMM routines on CPU/GPU[25, 26]. They were using power meters and measuring the whole system power. [18] studied the power and energy consumption of BLAS2 and MAGMA Hessenberg kernel with NVML. PAPI provide an uniform interface to measure AMD and Intel CPU power, which in turn calls RAPL on Intel CPUs [27].

7. CONCLUSIONS

The BLAST code uses high order finite element methods to solve compressible hydrodynamics problems. It requires the assembling of corner force matrices and the solution of both sparse and dense linear algebra problems. In this paper, we redesigned the most computational intensive part of BLAST for CPU-GPU clusters. Our goal is to maximize the performance and lower the energy consumption of BLAST.

We redesigned a base CUDA implementation and modularized our new implementation into a set of linear algebra routines. Our optimized routines exceed CUBLAS library routines substantially in performance. Compared to the base implementation, our redesign and optimization achieved better performance per watt: resulting in 60% less time to sultion and a 10% reduction in power consumption. To adapt to high order methods, we further introduced an autotuning technique to tune our CUDA kernels. Due to their general purpose, these linear algebra routines can be utilized by other applications.

Our proposed hybrid solution has proven to be very beneficial in performance and energy efficiency, especially for high order finite element simulations; reestablishing the appeal of high order methods on GPUs. Compared to low order methods, the speedup and greenup of high order finite element methods are greater.

Acknowledgments

Implementations were done during an internship at LLNL. Some optimizations were done at University of Tennessee, Knoxville. The authors thank Barry Rountree for providing the CPU power graph. The authors would like to thank the NSF and NVIDIA for supporting. The authors thank the testing support of the Performance End Station PEAC Project sponsored by DOE under Contract No. DE-AC05-00OR22725. A portion of 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.

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] http://www.top500.org, 2013

[3] http://www.green500.org/

[4] P.Wang, C.Yang, Y.Chen, Y.Cheng *Power Gating Strategies on GPUs*, ACM Transactions on Architecture and Code Optimization, Volume 8 Issue 3, October 2011.

[5] J.Zhao, G.Sun, G.H.Loh, Y.Xie *Energy efficient GPU Design with Reconfigurable Inpackage Graphics Memory*, ISLPED,12

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

[7] ACP The Truth About Power Consumption Starts Here: http://www.amd.com/us/Documents/43761D-ACP-PowerConsumption.pdf

[8] Intel 64 and IA-32 Architectures Software Developer's. http://download.intel.com/products/processor/manual/

[9] https://developer.nvidia.com/nvidia-management-library-nvml

[10] BLAST: http://www.llnl.gov/casc/blast/

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

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

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

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

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

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

[17] Rotem, E., Naveh, A., Rajwan, D., Ananthakrishnan, A, E, Weissmann. *Power-management architecture of the Intel micro-architecture codenamed Sandy Bridge*, IEEE Micro, volume 32, no. 2, pp. 20âĂŞ27, 2012

[18] K.Kasichayanula, D.Terpstra, P.Luszczek, S.Tomov, S.Moore, G.D.Peterson. *Power Aware Computing on GPUs*, SAAHPC, 2012.

[19] S.Hong, H.Kim. *An Integrated GPU Power and Performance Model*, ISCA, 2010.

[20] 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.

[21] C.Cecka, A.Lew, E.Darve. *Assembly of finite element methods on graphics processors*, Numerical Methods in Engineering. Volume 85, Issue 5, 4 February 2011.

[22] J.Bolz, I.Farmer, E.Grinspun, P.Schroder. *Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid*. ACM Transactions on Graphics, 2003;

[23] Godel, N. Nunn, N. Warburton, T. Clemens, M. *Scalability of Higher-Order Discontinuous Galerkin FEM Computations for Solving Electromagnetic Wave Propagation Problems on GPU Clusters*. Magnetics, IEEE Transactions, Aug 2010.

[24] 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.

[25] S.Huang, S.Xiao, W.Feng. *On the Energy Efficiency of Graphics Processing Units for Scientific Computing*, IPDPS, 2009

[26] Y.Abe, H.Sasaki, M.Peres, K.Inoue, K.Murakami, S.Kato. *Power and Performance Analysis of GPU-Accelerated Systems*, USENIX, 2012 Worshop on Power-Aware Computing and Systems

[27] V.M. Weaver, M.Johnson, K.Kasichayanula, J.Ralph, P.Luszczek, D.Terpstra, S.Moore. *Measuring Energy and Power with PAPI*, Parallel Processing Workshops, 2012 41st International Conference on Sep, 2012

[28] D.Lukarski, T.Skoglund. *A priori power estimation of linear solvers on multi-core processors*, UPMARC Winter Meeting 2013