Flexible Hardware Mapping for Finite Element Simulations on Hybrid CPU / GPU Clusters

Aaron Becker
abecker3@illinois.edu

Isaac Dooley
idooley2@illinois.edu

Laxmikant V. Kalé
kale@illinois.edu

I. Introduction

The ever increasing peak floating-point performance and memory bandwidth of GPUs is making them increasingly ubiquitous in the high performance computing community. With increasing adoption of GPUs in cluster environments, applications that cannot take advantage of this hardware will be at a distinct disadvantage. For the class of applications that can achieve massive speedups of 100x or more on the GPU, the way forward is clear: maximum performance will depend on utilizing all available GPUs as efficiently as possible, with the CPU most likely relegated to managing the data flowing into and out of the GPU. However, for applications that can benefit from GPU execution, but may experience speedups that are only in the range of 5-10x, the appropriate relationship between the CPU and GPU is more difficult to determine, and may depend upon the specifics of the algorithm and hardware in question.

Finite element applications generally fit the description of codes that benefit from GPUs, but probably not by more than 15-20x even in the best case. In a cluster environment where data must be transferred to the CPU at regular intervals for synchronization, speedups less than 10x are typical. When cluster nodes have 8 CPU cores and more, it is clear that maximum performance will require taking full advantage of execution on both CPU and GPU.

We present an API and supporting software layer for finite element applications on unstructured meshes in hybrid CPU/GPU environments that allows runtime mapping of mesh partitions to either CPU or GPU hardware and effectively overlaps CPU and GPU work. This layer sits on top of the ParFUM [6] framework for unstructured meshes and takes advantage of its support for synchronization of shared nodes between mesh partitions. ParFUM in turn relies on the Charm++ parallel runtime system [4]. This software layer manages the creation and deletion of GPU memory buffers and the transfer of node and element data to and from the GPU at each synchronization point. It also provides a consistent API for accessing that data in both CPU and GPU functions, allowing very similar code for equivalent CPU and GPU kernels. We demonstrate the effectiveness of this scheme by presenting a functionally graded material simulation that scales to 128 nodes of the National Center for Supercomputing Applications (NCSA) Lincoln cluster, with a speedup of 2023 over a single CPU core.

II. API

We provide a software layer between ParFUM and the application which manages memory buffers on the GPU that correspond to ParFUM node and element fields. Mesh partitioning and distribution is handled by ParFUM, and at runtime a partition can determine which type of hardware it has been mapped to through a function call. We provide iterators that yield all nodes or elements on the local mesh partition and data access functions to extract associated data. A typical CPU kernel implementation loops over nodes or elements, applying an update rule to each. In the following example, a, F, and dt are variables representing acceleration, force, and time step size, respectively.

```
nodeIterator itr;
for (nodeItr_Begin(itr);
     nodeItr_IsValid(itr);
     nodeItr_Next(itr)) {
  n_data = node_GetData(itr);
  for (int i=0; i<dof; ++i) {
    const FP_TYPE a_old = n_data->a[i];
    n_data->a[i] = -n_data->F[i] / n_data->mass;
    n_data->v[i] += 0.5*dt*(n_data->a[i] + a_old);
  }
}
```

The type `FP_TYPE` can be resolved to either `float` or `double` at compile time to facilitate testing and validation at both levels of precision. Although current GPUs suffer a large performance penalty to peak FLOPs in double precision, we have observed that many unstructured mesh codes are strongly bandwidth limited (for example, both [5] and our own code described here) and may suffer far less from the use of double precision than the difference in peak rates suggests.

Kernel implementations targeted to the GPU are generally very similar to their CPU equivalents, with the exception that each GPU thread is only responsible for one node or element. The nodes or elements (depending on the kernel in question) are broken into thread blocks of uniform size, and we define `my_node` and `my_element` macros to identify the node or element a thread is responsible for based on its thread and block indices. A GPU-specific implementation of the function that acquires node or element data fetches a pointer to the relevant data from device global memory. The result is essentially the same as the inner loop of the CPU implementation. Depending on the memory access patterns of the kernel and the memory layout of mesh data structures,

padding to ensure coalesced reads or use of shared memory may also be points of departure from the CPU implementation. The following code fragment shows the GPU equivalent of the CPU kernel above.

```
n_data = node_GPU_GetData(my_node);
for (int i=0; i<dof; ++i) {
  const FP_TYPE a_old = n_data->a[i];
  n_data->a[i] = -n_data->F[i] / n_data->mass;
  n_data->v[i] += 0.5*dt*(n_data->a[i] + a_old);
}
```

III. Hardware Mapping

Effective mapping of work onto hardware resources is critical for high performance in a hybrid environment. We expect unstructured mesh application kernels to vary widely in terms of their relative speed on the CPU and the GPU. Published results show speedups from approximately 2x to 30x when run on the GPU [1], [2], [5], [8], depending on the amount of data per element or node, the order of the method used, and the phenomenon being modeled. Hardware configurations will vary widely in terms of the number of CPU cores available per node, the capability of those cores, and the number and quality of attached GPUs. In some clusters hardware capabilities will vary from node to node, including possibilities such as a the use of nodes from both NCSA's Abe and Lincoln clusters by a single application run.

In light of the variability in relative CPU/GPU performance and quantity, we aim to be as flexible as possible when mapping mesh partitions to hardware resources. Aside from initial mesh import and partitioning, which we confine to the CPU, and synchronization between partitions, which in a cluster environment must take place on the CPU, we allow any part of the application to execute on either CPU or GPU hardware on a per-partition basis.

Rather than attempting to determine the optimal partition size for each hardware resource that minimizes iteration time, we overdecompose the mesh and map multiple partitions onto each CPU or GPU. This greatly simplifies the task of initial partitioning, because we need only produce a large number of uniform size partitions, rather than attempting a multi-constraint partition that produces an appropriately sized partition for each hardware resource while still respecting locality. It also decouples the partitioning problem from the application and the particular hardware it will run on, which simplifies the task of tuning the application for new hardware and makes features like migration of mesh partitions to improve load balance possible [3].

Our approach does have some drawbacks relative to the strategy of creating a custom size partition for each type of hardware. Each mesh partition introduces some amount of memory overhead, and there is scheduling overhead associated with colocating multiple partitions on the same processor. In addition, the increased number of partitions means an increase in the total size of inter-partition boundaries and a corresponding increase in communication volume. However, if our partition respects locality most of the increase in boundary size will be among partitions located on the same hardware node. The cost of synchronization operations is determined by the amount of inter-node communication, greatly decreasing the performance impact of increased intra-node boundaries, if not eliminating it altogether.

When we map mesh partitions on GPU hardware, those partitions will still need to execute code on a CPU during synchronization phases. We therefore designate one CPU core per GPU to act as a manager for that GPU's partitions. All GPU kernel invocations and memory transfers are conducted asynchronously, and we associate a user-level thread with each partition. The thread yields after its operations have been enqueued, allowing the CPU to enqueue a large amount of GPU work and then poll for completion at synchronization points. To avoid wasting resources while waiting for the GPU partitions to complete, we can assign CPU partitions to the GPU manager cores that they execute after enqueuing all GPU work. With a good hardware mapping, the CPU partitions on the manager core will finish just as the GPU partitions are finishing, leading to efficient overlap of work on the CPU and the GPU.

IV. Application Study

We have used this framework to develop a simple unstructured mesh code for simulating functionally graded materials (FGM). FGM simulations handle materials whose properties vary spatially, and are used in areas from structural materials to optoelectrical devices [7], [9]. In the case of our application this leads to a unique stiffness matrix for each element. Our timestep loop consists of (1) updates to nodal displacements based on velocity and acceleration, followed by (2) calculation of the forces on each element computed from its displacement vector and the stiffness matrices of its surrounding elements, (3) synchronization to update forces on boundary nodes which have element neighbors on multiple partitions, then (4) update of nodal velocity and acceleration based on net forces, and finally (5) application of boundary conditions.

The most significant point of departure in our GPU implementation is in the force summation step. Because forces are computed by each element in its own GPU thread and these forces are accumulated onto nodes with many element neighbors, this step would suffer from data races if we simply summed the forces onto the nodes, as in the CPU version. Instead, on each node we reserve a separate memory location for each neighboring element to place its force contribution, and these forces are summed in a subsequent node kernel. This approach trades off some memory overhead in order to avoid synchronization. In a recent implementation of earthquake simulations on the GPU [5], the authors address the same issue using a graph coloring technique to execute only sets of elements that are known not to conflict simultaneously. This alternate approach avoids extra memory use at the cost of some locality and a smaller set of threads that can run concurrently.

We ran this application on the Lincoln cluster at the NCSA, a heterogeneous cluster consisting of 192 nodes with an Infiniband interconnect. Each node consists of two quad core

Intel Harpertown CPUs two NVIDIA Tesla GPUs (half of a Tesla S1070 unit).

To characterize the performance of our CPU and GPU kernels, we prepared modified kernels in which the section we wish to study is eliminated. We time the modified versions against the original kernels to identify their impact on performance. To prevent the compiler from optimizing away operations which we wish to time in our modified kernels, we insert a simple arithmetic expression that depends on the calculation of any otherwise unused values. This prevents the compiler from optimizing out the instructions we wish to time without meaningfully changing the performance characteristics of the kernel.

The force calculation kernel occupies 93% of execution time for CPU kernels and 61% of execution time for GPU kernels (for which 21% of the time is spent transferring data to and from the host CPU). This kernel consists of three steps: assembling nodal displacements into a vector, performing a matrix-vector update with the element's stiffness matrix, and writing back the resulting forces to the nodes.

We find that on CPU partitions, reading displacement data takes 24% of run time, matrix-vector update takes 39%, and force write-backs take 24%. For GPU partitions, reading displacement data takes 37% of the time, the matrix-vector update is only 9%, and force write-backs are 54%. These numbers somewhat under-count the impact of force write-backs, as they must still be accumulated in a subsequent kernel. We conclude that the GPU mesh partitions are bandwidth limited, which is corroborated by the fact that a double precision version of our application is only about two times slower than the single precision version, despite the fact that double precision arithmetic is approximately 12 times slower than single precision on these GPUs.

We determined the best mapping of mesh partitions to hardware resources via hand tuning. In principle an autotuning framework could be used to determine a good static mapping, and the Charm++ load balancer could be extended to support hybrid environments, but for this application a static mapping was sufficient. We achieved the best performance with two mesh partitions per normal CPU, one partition per GPU-managing CPU, and 17 partitions per GPU.

To analyze scalability, we perform weak scaling. On one node we use a mesh with 235K elements split into 48 partitions. On 128 nodes the simulation uses a 30.1M element mesh split into 6144 partitions. Figure 1 shows scaling for both single and double precision variants, with parallel efficiency of 97%.

V. Conclusions

Efficient use of all available CPU and GPU resources is critical in attaining the best possible performance in unstructured mesh codes in hybrid clusters. We provide supporting software that allows a flexible assignment of work to CPUs and GPUs on a per-partition basis. Any portion of the solution procedure can be executed on either CPU or GPU, and work can be effectively overlapped between synchronization points.

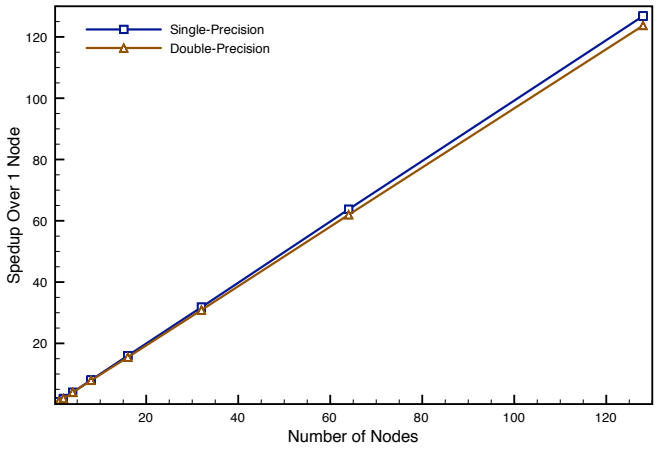

Fig. 1. Weak scaling of the application relative to 1 node. Each node contains 48 mesh pieces, with 4900 tetrahedral elements in each piece.

This software layer has proven effective in the development of a simulation of functionally graded materials on the Lincoln cluster.

References

[1] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. H. Buijssen, M. Grajewski, and S. Turek. Exploring weak scalability for fem calculations on a gpu-enhanced cluster. *Parallel Computing*, 33(10-11):685 – 699, 2007. High-Performance Computing Using Accelerators.

[2] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, and S. Turek. Using GPUs to improve multigrid solver performance on a cluster. *International Journal of Computational Science and Engineering (IJCSE)*, 4(1):36–55, 2008.

[3] L. V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.

[4] L. V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[5] D. Komatitsch, D. Michéa, and G. Erlebacher. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *Journal of Parallel and Distributed Computing*, In Press, Corrected Proof:–, 2009.

[6] O. Lawlor, S. Chakravorty, T. Wilmarth, N. Choudhury, I. Dooley, G. Zheng, and L. Kale. Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, 22(3-4):215–235.

[7] B. Shen, M. Hubler, G. H. Paulino, and L. J. Struble. Functionally graded fiber-reinforced cement composite: Processing, microstructure, and properties. *Cement and Concrete Composites*, 30(8):663–673, 2008.

[8] M. Woolsey, W. Hutchcraft, and R. Gordon. High-level programming of graphics hardware to increase performance of electromagnetics simulation. pages 5925–5928, June 2007.

[9] M. Wosko, B. Paszkiewicz, T. Piasecki, A. Szyszka, R. Paszkiewicz, and M. Tlaczala. Application and modeling of functionally graded materials for optoelectronic devices. pages 87–89, July 2005.