Memory and Parallelism Tuning Exploration using the LULESH Proxy Application

Ian Karlin, Jim McGraw, Jeff Keasler, Edgar Leon and Bert Still
Lawrence Livermore National Laboratory
Livermore, CA, 94551-0808
Email: karlin1,mcgraw1,keasler1,leon@lbnl.gov

Esthela Gallardo
University of Texas, El Paso
El Paso, TX 79902
Email: egallardo5@miners.utep.edu

Abstract—Current and planned computer systems present challenges for scientific programming. Memory capacity and bandwidth are limiting performance as floating point capability increases due to more cores per processor and wider vector units. Effective use of the new hardware requires finding greater parallelism while using relatively less memory. In this poster, we present how we tuned the Livermore Unstructured Lagrange Explicit Shock Hydrodynamics proxy application for on-node performance resulting in 62% fewer memory reads, a 19% smaller memory footprint, 77% more floating point operations vectorizing and less than 0.1% serial section runtime. Tests show decreased serial runtime of up to 57% and parallel runtime reductions of up to 75%. We are applying these optimizations to GPUs and a subset of ALE3D, from which the proxy application was derived. So far we achieve up to a 1.9x speedup on GPUs, and a 13% runtime reduction in the actual application for the same problem.

I. INTRODUCTION

Hydrodynamics is widely used in application codes to couple fluid flow to matter. It can consume up to one third the runtime of these applications. To provide a simpler, but still full-featured problem to test various tuning techniques and different programming models the Livermore Unstructured Lagrange Explicit Shock Hydro (LULESH) mini-app was created as one of five challenge problems in the DARPA UHPC program [1]. LULESH solves the sedov problem by modeling one octant of a symmetrical blast wave.

We are using LULESH to test optimization techniques and programming practices that increase the performance of code on current and future architectures. By using a mini-app we can quickly explore and evaluate techniques that hold promise before making the more extensive changes needed in production codes. We focus on increasing hardware parallelism utilization, reducing memory traffic and decreasing memory footprint. Optimizations target on-node memory bandwidth, memory footprint and parallelism, because architectural trends are resulting in machines with less memory per core, less relative bandwidth and more on-node parallelism.

We applied six optimizations to LULESH: loop fusion, array contraction, data layout changes, increased vectorization, NUMA aware allocation and allocation of temporaries outside the timestep loop. These changes reduced last level cache misses by over 62%, the global state size of the program by 19%, serial section to less than 0.1% of the overall runtime, and increased the percentage of floating point operations that vectorize by 77%. Tests on five CPU architectures show runtime reductions of up to 57% in serial and speedups of up to 4x in parallel. From the lessons learned tuning LULESH, we are porting changes back to a more complex proxy-app xALE, which contains all the features of the ALE3D hydro. As of this writing, we have improved xALE performance by over 13% for the same problem.

II. TUNING LULESH

To tune LULESH we performed multiple optimizations. Optimizations that help on all architectures are presented first and followed by optimizations that only helped on Linux nodes. Due to space constraints we focus on the BG/Q system and a single problem size, but add in data from Linux clusters and other problem sizes when relevant.

Loop fusion was applied to loops that traverse the same iteration space. By fusing loops we were able to contract arrays that stored data produced in the first loop and consumed in the second loop. We also moved conditional statements performing data-validity checks from where they were in the code to where the data was calculated. These changes reduced last level cache misses by 62% and the maximum amount of data present in memory by 19%. The resulting code contains 12 parallel regions and 12 loops, down from 30 parallel regions and 45 loops. The performance gains from loop fusion are shown in Figure 1 for a BG/Q system. The loop fusion line includes both loop fusion and array contraction optimizations. Runtime reductions start at 6% in serial and become 26% when all 64 threads are use. Strong scaling increases from 25.05x to 31.82x because we reduce the number of memory-bound loops. However, fusion decreased performance for certain loops, but this varies with both problem size and architecture used. Also, fusing loops that vectorize with others that do not results in a loop that does not vectorize.

Allocating temporaries outside of the timestep loop reduces translation lookaside buffer (TLB) misses and allows temporaries to be allocated as arrays of structs. On Linux clusters TLB misses occur due to the first touch allocation policy. When a malloc is called, the operating system sets aside the space requested for the data, but does not setup the virtual to physical mapping of pages. When the first element in a page is touched the operating system then sets up the mapping causing multiple TLB misses. Using large page
sizes also reduces the number of misses and has a similar performance impact. The effect of TLB misses on runtime is largest when most of the cores on a node are used, because the OS handles TLB misses in serial. On BG/Q this optimization also helped performance, however, the extra runtime was not on first touch, but in the serial sections of the code where the allocations and deallocations occurred. The alloc line in Figure 1 shows the performance gains from this optimization when added to the loop fusion. Globally allocating temporaries has a similar runtime impact, about 35 seconds, when four or more processors are used. On Linux clusters this optimization has a larger impact with runtime reductions ranging from 20-50%.

We also changed the data structures used to store the global mesh from a struct of arrays to array of structs. By interleaving four node centered triples, 4 element centered triples and two element centered pairs that only access data in the same loop, we were able to further decrease runtime on BG/Q by 17% on 64 threads. When arrays that only sometimes access data in the same loop are combined performance decreased. This optimization improved performance the most when indirect references are used to fetch data between centerings. For example, when the x, y, and z positions are read from a corner during a element centered loop. However, performance gains occur from interleaving data that are always accessed consecutively (eg., p and q). On Linux clusters we saw runtime reductions of about 10% which are attributed to a decrease in the number of TLB and L1 cache misses. On BG/Q a 21% serial and 45% whole node runtime reduction was achieved. Also, scaling improved from 25.05x to 35.73x.

Vectorization and NUMA aware allocation only helped on Linux clusters. Vectorization due to trouble interacting with the BG/Q compiler and NUMA aware because BG/Q has a flat memory space. NUMA aware allocation uses the Mcsup library [2] which allocates memory to the same socket that it binds tasks to. Unrolling loops and inlining functions calls in the hourglass, stress and kinematics computations, allowed the outer loop of the computation to vectorize resulting in 77% more flops vectorizing for a total of 87%. Increased vectorization reduced overall runtimes by up to 33% and individual loops by up to 50%. Also, NUMA aware allocation produced speedups of at least 20%. Overall a 2-4x speedup was achieved on Linux nodes.

Initial work tuning the CUDA version of LULESH includes fusing the hourglass routine and performing the data allocation changes. The data allocation changes reduce runtime by 21-44%. However, loop fusion of the hourglass increases runtime.

III. CHANGES PORTED INTO XALE

We are applying the lessons we learned to xALE where we have fused 7 loops pairs and rerolled loops to increase vectorization. While tuning xALE we noticed significant differences from LULESH. One obstacle to loop fusion in xALE is how multi-material problems are handled. When a loop over multiple materials is adjacent to a loop over all elements, fusing them requires extra computation and possibly extra data motion to occur. For now we have only fused one set of loops where data motion will not increase. Another difference is xALE’s more complex control flow. Multiple execution paths necessitate moving conditional statements that determine control flow to an earlier point in the control flow to allow certain loops to be fused. Finally, xALE is not written in a high level approach that makes array of struct and struct of array changes easy[3]. Our changes reduced the runtime of the sedov problem on one node eight core node by 13% and increased scalability from 5.45x to 5.89x.

IV. FUTURE WORK

We are continuing to port successful code changes to the GPU code and xALE. We also, plan to explore “lock free” reorderings and how to structure codes to allow compiler tools, such as ROSE, to apply these techniques automatically and/or enable auto-tuning of the code.

REFERENCES


This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.0