Portable Performance of Large-Scale Physics Applications

Toward Targeting Heterogeneous Exascale Architectures Through Application Fitting

William Killian
University of Delaware
Lawrence Livermore National Laboratory
killian@udel.edu

George Zagaris (Advisor)
Lawrence Livermore National Laboratory
zagaris2@llnl.gov

Brian Ryujin (Advisor)
Lawrence Livermore National Laboratory
ryujin1@llnl.gov

Brian Pudliner (Advisor)
Lawrence Livermore National Laboratory
pudliner1@llnl.gov

John Cavazos (Advisor)
University of Delaware
cavazos@udel.edu

ABSTRACT

Physics simulations are one of the driving applications for supercomputing and this trend is expected to continue as we transition to exascale computing. Modern and upcoming hardware design exposes tens to thousands of threads to applications, and achieving peak performance mandates harnessing all available parallelism in a single node. In this work we focus on two physics micro-benchmarks representative of kernels found in multi-physics codes. We map these onto three target architectures: Intel CPUs, IBM Blue Gene/Q, and NVIDIA GPUs. Speedups on CPUs were up to 12x over our baseline while speedups on Blue Gene/Q and GPUs peaked at 40x and 18x, respectively. We were able to achieve 54% of peak performance on a single core. Using compiler directives with additional architecture- aware source code utilities allowed for code portability. Based on our experience, we list a set of guidelines for programmers and scientists to follow towards attaining a single, performance portable implementation.

1. SUMMARY

Physics simulations are one of the driving applications for supercomputing and this trend is expected to continue as we transition to exascale computing. At the same time, advances in modern hardware design and architecture expose tens to thousands of threads to applications for computation. Single node performance is crucial in harnessing the massively parallelism available and constitutes a defining characteristic towards exascale computing.

1.1 Mesh Overview

We are optimizing two physics kernels which operate on three-dimensional meshes. There are a handful of properties unique to these three-dimensional meshes. These properties restrict our work to kernels with similar data layout and access patterns. A mesh consists of nodes with zones existing between the node lattice. This mesh has many properties for each zone, such as volume, density, and position. There exists a layer of phony zones around the entire structure. Multi-material zones are all grouped together while single-material zones have stride-one access.

1.2 Architecture Mapping

We map these applications onto three target architectures: Intel CPUs, IBM Blue Gene/Q, and NVIDIA GPUs. There are properties of an architecture to consider when mapping an application. First, we consider properties of the memory hierarchy. How much cache does the system have? How much data fits into a single line of cache? If the architecture has configurable cache, should we use the cache? Then we consider the computation capability. How many different instructions can be issued at the same time? Does the architecture support advanced features such as vectorization? Are instructions issued in order? Finally we analyze thread-level parallelism. How many threads can run concurrently? Can we modify the workload to ensure there is enough work to be done per core?

1.3 Code Modifications

From the properties of the architectures and expert knowledge, we propose and apply source code modifications to better target different architectures. These modifications are designed to be platform portable which allows for high code reuse even when targeting CPUs and GPUs.

- *Improving data alignment*: Alignment enables better code generation and improves cache performance [2, 1]. Usually this is hidden from the programmer, but there are built-in functions and source code directives available across various compilers. Data from both the stack and heap can be aligned to best fit the architecture.
- *Eliminate common sub-expressions*: When initially designing algorithms, we rarely consider re-computation or elimination of sub-expressions. After modifications, we reduce the inner loop of one kernel by 51 memory accesses and 30 floating-point operations.

- *Loop decoupling*: Sometimes nested loops have inter-loop dependences. By eliminating these dependences, individual loop iterations can safely be run in parallel. This provides over a 2x speedup to our one kernel.

- *Compiler hints and optimizations*: . The compiler may know more about the target architecture than most programmers. Passing a good set of options to the compiler along with the first three modifications will help achieve speedups [2, 1].

- *Workload parallelization*: CPUs implementations are parallelized with OpenMP [7](version 4.0 if supported) and NVIDIA GPUs are targeted through CUDA [4].

1.4 Kernel Descriptions

StressWork is a small, bandwidth-limited microkernel consisting of 19 floating-point operations for each zone, with an additional 20 floating-point operations performed for each mixed zone. Each loop iteration is independent, so the overall algorithm is embarrassingly parallel. A key code modification improving speedup is making the iterations over the mixed zones completely independent. This change provides improved throughput across all architectures. Alignment, loop decoupling, and compiler optimizations are applied to *StressWork*.

StressAcc is another small bandwidth-limited microkernel consisting of 180 floating-point operations for each zone, updating the acceleration values for each neighboring node. When parallelizing this algorithm, accelerations are calculated for each zone and update each neighboring node in a thread-safe manner. To ensure thread safety, we precalculated index lists to group conflict-free updates when targeting CPUs; we use atomic operations when targeting the GPU. There is an approximate 50% penalty when using atomic operations on the GPU, but greatly minimized code specialization of the computation kernel. Alignment, flop reduction, and compiler optimizations and hints are applied to *StressAcc*.

1.5 Performance Results

When executing the *StressWork* kernel, we observe speedups up over 16x over baseline on the 8-core Intel SandyBridge. Over a 35x speedup is observed when running on Blue Gene/Q. Both Intel and IBM architectures exhibit good strong scaling for *StressWork*. Speedups on NVIDIA K20Xm slowly increase up to over 20x over baseline then vary between 18x and 21x when performing weak scaling. Both CPU architectures exhibit good strong scaling while the GPU architecture initially exhibits good weak scaling.

We observe speedups up over 8x over baseline on 8 cores when running *StressAcc* on Intel SandyBridge. Over a 45x speedup is observed when running on Blue Gene/Q. Both Intel and IBM architectures exhibit good strong scaling for *StressAcc*. Speedups on NVIDIA K20Xm remained relatively constant at 12x over baseline when performing weak scaling.

We also analyze the peak performance of the kernels running on all architectures. *StressWork* running on SandyBridge had the highest throughput at 50 GFLOP/s. *StressAcc* had the highest throughput when running on the NVIDIA K20Xm. The kernels best mapped to Intel SandyBridge with 30.2% and 16.58% of peak on multicore and 54.69% and 16.9% of peak on single core observed. Both kernels will see performance improvements from new architectures features such as on-chip memory and high bandwidth memory (HBM) [3, 6].

Acknowledgement

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-ABS-675717

2. REFERENCES

[1] IBM Corporation. Optimization and programming guide for blue gene/q. Online, 2012. http://www-01.ibm.com/support/docview.wss?uid=swg27027069&aid=1.

[2] Intel Corporation. Vectorization essentials. Online, 2014. https://software.intel.com/en-us/articles/vectorization-essential.

[3] Lawrence Livermore National Laboratory. Coral/sierra. Online, 2014. https://asc.llnl.gov/coral-info.

[4] NVIDIA Corporation. About cuda. Online, mar 2015. https://developer.nvidia.com/about-cuda.

[5] NVIDIA Corporation. Profiler user guide. Online, jun 2015. http://docs.nvidia.com/cuda/profiler-users-guide/axzz3hQArd89w.

[6] Oak Ridge National Laboratory. Summit. Online, 2014. https://www.olcf.ornl.gov/summit/.

[7] OpenMP Architecture Review Board. Openmp application program interface. Online, jul 2013. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.