Portable Performance of Large-Scale Physics Applications

Toward Targeting Heterogeneous Exascale Architectures Through Application Fitting

William Killian[1,2], George Zagaris[2] (Advisor), Brian Ryujin[2] (Advisor), Brian Pudliner[2] (Advisor), John Cavazos[1] (Advisor)

[1] University of Delaware, [2] Lawrence Livermore National Laboratory

University of Delaware — College of Engineering — Department of Computer & Information Sciences

Lawrence Livermore National Laboratory

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.

Three-Dimensional Physics Mesh Application Overview

We are optimizing two physics kernels which operate on three-dimensional structured grids. These applications are representative of kernels found in large scale multi-physics codes.

- Each mesh consists of nodes with zones existing between the node lattice
- The mesh has many properties for each zone, such as volume, density, and position
- The mesh has a layer of ghost (or phony) zones around the entire structure
- Multi-material zones are all grouped together while single-material zones have stride-one access

Application Fitting on Advanced Architectures

There are a few critical properties of an architecture to consider when mapping an application. We considered the following characteristics:

Memory Hierarchy

- What is the cache configuration?
- Should we use the cache (if applicable)?

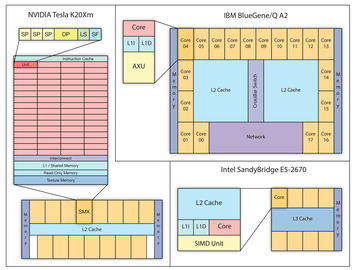
Computation Capability

- Instruction issue width: How many different computations can be issued at the same time?
- SIMD/SIMT execution: Does the architecture support advanced features such as vectorization?

Parallelism

- How many threads can run concurrently?
- How should work be distributed on the architecture?

Architecture Overview for Targeted Systems



Code Modifications Toward Good Architecture Fitting

Data Alignment - Alignment enables better code generation and improves cache performance. We abstracted away compiler-specific hints to a common programming model to aid with data alignment.

Decouple Loops - Removing inter-loop dependences help improve scalability. Both kernels were modified to take advantage of independent loop iteration access.

Reduce Floating-Point Operations - We should consider elimination of sub-expressions. After modifications, we reduce the inner loop of one kernel by 51 memory accesses and 30 floating-point operations.

Parallelize with High-Level Programming Models - OpenMP is used when running on Intel SandyBridge and IBM Blue Gene/Q while CUDA is used when targeting NVIDIA GPUs. Code portability is maintained.

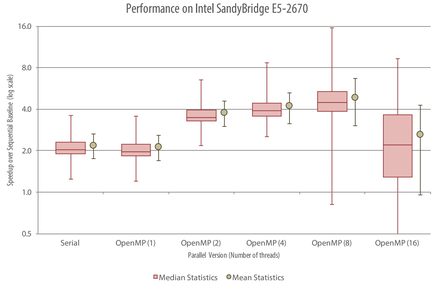
Compiler Hints and Optimizations - Manually optimizing an application can be hard. We use many best-practice compiler optimizations specified by the vendor for each architecture.

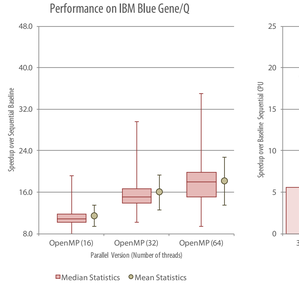
Experiment Configuration and Execution

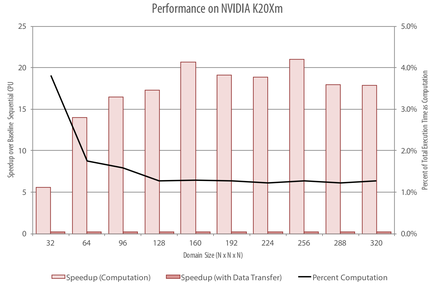
Each microbenchmark has a variety of input parameters. For performance evaluation, we consider all of the input permutations. There are 1080 versions of each workload running on each CPU execution environment. When running on GPUs, we fixed all of the parameters except those that change access patterns resulting in 18 versions.

We plot the speedups based on the execution environment (number of cores) on CPU architectures to exhibit strong scaling. When running on GPUs, we adjust the domain size to exhibit weak scaling. Box plots (median statistics) and average and standard deviation (mean statistics) show speedups observed during execution.

StressWork - An Embarrassingly Parallel Multi-Material Kernel







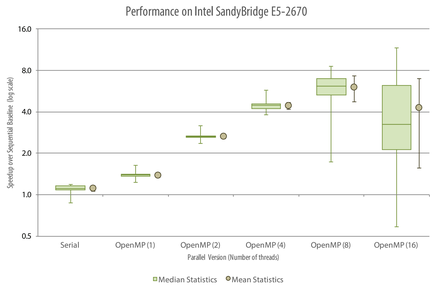
Kernel Description

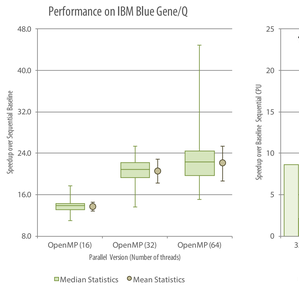
- 19 FLOPs for each single-material zone and 20 FLOPs for each mixed-material zone
- 18 input arrays and one output array
- Each loop iteration is independent

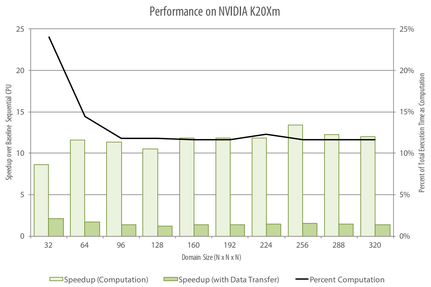
Performance Analysis

- Code changes yield up to 3.8x speedup on a single core supports vectorized code
- Large variance of speedup suggests access pattern greatly impacts performance
- Data transfer dominates GPU execution times
- 16x speedup on single-socket 8-core Intel, 36x on Blue Gene/Q, and 21x on GPU

StressAcc - A Stress Tensor Kernel with Race Conditions







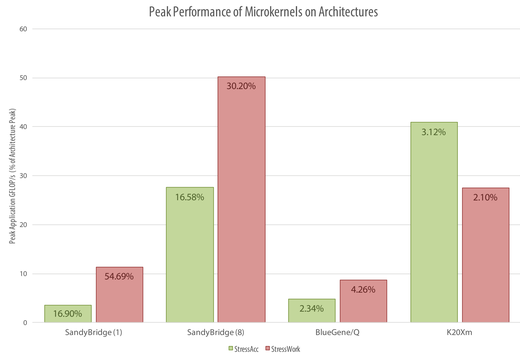
Kernel Description

- 180 FLOPs for each zone, and zones update neighboring nodes (race condition)
- Because of the race condition present, independent access lists are precalculated on CPUs. Atomic reduction operations are use when targeting GPUs
- Six input arrays and three output arrays

Performance Analysis

- 1.3x speedup observed when running on a single CPU core
- Small variance of speedup suggests varying input doesn't impact speedup
- Real speedup still observed even with data transfer on GPU
- 9x speedup on single-socket 8-core Intel, 45x on Blue Gene/Q, and 13.4x on GPU

Performance Evaluation - Using Parallel Languages to Scale



- StressWork fit to SandyBridge very efficiently (up to 55% of peak on a single core)
- Peak performance of Blue Gene/Q was much lower than anticipated
- Although the performance relative to peak performance of StressACC was very low (3%) the raw GFLOPs performance was highest on NVIDIA GPUs.
- Application profiling showed full memory bandwidth utilization on all architectures.

Conclusion

- StressWork inputs greatly vary the workload while StressAcc is consistent
- Using portable design principles results in kernel speedups on all architectures
- We apply minimal code modifications to support "write once, run anywhere"
- SIMD code generation improved execution on single-core Intel SandyBridge
- GPU speedups similar to Intel CPU speedups and computation throughput
- Applications are memory-bound on all architectures

Future Work

Newer Architectures

- Current limitations of targeting the existing architectures is memory bandwidth being fully utilized. Without reformulating the application, improvements will come from upcoming advancements in architecture.
- Newer architectures will have additional layers of memory (eDRAM) or high-bandwidth memory (HBM) to increase the memory bandwidth feeding computations.

Library Encapsulation

- There is already high code reuse across architectures, but seperate files are required
- A library should easily abstract away the execution policy and data transfer of a kernel
- The C++ Parallelism Technical Specification (C++ Parallelism TS) could be used as a starting point with CUDA support added for targeting NVIDIA GPUs.

Optimize for Bandwidth

- Most optimizations target performance improvement through cache fitting and minimizing execution time and energy.
- Considering bandwidth throughput as a guide for optimization selection could lead to better overlap of data and compute and reshape access patterns.

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

LLNL-POST-678102