RAJA Performance Suite

Summary Version

Tagged version 0.2.3 in GitHub project https://github.com/LLNL/RAJAPerf

Please use the gzipped tarfile at **the link named for the release** located here: https://github.com/LLNL/RAJAPerf/releases (v0.2.3)

Note: v0.2.3 contains some minor changes to the output files based on initial feedback from vendors. There are no other changes to the Suite.

Purpose of Benchmark

The intent of the RAJA Performance Suite is to compare execution performance between baseline and RAJA variants of various loop kernels. The main goals of the Suite are:

- To assess performance of executable code generated by compilers (including different versions) to determine: which RAJA C++ abstractions present optimization challenges for compilers, which combinations of C++ constructs and parallel programming models (e.g., OpenMP) are problematic, etc.
- To serve as a source of kernels ready to share with compiler teams when performance issues are observed so those issues can be addressed.
- To verify that resolved issues do not re-appear in newer compiler versions.
- To monitor performance changes as RAJA evolves and assess performance of new RAJA features.

Characteristics of Benchmark

The RAJA Performance Suite contains a collection of loop kernels. Each kernel appears in RAJA and non-RAJA variants using various parallel programming models, such as OpenMP and CUDA. The non-RAJA variant of a kernel for each programming model is considered the "baseline" against which the RAJA variant should be compared. The kernels are drawn from a variety of sources, including HPC benchmark suites and real applications to cover a wide range of numerical algorithm patterns. The code is written in ISO standard C++ 11.

The Suite is designed to easily run different performance experiments in a simple, script-driven manner. In particular, the Suite has various run-time configuration options, controlled via command line options. These options allow one to: select which kernels and variants to execute, choose sizes of loop iteration spaces, number of times kernels are run, which reference variant to use for speedup reports, etc. The Suite can also be compiled with individual variants enabled and disabled, which may be helpful to inspect assembly code to identify issues that result when different programming models are combined in a single source file.

The README.md file which is visible on the GitHub project page, at the URL provided above, contains instructions for building, running, and adding kernels and variants to the suite.

Mechanics of Building Benchmark

See the RAJA Perf Suite README.md file which is visible of the RAJA Perf Suite GitHub project page for instructions on retrieving the code and building the code.

Specifically, instructions for cloning the RAJA Performance Suite repo and building the suite can be found at https://github.com/LLNL/RAJAPerf - building-the-suite. Note that the ‘scripts’ directory, which use CMake cache files in the ‘host-configs’ directory are useful references for selecting CMake options and compilers we use on Livermore Computing platforms.

Mechanics of Running Benchmark

After configuring and building the RAJA Performance Suite, the Suite is run by executing the executable in the build directory. Due to the way our build system is set up currently, we generate two executables. The executable ./bin/raja-perf.exe contains all CPU and CUDA kernel variants. The executable ./binraja-perf-nolibs.exe contains all CPU and OpenMP target variants. This will change in a future release so that there is only one executable generated that contains all variants.

Passing the ‘–help’ (or ‘-h’) option to the executable will output a summary of all command line options, which describes how to run the Suite in desired configurations.

When the Suite is run, several output files are generated (mostly comma-separated-value text files). These files report execution timings for each kernel variant, speedups compared to a reference variant, kernel output checksums, and FOM (figure of merit). The name of each output file is descriptive of its contents.

Figure of Merit (FOM) Output Report:

The figure of merit for each kernel is a relative speedup (or slowdown) of the RAJA variant for a given programming model back-end vs. the baseline kernel variant for that programming model. In the FOM report, each RAJA variant that is slower than its corresponding baseline for a particular programming model, by a given tolerance, is marked as “OVER_TOL” in the report in the column to the right speedup (or slowdown) value for the RAJA variant. The default tolerance is 10%. The tolerance can be changed via a command line option. Run the executable with the ‘-h’ option to see how this is done.

We would like vendors to report FOM numbers for kernel variants in the Suite associated with each programming model that is relevant to their architecture, including sequential variants. FOM should be reported for the default kernels sizes in the Suite, which are chosen to be representative of real application kernels.

Ideally, we would like each RAJA variant to be no slower than its corresponding baseline variant. However, issues related to compiler optimizations and as well as internal RAJA implementations likely makes this impossible at present. Nevertheless, we view this Suite as a critical communication tool in interactions between DOE Laboratories and compiler vendors.

When certain kernels and variants are reported as 'OVER_TOL' according to the tolerance value, we would like vendors to analyze these cases. Specifically, we are interested in details about why this happens. For example,

- What is the cause of compiler optimization issues that is causing the RAJA variant to run slower? Is optimization hindered due to:
  - Choices made in the compiler implementation or interpretation of C++ language features?
  - Choices made by the RAJA team in its use of C++ language features?
- Does the combination of C++ and a parallel programming model hinder performance? For example, is this due to:
  - Choices made in compiler implementation or interpretation of the programming model standard (if applicable; e.g., OpenMP) or in the way the programming model is designed to work?
  - Choices made by the RAJA team in using the programming model?
- Etc.

We are genuinely interested in understanding the root causes of performance differences. Our goal is to improve RAJA as well as compilers. When there are perceived issues with RAJA, we would like to learn what vendor analysis reveals. When compiler shortcomings are revealed, we are interested in estimates of what can be done to improve compiler support for RAJA kernel by machine acceptance time.

Benchmark Verification:

The checksum report can be used to determine whether all kernel variants have generated correct results. The checksum for each executed variant is reported as well as the difference between it and whichever reference version is used. All kernels are set up so that a checksum of zero indicates that a variant did not execute for some reason. Note that, due to differences in compiler optimizations, order of operations for parallel variants, etc. the checksum differences are not expected to be identically zero in all cases. Depending on the kernel variant and how it is run, a checksum difference of < 10e-6 (most of the time the observed difference is much less than this) is considered a correct result.

Figure of Merit Data on BG/Q

Due to what the RAJA Performance Suite is designed to represent (RAJA usage patterns in LLNL application codes) and assess (execution timings between RAJA kernel variants and non-RAJA (baseline) kernel variants), Figure of Merit data on BG/Q is not really relevant moving forward.

Vendor supported compilers on that architecture do not support all C++ 11 features required to compile the RAJA Performance Suite. The only viable compile that we have access to for the BG/Q architecture is the clang compiler that has been ported to it. This is not an IBM-supported compiler and so it is not clear how well it is optimized for the BG/Q architecture.