Discovering Optimal Execution Policies in KRIPTK using RAJA

William K. Killian1,2, Adam J. Kuenen2 (Advisor), Ian Karlin2 (Advisor), John Cavazos1 (Advisor) 1 University of Delaware 2 Lawrence Livermore National Laboratory

Motivation
- Legacy physics applications need updating to run well on newer architectures but are not always designed for architecture flexibility
- With architectures changing frequently (multicore, many-core, GPU), applications need to be adaptable to many different architectures.
- Adaptive, flexible programming layers are necessary to intelligently search large optimization spaces.

KRIPTK
- KRIPTK is a proxy application for Sn particle transport developed at LLNL
- Highly dimensional: composed of directions, groups, zones, and moments
- Many possible nestings of data and execution. Difficult to find the best
- Solves the linear Boltzmann equation using sweeps over a 3D domain space
- Goal: find optimal execution policies for common configurations of KRIPTK

RAJA Performance Portability Layer
- Provides C++ abstractions to enable architecture portability
- Predefined execution policies exist for SIMD, OpenMP, and CUDA
- Nested and advanced loop transformations (tiling, reordering) are available
- Goal: use RAJA to drive optimization search exploration for KRIPTK

Example RAJA Execution Policy to apply
```cpp
PolicyDescription and Generation
Policy Search Space
- Four execution policies: sequential, SIMD, OpenMP, collapsed OpenMP
- Five tiling policies: no tiling and fixed tiles of sizes 8, 32, 128, and 512
- Considered only loop valid nests, tiles must fit in L3 cache, no nested thread parallelism, OpenMP clauses only with OpenMP loop nests
- Policies are generated for each independent loop nest
- Five different loop nests:
  1. LTimes | L | - 4-nested loop with 850K versions
  2. LPlusTimes | L^2 | - 4-nested loop with 850K versions
  3. Scattering | L^2 | - 4-nested loop with 850K versions
  4. Sweep | H^2 | - 3-nested loop with 2.9K versions
  5. Source | Q | - 2-nested loop with 0.45K versions

Optimization Space Exploration
- Assume kernel executions are independent of one another
- Too costly to run each execution policy for a larger Sn transport code.
- We propose two different strategies to explore the optimization space
- Goal: find optimal execution policies of kernels without exhaustive execution

Hill-climbing Strategy
- Version V ← all versions of KRIPTK
- F ← all features of a loop nest count < 0 do while count < threshold
  p ← rand(V)
  best ← p
  foreach i, f shuffle (enumerate(F)) do
    foreach option ∈ F_i do
      p_i ← option
      count ← count + 1
      if (p_i) < (time(best)) best ← p_i end end
- Limited to 10% of total search space
- Speedup up to 3.1% over baseline

Subspace Search Strategy
- Version V ← all versions of KRIPTK
- F ← all features of a loop nest count < 0 do while count < threshold
  p_i 1 ← rand(V)
  foreach i, f shuffle (enumerate(F)) do
    foreach option ∈ F_i do
      p_i 2 ← option
      count ← count + 1
      if (p_i 2) < (time(best)) best ← p_i 2 end end
- Limited to 20% of total search space
- Speedup up to 25.3% over baseline

Performance Analysis
Exhaustive Execution
- Architecture: dual-socket Intel Xeon E5-2670, 32GB DDR3 RAM
- Compiler: Clang 3.8.0 with OpenMP support (-03 -march=native)

Comparison to Exhaustive Execution
- To evaluate our search strategies, we ran all generated versions of KRIPTK.
- The best discovered policies improves over the baseline performance of the entire KRIPTK proxy application by 19.5%.
- Hill-climbing achieves up to 95.6% of optimal performance while subspace search achieves up to 98.8% of optimal performance.

Conclusion and Future Work
- Used the RAJA performance portability layer to explore a large optimization space efficiently within the KRIPTK Sn transport proxy application
- Two different search space strategies can yield results up to 98.8% of optimal while only exploring 20% of the total search space.
- The best known execution time of KRIPTK improves by 19.5%.

Future Work
- Expand results to include GPU execution policies (NVIDIA Kepler/Pascal) and nested parallelism with many-core (Intel Knights Landing) architectures
- Augment tiling policies to include multi-level tiling. This will be useful when targeting future architectures with complex memory hierarchies.
- Construct an accurate control-flow graph-based performance prediction model. The predictor replaces exhaustive execution with only compilation.

Acknowledgments and Resources

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