Discovering Optimal Execution Policies in KRIPKE using RAJA

William K. Killian
University of Delaware,
Newark, DE
Lawrence Livermore National
Laboratory, Livermore, CA
killian@udel.edu

Advisor: Adam J. Kunen
Lawrence Livermore National
Laboratory, Livermore, CA
kunen1@llnl.gov

Advisor: Ian Karlin
Lawrence Livermore National
Laboratory, Livermore, CA
karlin1@llnl.gov

Advisor: John Cavazos
University of Delaware,
Newark, DE
cavazos@udel.edu

ABSTRACT

Newer architectures create application porting problems for legacy physics applications. With architectures changing frequently (multicore, many-core, GPU), applications need to be adaptable to many different architectures. "KRIPKE is a proxy application whose primary purpose is to research how data layout, programming paradigms, and architectures affect the implementation and performance of Sn transport" [4]. We create policy generation tools that emit RAJA policies to generate over 850,000 versions of KRIPKE. RAJA provides a performance portability layer with high-level C++ constructs to allow its users to quickly change execution policies for CPUs and GPUs [3]. We use hill-climbing and subspace search strategies to explore this space without the need for exhaustive search. With hillclimbing, we explore 10% of the total versions while achieving 95.6% of optimal performance. The subspace search strategy yields 98.8% of optimal while only exploring 20% of all versions. Overall, we improved the best existing version of KRIPKE by 19.5%.

1. INTRODUCTION AND BACKGROUND

Reaching exascale power and performance targets requires many-core or hybrid architecture designs. Achieving good application performance across these architectures often requires invasive source code transformations and optimizations. Many legacy applications were developed in a stable time of computing with single-core nodes offering scalability with MPI. With the growing diversity of high performance computing node architectures, codes now need to be written to adapt various architectures easily.

One application developed under this stable environment is ARDRA [2]. ARDRA is an Sn transport code with a fixed data layout and execution policies developed in C++. As newer architectures became prominent over the development period, manually intensive source code changes had to be made to achieve good performance. KRIPKE was developed as a proxy application of ARDRA to explore more flexible data layout and execution policies.

Language abstraction layers such as Kokkos [1] and RAJA [3], are being developed to address the inflexibility of current programming models. In this work we use the RAJA version of KRIPKE to explore the performance impacts of execution policies and loop transformations. KRIPKE contains five loops representative of the data and compute challenges found within ARDRA; however, even with the small number of loops the exhaustive optimization space includes over 10^{24} versions. To practically tackle this problem requires intelligent search strategies and reasonable assumptions to find near-optimal execution policies and data layout for KRIPKE.

2. POLICY GENERATION AND SEARCH

Kernel	Loop Size	Versions Generated
LTimes [L]	4	850,000
LPlusTimes [L^+]	4	850,000
Scattering [Σ_s]	4	850,000
Sweep [H^{-1}]	3	2,900
Source [Q]	2	450

Table 1: Kernel descriptions for KRIPKE

$$\Psi_{i+1} = H^{-1}L^{+}\left(\Sigma_s L\Psi_i + Q\right) \qquad (1)$$

There are five loop nests present within KRIPKE which are used to solve Equation 1. Although KRIPKE can have one of six unique data layouts, we only consider one data layout for this work. Our optimization search space defined by the following options:

- **Execution Policies:** sequential, SIMD, OpenMP, and collapsed OpenMP.
- **Tiling Policies:** none and fixed tiling sizes of 8, 32, 128, and 512.
- **Loop Reordering:** the order of the loops in a single loop nest can be permuted.
- **Kernel Independence:** each loop nest is assumed to have no performance side effects.

Policies are independent for individual loop nests. Each loop in the loop nest has exactly one execution policy and one tiling policy. Each loop nest has a single permutation applied. Table 1 outlines the kernels, their loop size, and the total number of generated versions.

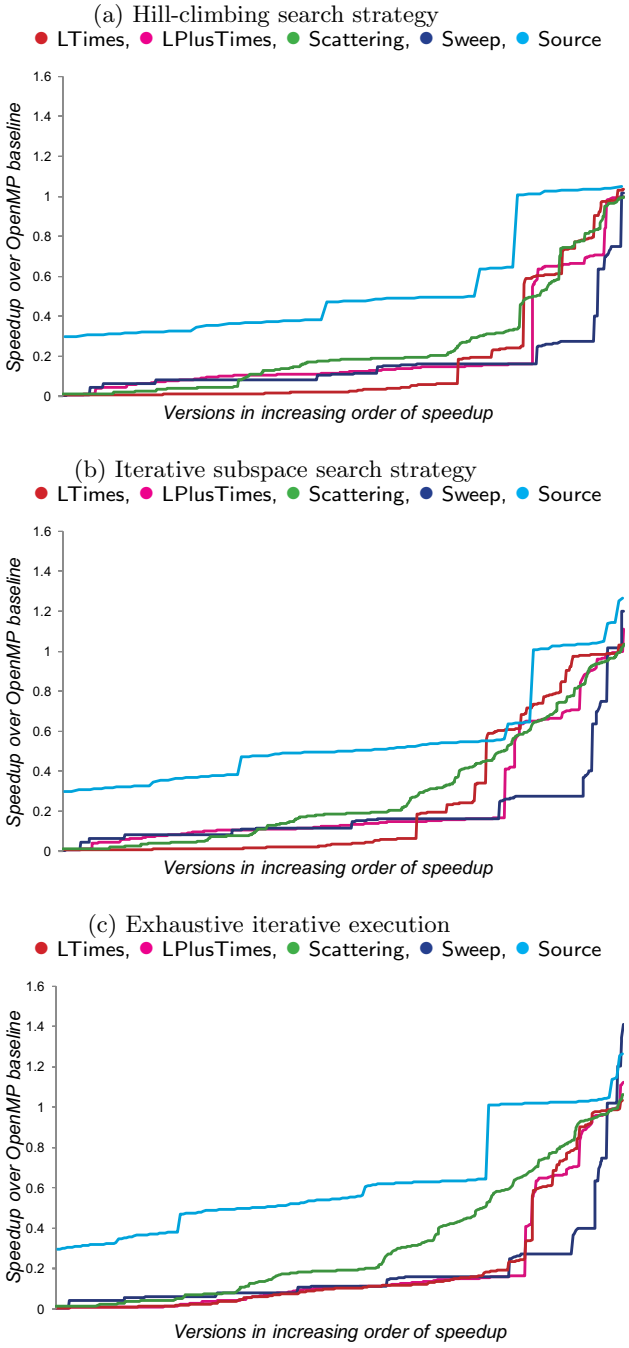

Figure 1: Speedup curves for both search strategies and exhaustive iterative execution

Our goal is to find optimal execution policies of kernels without exhaustive execution. With 850,000 versions in our restricted search space, it is too expensive to exhaustively execute versions of a complete physics code. We propose two different search space strategies, hill-climbing and iterative subspace search, to search the optimization space. We limit hill-climbing to explore up to 10% of the total space. Iterative subspace search will explore at least $1/k$ (where k is the largest number of options for any feature) which creates a lower bound of exploration to 20% of the total space.

3. PERFORMANCE ANALYSIS

We evaluate the performance of KRIPKE on a dual-socket Intel Xeon E5-2670 with 32GB of DDR3 RAM compiled with Clang 3.8.0. Figures 1a and 1b show ordered speedup curves for each kernel with hill-climbing and iterative subspace search respectively. We observe up to 3.1% speedup with hill-climbing and 25.3% speedup with subspace search over the baseline kernels.

Figure 1c shows sorted speedup curves for all versions of all kernels. The best discovered policies of each kernel improves over the baseline performance of the KRIPKE proxy application by 19.5%. Individual kernels within KRIPKE achieve up to a 40% speedup. Hill-climbing achieves up to 95.6% of optimal performance while subspace search achieves up to 98.8% of optimal performance.

4. CONCLUSION AND FUTURE WORK

We used the RAJA Performance Portability Layer to easily explore a large optimization search space efficiently within KRIPKE. We show that using two different search space strategies can yield performance up to 98.8% of optimal while exploring 20% of the total search space. Our future work includes:

- Include data layout as a search space parameter
- Expanding results to include GPU execution policies and nested parallelism with many-core architectures
- Augmenting tiling policies to include multi-level tiling
- Constructing an accurate control-flow graph-based performance prediction model. The predictor replaces exhaustive execution with only compilation.

5. REFERENCES

[1] Carter Edwards, H., Trott, C. R., and Sunderland, D. Kokkos. *J. Parallel Distrib. Comput. 74*, 12 (Dec. 2014), 3202–3216.

[2] Hannebutte, U., and Brown, P. Ardra: Scalable Parallel Code System to Perform Neutron–and Radiation–Transport Calculations. Tech. Rep. UCRL-TB-132078, Lawrence Livermore National Laboratory, 1999.

[3] Hornung, R., and Keasler, J. The RAJA Portability Layer: Overview and Status.

[4] Kunen, A., Bailey, T., and Brown, P. KRIPKE-A Massively Parallel Transport Mini-App. Tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2015.

6. ACKNOWLEDGEMENTS

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-ABS-698560)