A RAJA-based Tuning Framework for Multi-Platform Performance Prediction

William K. Killian[1,2], Adam J. Kunen[2], Ian Karlin[2], John Cavazos[1]   [1] University of Delaware [2] Lawrence Livermore National Laboratory

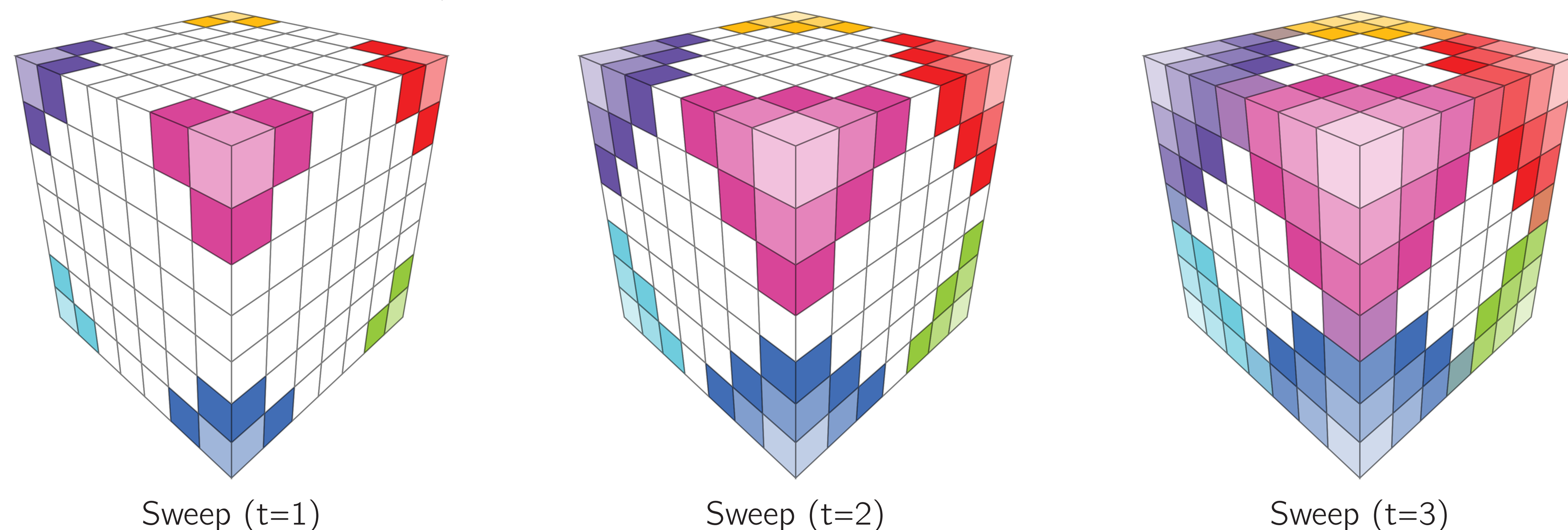
Motivation

- 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.

KRIPKE

- KRIPKE 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 Bolzmann equation using sweeps over a 3D domain space
- **Goal: find optimal execution policies for common configurations of KRIPKE**

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



Time sequence of the sweep kernel (H^{-1}) moving through the mesh. Multiple sweeps can occur at the same time. Grid contention occurs when a location has equal manhattan distance from two or more sources (corners).

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 performance prediction for KRIPKE**

Example RAJA Execution Policy to apply

```
NestedPolicy<
 ExecList<
  seq_exec, seq_exec,
  omp_for_nowait_exec, simd_exec>,
 OMP_Parallel<
  Tile<
   TileList<
    tile_none, tile_none,
    tile_none, tile_fixed<512>>,
   Permute<PERM_JIKL>
  >
 >
>
```

Basic loop implementation

```
for d in range(0,dom<IDirection>(id)):
  for nm in range(0,dom<IMoment>(id)):
    for g in range(0,dom<IGroup>(id)):
      for z in range(0,dom<IZone>(id)):
```

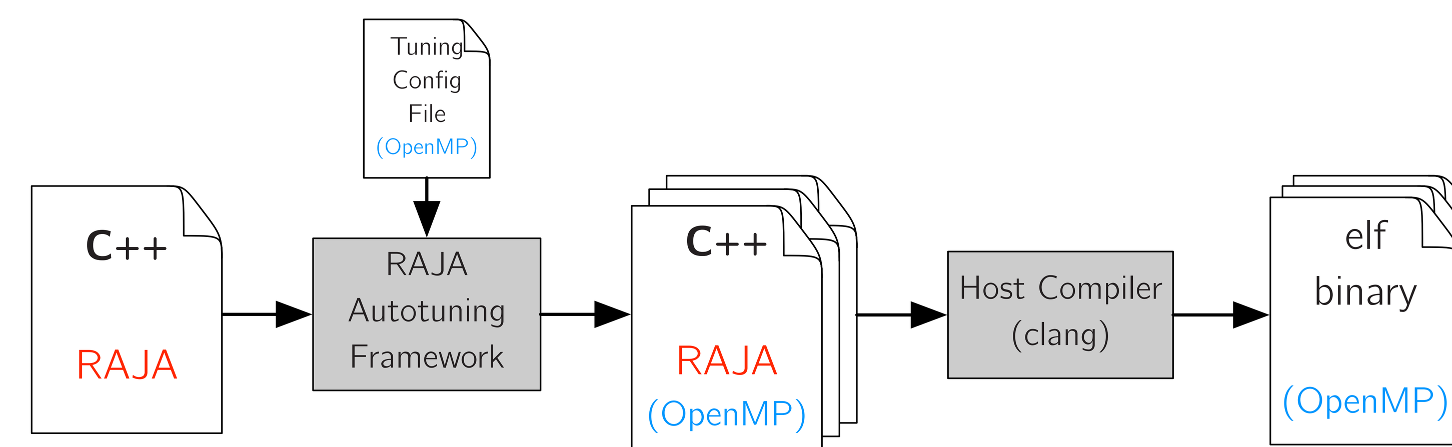
Nested Policy applied to loop

```
#pragma omp parallel
for z2 in range(0,dom<IZone>(id),512):
  for d in range(0,dom<IDirection>(id)):
    for nm in range(0,dom<IMoment>(id)):
      #pragma omp for nowait
      for g in range(0,dom<IGroup>(id)):
        for z in range(z2,z2+512):
```

Policy Description and Generation

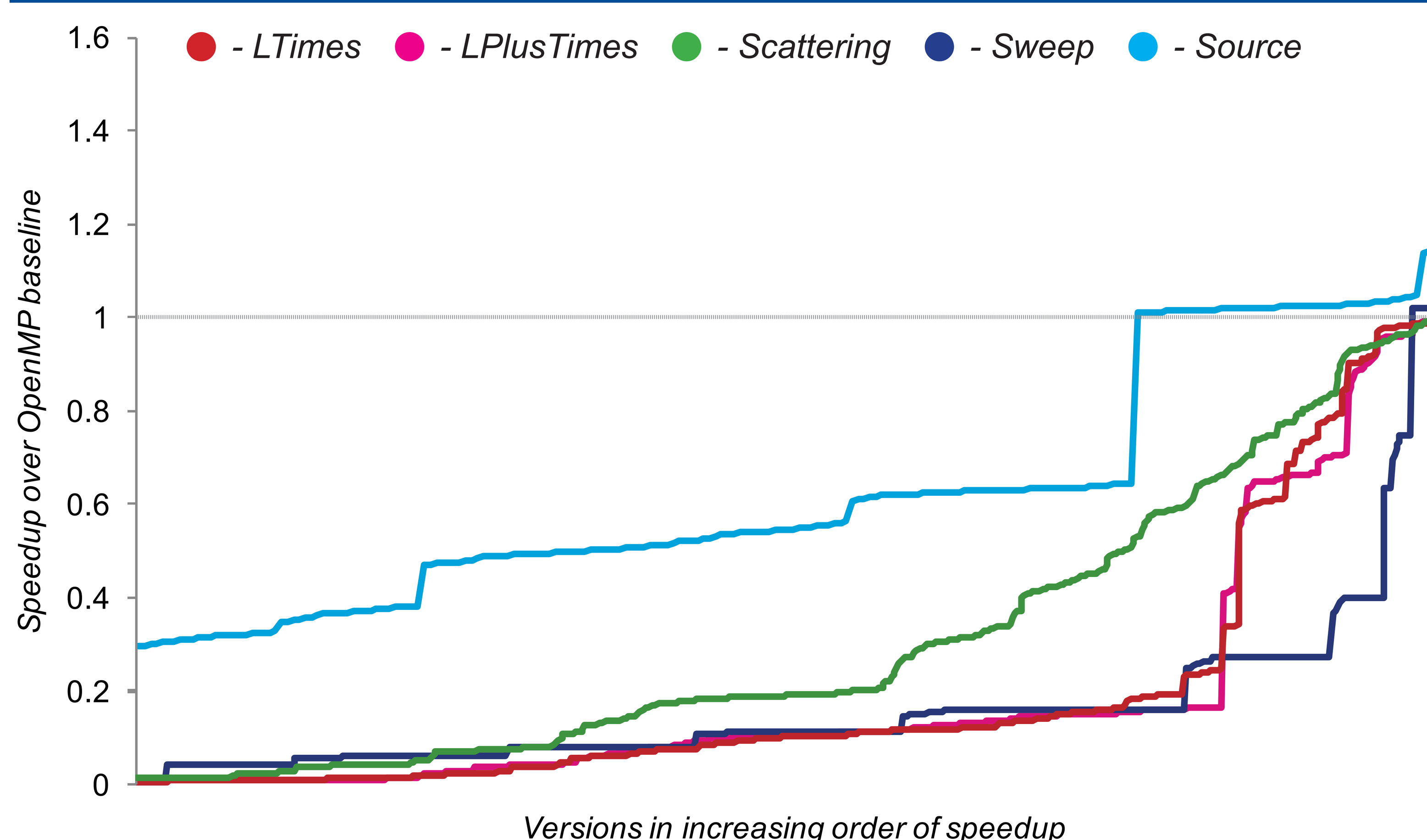
- Execution policies: sequential, SIMD, OpenMP, collapsed OpenMP
- Tiling policies: no tiling and fixed tiles of sizes 8, 32, 128, and 512
- Constraints: standard language conformance, tiles must fit in L3 cache, no nested parallelism
- Policies are generated for each independent loop nest
- Kernel loop nest size: three 4-nested, one 3-nested, one 2-nested)

RAJA as a Tuning Framework



- Each loop nest is identified and re-written as a RAJA *forallN* loop
- The tuning configuration file describes the limitations above in a JSON format that the framework can process
- The framework will generate all accepted versions of each file and generate a corresponding header file for a loop nest, representing each applied policy as a type (e.g. `NestedPolicy`)
- Each version can then be compiled with a C++11 enabled compiler

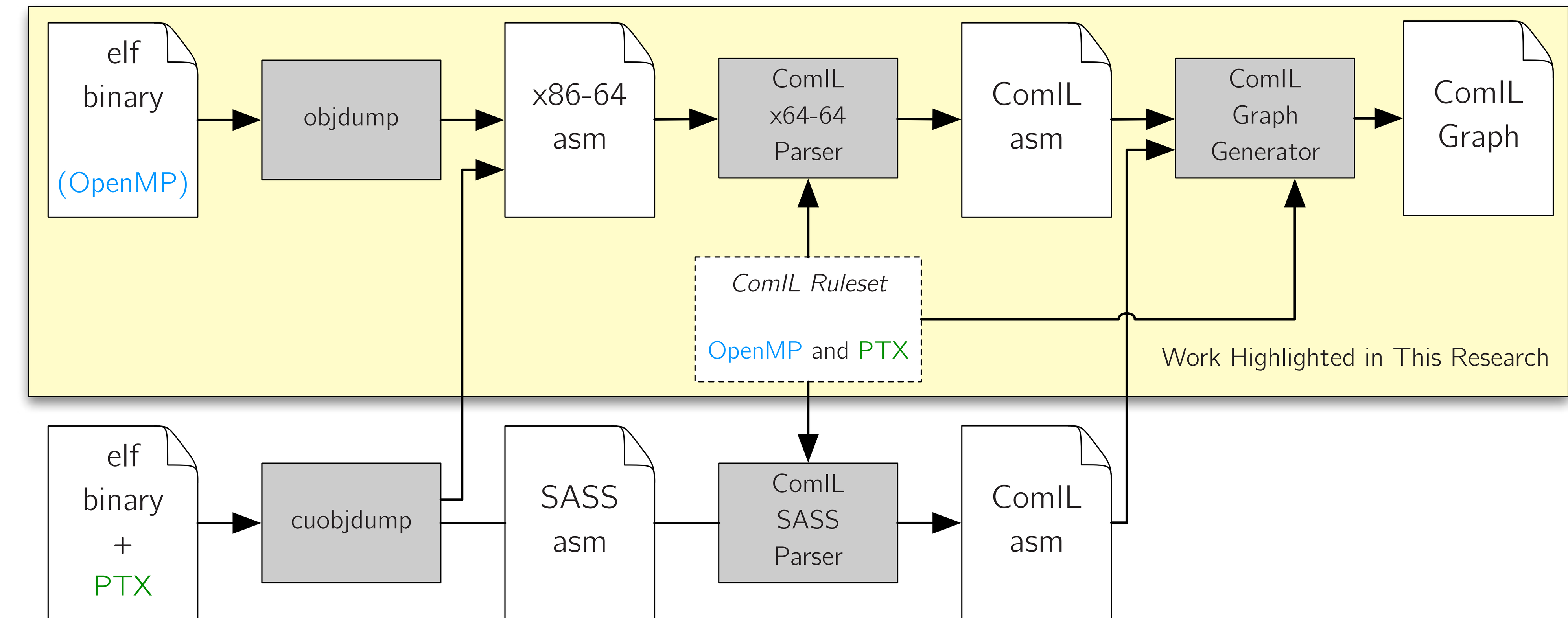
Performance Analysis



- **Architecture:** dual-socket Intel Xeon E5-2670, 32GB DDR3 RAM
- **Compiler:** Clang 3.8.0 with OpenMP support (`-O3 -march=native`)
- The best independently discovered policies yields an overall application speedup over the basline OpenMP KRIPKE version by 19.5%

Feature Extraction and Performance Prediction

- ComIL is a ***Com**mon **I**nstruction format for **L**earning*, capable of representing many modern ISAs (currently parsers exist for x84-64 and PTX)
- Special considerations for the construction of the graph and instructions are made for OpenMP and CUDA (specifically calls to parallel/device regions)



Performance Prediction with Graph-Based Program Features

- Split versions into three groups: training (10%), test (10%), and validation (80%)
- Extract projected eigenvectors from the adjacency matrix of each graph and create a projected view of the features, yielding a graph spectral feature (GSF) for each version
- Feed the training set through a deep neural network with a regression outcome on the execution time of a given instance. Use the test set to minimize overfitting.
- With the generated model, evaluate the remaining 80% of instances and compare the predicted performance to the actual performance. KRIPKE has mean accuracy of 93.3%

Conclusion and Future Work

- Used the RAJA performance portability layer to explore a large optimization space efficiently within the KRIPKE Sn transport proxy application
- The best known execution time of KRIPKE improves by 19.5%.
- Graph-based program feature extraction allows for an architecture-portable way of charactering programs for multiple architectures
- Execution time prediction from 20% of random kernel samples (10% train, 10% test) for KRIPKE achieves mean accuracy of 93.3%

Future Work

- Expand results to include GPU execution policies (NVIDIA Kepler/Pascal) and nested parallelism with many-core (Intel Knight's Landing) architectures

Acknowledgments and Resources

[1] A. J. Kunen, T. S. Bailey, P. N. Brown, *KRIPKE - A Massively Parallel Transport Mini-App*, American Nuclear Society M&C, 2015 [https://codesign.llnl.gov/kripke.php]

[2] R. D. Hornung and J. A. Keasler, *The RAJA Portability Layer: Overview and Status, Tech Report*, LLNL-TR-661403, Sep. 2014. [https://github.com/llnl/RAJA]

[3] W. Killian, A. J. Kunen, I. Karlin, J. Cavazos, *Discovering Optimal Execution Policies in KRIPKE using RAJA*, ACM SRC SuperComputing 2016. [http://www.udel.edu/003786]