Neural Network Assisted Tile Size Selection

Mohammed Rahman, Louis-Noël Pouchet and P. Sadayappan

Dept. of Computer Science and Engineering
Ohio State University

June 22, 2010

iWAPT 2010 Workshop
Berkeley, USA
Overview

Situation:
- New advances in parametric tiling → more user code to be tuned
- The problem of tile size selection is complex and unsolved!

Our approach:
- Use machine learning to create a performance predictor of tile size performance, for a specific program
- Rely on the distribution shape to extract promising subspaces for empirical search
- Outcome: < 2% of the space traversed → 90+% of maximal speedup achieved
Tiling

- Tiling partition the computation into blocks
- Note we consider only rectangular tiling here
- For tiling to be legal, such a partitioning must be legal
Parametric Tiling

Automatic parametric tiling [ICS’09, CGO’10]:

- Produce code where the tile dimensions are parameters
- Seamlessly find/apply all required transformation to make the code tilable
- Actual tile sizes are given at run-time
- Very useful for tile size selection (no need to recompile)
- Recent progresses have generalized the approach:
  - Operates on arbitrary affine-control loops (imperfectly nested)
  - Produce good quality code
  - Even expose pipeline-parallelism if needed
  - Software (from OSU): Pluto, PrimeTile/DynTile/PTile
Tile Size Selection

Problem: how to select the tile size to have the best performance?

- data reuse *within the execution of a tile*;
- data reuse *between tiles*;
- the layout in memory of the data used in a tile;
- the relative penalty of misses at each level of the hierarchy, which is machine-dependent.
- the cache replacement policy;
- the interaction with other units, such as prefetching;
- the interaction with vectorization, to enable a profitable steady-state for the vectorized loop(s);
- ...

Ohio State
Performance Distribution

Performance distribution of fdtd-2d and syr2k

- Search space: 10648 possible tile sizes
  - \{1, 2, 4, 6, 8, 10, 12, 16, 30, 32, 40, 48, 64, 100, 128, 150, 200, 256, 300, 400, 500, 600\}
- Machine: Core i7 (1 thread)
- 2 "standard" distribution shapes
**Ojectives**

**Correlate execution time with tile sizes**

- (Static) performance models do exist...
- ... but fail to capture the interplay between all hardware components
- Usually better suited for well-known problems (eg, uniform reuse + square tiles)
- Another view: pruning the space of poor-performing tile sizes

**Our approach:**

- Build a neural network to model the performance distribution
- Focus directly on the execution time
- ANN dedicated to a specific program + dataset size
Neural Network

Layout:

- Fully connected, multi-layer perceptron (MLP)
- Input layer: the tile sizes \( T_i, T_j, T_k \)
- Output layer: predicted execution time
- One hidden layer consisting of 30 hidden neurons
- Use Stuttgart Neural Network Simulator library

Training:

- Select 5% (530 tuples) from the search space of 10648
- Run the program on the machine using the tile size specified by the tuples
- Train with resilient back-propagation (rprop), using the actual execution time for a tuple
- Standard 10% cross-validation procedure
Performance Prediction [1/2]

fdtd-2d: Predicted versus Actual Performance

dsysr2k : Predicted versus Actual Performance
Performance Prediction [2/2]

**Lu: Predicted versus Actual Performance**

- **Execution Time in Seconds**
- **Tile sizes (Ti:Tj:Tk)**
  - ExTime (Actual)
  - ExTime (Predicted)

**Dgemm: Predicted versus Actual Performance**

- **Execution Time in Seconds**
- **Tile Sizes (Ti:Tj:Tk)**
  - ExTime (Actual)
  - ExTime (Predicted)
Discussions

- for trmm, lu, 2d-jacobi, syr2k and doitgen, predict more than 90% of our search space with less than 10% deviation for the actual execution time
- In total, can predict 80% and more with less than 10% deviation
- Usually smaller deviation for the best tile sizes

→ These ANN are able to model the performance distribution

Openings:
- Program classifier w.r.t. performance distribution
- Training: do not "fit" that much the training points?
Selecting the Best Tile Size

The performance distribution can drive the empirical search to focus on promising subspaces

Tile size selection:

- Random approach has a huge variability on some distribution shapes
- Exhaustive search is likely not needed
- Need for an intermediate solution
  - Low number of empirical runs
  - Good convergence, good variability
  - General enough to work on arbitrary user codes
Overview of the Algorithm

1. Generate a parametrically tiled code

2. Randomly select $x\%$ of the tile size space, and run them on the machine

3. Train an ANN using this data

4. Use the ANN to predict performance of the entire space

5. Collect $y$ tile sizes that are predicted best and not already ran

6. Run the $y$ tile sizes on the machine, output the best found
Experimental Setup

- Studied various kernels (perfectly/imperfectly nested, BLAS & stencils)
- Only focused on single-threaded execution, on an Intel Core i7

- Comparison: simple random search (R), ANN search (ANN)
- Repeat each experiment 100 times, for various sampling rate
## Experimental Results ($y = 50$)

<table>
<thead>
<tr>
<th></th>
<th>doitgen</th>
<th>gemm</th>
<th>syr2k</th>
<th>lu</th>
<th>2d-jacobi</th>
<th>fdtd-2d</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1%</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-best</td>
<td>100%</td>
<td>99.86%</td>
<td>98.15%</td>
<td>99.89%</td>
<td>99.91%</td>
<td>97.75%</td>
</tr>
<tr>
<td>R-average</td>
<td>98.71%</td>
<td>96.29%</td>
<td>94.80%</td>
<td>92.19%</td>
<td>94.10%</td>
<td>84.15%</td>
</tr>
<tr>
<td><strong>R-worst</strong></td>
<td><strong>95.35%</strong></td>
<td><strong>69.64%</strong></td>
<td><strong>89.81%</strong></td>
<td><strong>40.63%</strong></td>
<td><strong>17.69%</strong></td>
<td><strong>31.02%</strong></td>
</tr>
<tr>
<td>ANN-best</td>
<td>100%</td>
<td>99.86%</td>
<td>100%</td>
<td>92.62%</td>
<td>98.51%</td>
<td>100%</td>
</tr>
<tr>
<td>ANN-average</td>
<td>98.89%</td>
<td>96.35%</td>
<td>96.01%</td>
<td>92.62%</td>
<td>98.51%</td>
<td>100%</td>
</tr>
<tr>
<td>ANN-worst</td>
<td><strong>97.26%</strong></td>
<td><strong>82.93%</strong></td>
<td><strong>89.79%</strong></td>
<td><strong>79.68%</strong></td>
<td><strong>94.23%</strong></td>
<td><strong>66.53%</strong></td>
</tr>
<tr>
<td><strong>2%</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-best</td>
<td>99.97%</td>
<td>99.86%</td>
<td>98.71%</td>
<td>99.89%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>R-average</td>
<td>98.71%</td>
<td>96.42%</td>
<td>94.80%</td>
<td>92.87%</td>
<td>97.60%</td>
<td>84.10%</td>
</tr>
<tr>
<td><strong>R-worst</strong></td>
<td><strong>86.49%</strong></td>
<td><strong>67.89%</strong></td>
<td><strong>88.20%</strong></td>
<td><strong>45.29%</strong></td>
<td><strong>55.98%</strong></td>
<td><strong>27.30%</strong></td>
</tr>
<tr>
<td>ANN-best</td>
<td>100%</td>
<td>99.86%</td>
<td>100%</td>
<td>95.34%</td>
<td>98.55%</td>
<td>100%</td>
</tr>
<tr>
<td>ANN-average</td>
<td>98.89%</td>
<td>96.76%</td>
<td>96.69%</td>
<td>95.34%</td>
<td>98.55%</td>
<td>100%</td>
</tr>
<tr>
<td>ANN-worst</td>
<td><strong>97.26%</strong></td>
<td><strong>89.83%</strong></td>
<td><strong>89.65%</strong></td>
<td><strong>85.80%</strong></td>
<td><strong>94.17%</strong></td>
<td><strong>60.65%</strong></td>
</tr>
<tr>
<td><strong>3%</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-best</td>
<td>99.97%</td>
<td>99.86%</td>
<td>98.71%</td>
<td>99.89%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>R-average</td>
<td>98.77%</td>
<td>96.47%</td>
<td>94.80%</td>
<td>94.27%</td>
<td>98.39%</td>
<td>85.47%</td>
</tr>
<tr>
<td><strong>R-worst</strong></td>
<td><strong>94.89%</strong></td>
<td><strong>63.58%</strong></td>
<td><strong>87.99%</strong></td>
<td><strong>61.24%</strong></td>
<td><strong>84.54%</strong></td>
<td><strong>47.99%</strong></td>
</tr>
<tr>
<td>ANN-best</td>
<td>99.97%</td>
<td>99.86%</td>
<td>100%</td>
<td>95.34%</td>
<td>98.74%</td>
<td>100%</td>
</tr>
<tr>
<td>ANN-average</td>
<td>98.93%</td>
<td>97.14%</td>
<td>97.17%</td>
<td>95.34%</td>
<td>98.74%</td>
<td>100%</td>
</tr>
<tr>
<td>ANN-worst</td>
<td><strong>97.64%</strong></td>
<td><strong>91.01%</strong></td>
<td><strong>92.27%</strong></td>
<td><strong>85.80%</strong></td>
<td><strong>94.50%</strong></td>
<td><strong>63.34%</strong></td>
</tr>
<tr>
<td><strong>4%</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-best</td>
<td>99.97%</td>
<td>99.86%</td>
<td>98.71%</td>
<td>99.89%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>R-average</td>
<td>98.80%</td>
<td>96.65%</td>
<td>94.93%</td>
<td>92.19%</td>
<td>98.41%</td>
<td>85.55%</td>
</tr>
<tr>
<td><strong>R-worst</strong></td>
<td><strong>96.86%</strong></td>
<td><strong>69.73%</strong></td>
<td><strong>88.57%</strong></td>
<td><strong>52.03%</strong></td>
<td><strong>82.47%</strong></td>
<td><strong>43.74%</strong></td>
</tr>
<tr>
<td>ANN-best</td>
<td>100%</td>
<td>99.86%</td>
<td>100%</td>
<td>95.79%</td>
<td>98.90%</td>
<td>100%</td>
</tr>
<tr>
<td>ANN-average</td>
<td>98.99%</td>
<td>97.67%</td>
<td>97.20%</td>
<td>95.79%</td>
<td>98.90%</td>
<td>100%</td>
</tr>
<tr>
<td>ANN-worst</td>
<td><strong>98.28%</strong></td>
<td><strong>93.65%</strong></td>
<td><strong>92.66%</strong></td>
<td><strong>85.80%</strong></td>
<td><strong>94.50%</strong></td>
<td><strong>79.26%</strong></td>
</tr>
</tbody>
</table>
Some Related Work

Epshteyn et al. [LCPC’05]:
- Search-oriented contribution
- Uses regression curves to approximate the performance distribution
- Uses active learning to select good candidates for empirical evaluation
- Good results for BLAS kernels

Yuki et al. [CGO’10]:
- Aims at selecting/combining between different static models
- Uses program features to characterize accesses, train ANN
- Results demonstrated for matrix-like kernels
Conclusions and Future Work

ANN is a candidate approach to connect tile sizes with performance

- Good prediction quality
- Deviation usually smaller for the good points
- Combined search heuristic proposed:
  - Strong variability improvement over naive random approach
  - 90+% efficiency using < 2% of the space, likely can be improved further

Future work:

- **Generalization!**
  - Categorize benchmarks reg. the performance distribution shape
  - Dataset size
- Do not try to fit the random samples during training
  - Reduce the training time
  - problem: ANN configuration
This work was funded in part by the U.S. National Science Foundation through award 0926688 and the Defense Advanced Research Projects Agency through AFRL Contract FA8650-09-C-7915. The opinions and findings in this document do not necessarily reflect the views of either the United States Government or the Ohio State University.