Neural Network Assisted Tile Size Selection

Mohammed Rahman, Louis-Noël Pouchet, and P. Sadayappan

The Ohio State University
{rahmanm,pouchet,saday}@cse.ohio-state.edu

Abstract. Data locality optimization plays a significant role in reducing the execution time of many loop-intensive kernels. Loop tiling at various levels is often used to effectively exploit data locality in deep memory hierarchies. The recent development of frameworks for parametric loop tiling of user code has lead to a widening of the range of applications that could benefit from auto-tuning of tile sizes.
Current model-driven approaches suffer from limitations, such as the inability to accurately model the complex interplay between multiple hardware components that affect performance. Auto-tuning libraries such as ATLAS rely on extensive empirical search for tile size optimization, which has been shown to be very effective. However, the effectiveness of such approaches for arbitrary parametrically tiled user code has not been demonstrated.
We consider the problem of selecting the best tile sizes for arbitrary user-defined programs, by sampling in the full space of tile sizes. We have developed a technique to build a performance predictor associated with a specific program. Our approach uses statistical machine learning to train an artificial neural network (ANN) to predict the performance distribution of execution time for scientific kernels. We show how this search strategy significantly improves over the variability of random search. Our observations and results on various kernels also show promise for the use of ANNs in predicting the runtime behavior for variations of tiling configurations.

1 Introduction

Tiling [31, 32] is recognized as a critical transformation for achieving high performance for nested loop computations. It is well known that the choice of suitable tile sizes can have a significant impact on performance. With small tile sizes, the referenced data can all fit in cache, but insufficient reuse may be exploited within the iterations of a tile. With very large tile sizes, the data referenced within a tile may not fit in cache, resulting in capacity misses. The choice of an optimal tile size is a difficult challenge. There is a complex interplay between the capacities and latencies of caches and TLBs (Translation Lookaside Buffer) at different levels of the memory hierarchy, as well other factors such as hardware prefetching and the effectiveness of use of SIMD instruction sets.

Although several previous studies have considered analytical approaches to tile size optimization [4, 7, 21, 22], none have yet been demonstrated to be sufficiently generic and robust to be used in practice for selecting the best tile size for arbitrary programs, ranging from sequences of BLAS operations to stencil codes.

The current state of practice with tile size optimization is empirical tuning [28, 30, 2, 24, 25]. The highly successful ATLAS (Automatically Tuned Linear Algebra Software) system uses extensive empirical tuning at library installation time to find the best tile sizes for different problem sizes on that machine [28]. While the effectiveness of the ATLAS tuning strategy has been demonstrated on well-studied elementary BLAS operations, it remains to be proven that such an approach can provide similar performance quality when operating on arbitrary user codes that are not known a priori.

Recent advances have resulted in the development of software to automatically generate parametric tiled code [8, 18, 1, 9, 26, 19, 10, 14], where the tile sizes are denoted by symbolic variables that can be set at runtime. A tile size optimizer in such a general context must be able to handle arbitrary tiled codes and it is not possible to use a priori knowledge about the tiled code to design empirical search heuristics. Performing an extensive empirical search over the combinations of problem sizes and tile sizes usually has a prohibitive time cost. On the other hand, it seems unrealistic to be able to construct a purely analytical model for tile size optimization that is general, robust, and accurately models the numerous machine parameters and complexities that impact the optimal choice of tile sizes.

In this paper, we explore the use of machine learning to automatically build a sufficiently accurate model based on a small number of empirical executions of parametrically tiled code, which can be useful in predicting effective tile sizes. Rather than attempt to model a number of factors such as cache and TLB miss counts, we simply use total execution time as the single metric that is modeled as a function of problem size parameters and tile sizes. This model is then used to isolate a small number of predicted good tile sizes, that are the final candidates for empirical evaluation. We show that this two-step search technique outperforms a random search using the same total number of samples. More importantly, we show that our approach significantly improves the worst-case scenario when compared to random search, suggesting that our sampling approach may be a viable alternative to exhaustive search.

2 Problem Statement

Loop tiling is a crucial transformation for data locality improvement, as it attempts to speed up the computation time by partitioning the computation into blocks, such that the data elements required by a block fit in local memory (e.g., local data cache) [31]. However, a critical problem to address when performing tiling is the computation of the tile sizes, as it can dramatically impact performance. Small tile sizes will under-utilize the resources (e.g., data cache) and lead to sub-optimal performance, while tile sizes that are too large will lead to increased data cache misses and low performance. Most previous work has attempted to use analytical models to estimate the performance as a function of tile sizes. These models typically attempt to minimize a composite cost function using an idealized machine execution model [11, 7, 22, 20].

Designing such a model is a very complex task, and to accurately predict the performance requires that a number of aspects be taken into account:

- 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 at prefetching;
- the interaction with vectorization, to enable a profitable steady-state for the vectorized loop(s);

The current state-of-the-art in tile size selection fails to encompass in a single model all the possible interactions, and this usually results in inaccuracy of predictions of optimal tile sizes.

Empirical search has become a valuable alternative, in particular for the automatic and portable tuning of high-performance libraries such as ATLAS [28]. When generalizing to arbitrary programs to be tuned, one may proceed by successively testing, on the target machine, numerous possible tile sizes to identify the best choice. This tuning process is time-consuming, as it is very hard to predict the best tile sizes without traversing a significant portion of the search space. We illustrate this with two representative performance distributions with respect to tile sizes. In Figure 1, we show the performance variation for 10648 tile size combnations for two representative benchmarks: a 2D stencil computation (fdtd-2d) and a standard BLAS3 routine (dsyr2k), considering a 3D single-level tiling.

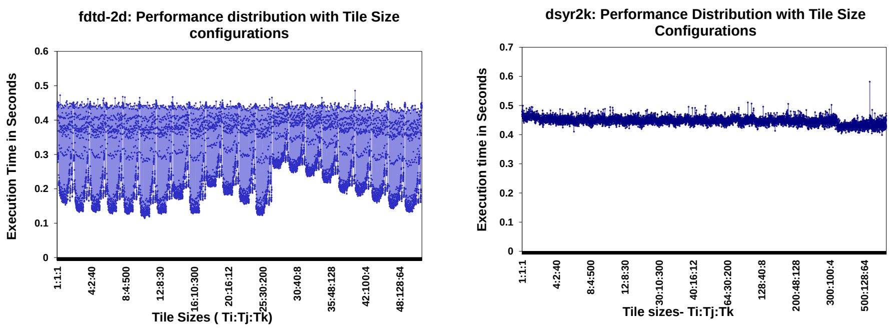

Fig. 1. Performance distribution of fdtd-2d and syr2k

For fdtd-2d we observe that the performance distribution is not uniform, and the space is plagued with poorly performing tile combinations. The ratio of the number of high-performance tile combinations (which achieve 95% or more of the maximal performance) is limited to a small fraction of the search space, below 1%. For such problems, it is expected that a purely random approach for tile sizes selection requires sampling above the ratio of good points to converge towards a good solution, because again of the non-uniformity of the distribution. Worse, randomly sampling a constant number of points can lead to the selection of very sub-optimal choices, as shown in Section 4.1. On the other hand, for dsyr2k we observe much less performance variation between the possible tile sizes. For such loops, a random approach is likely to quickly succeed in finding a good point in the search space, due to the abundance of such points.

The wider availability of parametrically tiled codes implies the need to tune tile sizes on a wide variety of codes, ranging from standard BLAS operations to stencil codes such as fdtd-2d. Extensive empirical search is very expensive, and an alternative approach that achieves the effectiveness of iterative tuning with a reduced search time is needed.

One major limitation of an empirical search technique based on random sampling is the challenge of providing convergence bounds. Different performance distribution shapes will require different sampling rates to statistically assure converge towards a good solution. In addition, there is no knowledge extracted from this random sampling: when facing a different problem size, repetition of the search is required.

We propose to address these problems by using a machine learning approach for the prediction of the *performance distribution* of tile sizes, for a given benchmark. We decouple the search of the best tile size into two steps. First, we randomly evaluate a fraction of the search space, using empirically measured execution time for those tile sizes to *train an artificial neural network* (ANN), for a given program and dataset size. Next, we use an extensive search based on the ANN model to determine a set of *predicted* good points, around the local minima of the performance distribution. These selected points are then empirically evaluated and the best one is chosen. We show through experimentation with seven benchmarks, that the approach is very effective in identifying tile sizes that result in execution time within 10% of the global minimum execution time for the entire search space.

3 Performance Prediction

We now present our modeling approach for performance prediction. We first detail our experimental setting in Section 3.1. We present the neural network layout selection in Section 3.2, and detail the training process in Section 3.3. We report extensive qualitative evaluation of the performance prediction model in Section 3.4.

3.1 Experimental Protocol

Testing platform Our experimental setup focuses on the tile size selection problem for the Nehalem architecture. Our test platform runs Linux, and is powered by an Intel Core i7 860 running at 2.8 GHz, with data cache sizes of 32 kB / core for Level 1, 256kB / core for Level 2, 8MB for Level 3 and a Level 1 TLB with 64 entries for small pages (4K). Note that the technique for performance prediction we present is not biased by any architectural parameters. So one can seamlessly apply the same approach on a different architecture.

Benchmarks We experimented with 7 benchmarks. doitgen computes the reduction sum of a three-dimensional matrix multiplied by a two-dimensional matrix, dgemm, dtrmm, dsyr2k are three standard BLAS3 computations, and lu performs a LU matrix decomposition. 2d-jacobi and fdtd-2d are two stencil computations that operates on 2D arrays. All these benchmarks use double-precision floating point arithmetic.

For each benchmark, a sequence of affine loop transformations was applied to make the loop nest(s) fully permutable, and hence tilable [3, 17]. A unique parametric tiled code was then generated, using `PrimeTile 0.3.0` [8, 18] on the transformed loop nest. Note that for stencils, it is necessary to skew the original iteration domains to enable tiling for all loops; this skewed and tiled version is generated seamlessly in the affine framework and is an example of the recent advances in automatic parametric tiling.

We consider only single-level tiling, and for all benchmarks we tile the 3 outer-most loops. We note the tile sizes for these loop levels T_i, T_j and T_k.

Compiler For all tested versions, including the original code, we used the same compiler together with the same optimization flags. Specifically, we used GCC 4.3.2 with option `-O3`, resulting in single-threaded programs, and the vectorization task is left entirely to GCC.

Dataset We have constrained the problem such that for each benchmark, we build and train a specific network. We do not vary the dataset size during the training process, it has been computed to exceed L2 cache size, and all arrays have been sized to 600×600.

An important observation is that we experimented with benchmarks that have statically predictable control flow, also known as SCoP [6]. On such programs, the execution does not depend on the value of the dataset, hence tuning for different dataset values is not needed. For our experiments we used a dense dataset of double precision floating point numbers.

Search space of tile sizes We perform a sampling of the space of all tile sizes, by evaluating along each dimension 22 possible sizes from the set $\{1, 2, 4, 6, 8, 10, 12, 16, 30, 32, 40, 48, 64, 100, 128, 150, 200, 256, 300, 400, 500, 600\}$. Since we consider a tiling along 3 dimensions, this leads to a search space of $22^3 = 10648$ possible tile sizes.

3.2 Neural Network Configuration

We use a fully connected, multi-layer perceptron (MLP), feed forward neural network. Our network consists of an input layer with three input parameters (the tile sizes: T_i, T_j, T_k), an output layer with one output parameter (predicted execution time), and one hidden layer consisting of 30 hidden neurons. Thus, our network has 34 units with 120 connections (edges). Input values are presented to the input layer and the predicted value is taken out of the output layer. We use a logistic activation function for the hidden layer and a linear activation function for the output in our fully connected feed-forward neural network. Each tile size configuration is presented as a three dimensional input vector to our ANN. The functional relationship between input and output variables are defined by the weights associated with the edges of the networks. These edge weights corresponding to each connection in the network are set during the training phase of building the models.

3.3 Training the Neural Network

Training the neural network is an iterative process that involves learning the edge weights from a sample of tuples from the search space consisting of input /output parameters. We used rprop (resilient back-propagation) as the training algorithm to train the edge weights of network with a learning rate of 0.001. Finally, we used SNNS (Stuttgart Neural Network Simulator) [34] to train and develop our neural network model described here.

In the experiments presented in this section, a random sample of approximately 5% i.e. 530 tuples from our search space of 10648 tuples are randomly selected for training and validation of the neural network. For each of these tuples, the actual performance is collected by running the program on the machine using the tile size specified by the tuple. Out of these 530 tuples, approximately 10% (50 tuples) are separated for validation and early stopping while the remaining 480 tuples are used for training the neural network. At every certain number of epochs of training, we fed the network to predict the output corresponding to the tuples in the validation set. We keep on training the network as we see the improvement in prediction on the validation set measured in sum square errors. When we see no improvement on validation set for certain number of counts, we stop training the network to prevent it from memorizing rather than learning the patterns.

3.4 Qualitative Analysis

We report in Figure 2 to Figure 8 the evaluation of our performance prediction model. For each benchmark, we report in the figure on the left hand side the actual performance and the predicted performance (that is, when computed through our neural network that was trained by randomly sampling 5% of the search space) of all tile sizes in our search space. These are sorted from the best to the worst performance: more variations on the right side of the graph indicates a higher error rate for tile sizes which have the lowest performance. On the figure on right hand side, we plot the overall performance distribution, sorted with respect to the tile sizes. The more chaotic the distribution, the more complex is the function the ANN must learn.

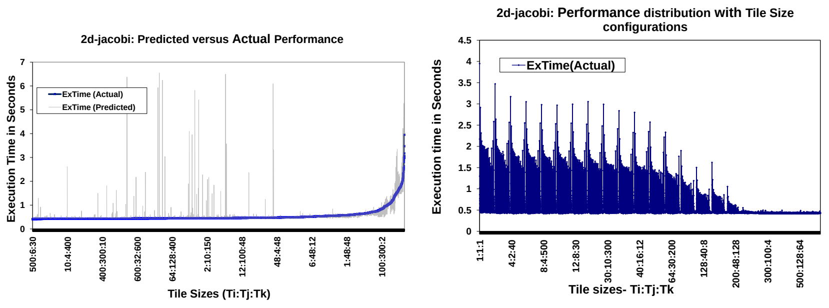

Fig. 2. 2d-jacobi Evaluation

We observe that our networks are able to accurately model the execution time in particular regarding the best performing versions (that is, points on the left size of the x axis). For the case of complex distributions such as lu, fdtd-2d or trmm the ANN is able to learn a function that correctly interpolates all local minima, despite a lower quality of prediction for tile sizes that perform among the worse in the space.

We also observe that for five of the benchmarks — trmm, lu, 2d-jacobi, syr2k and doitgen — we could predict more than 90% of our search space with less than 10% deviation for the actual execution time, and for all benchmarks we can predict 80% and more with less than 10% deviation. Some of the configurations leading to higher than 10% deviations are extreme cases of configurations like tile size being extremely small

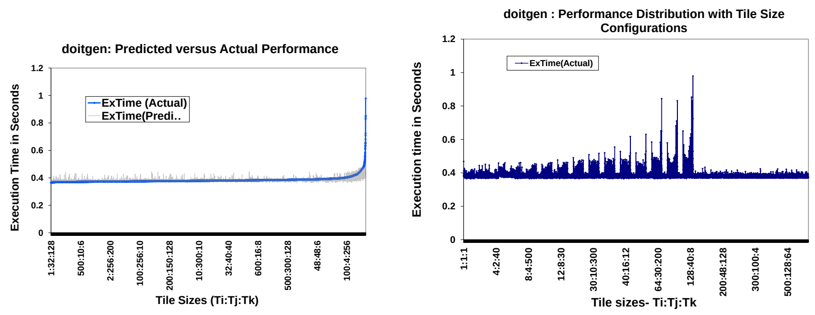

Fig. 3. doitgen Evaluation

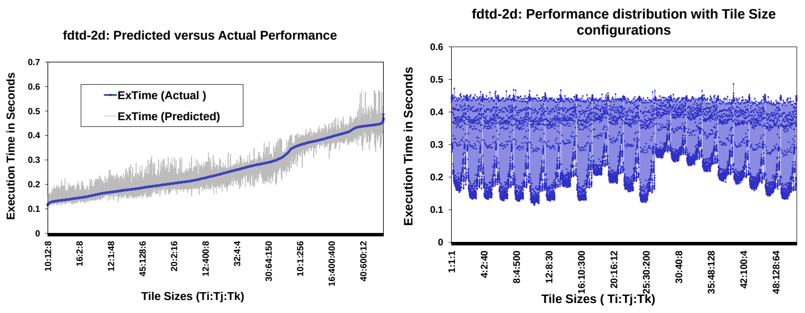

Fig. 4. fdtd-2d Evaluation

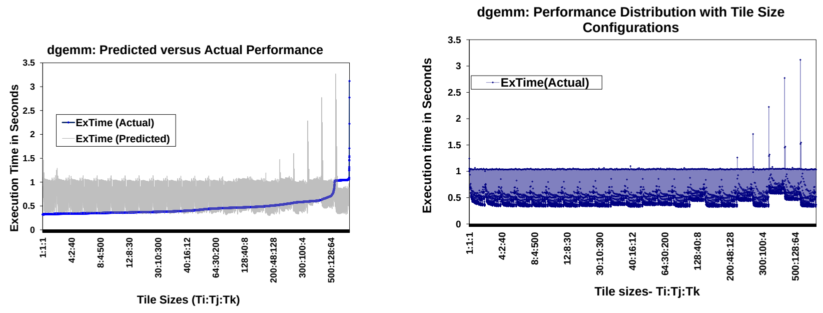

Fig. 5. dgemm Evaluation

(e.g. 1, equivalent to not tile the dimension). Such extreme tile sizes lead to statistically outliers execution time making it difficult for the ANN to learn or to predict.

An important observation is that we are interested in learning the *performance distribution*, and in particular to accurately predict which tile size will have the best execution

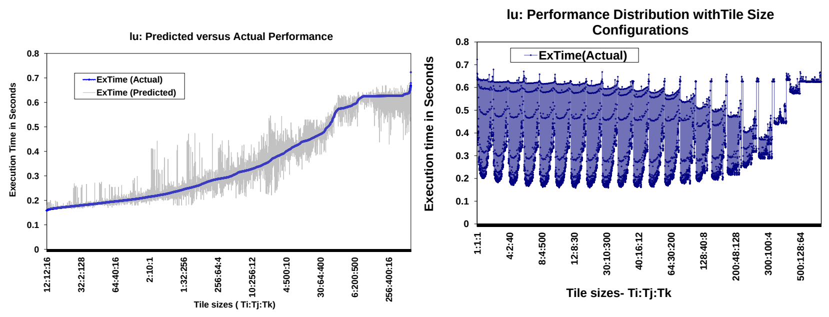

Fig. 6. lu Evaluation

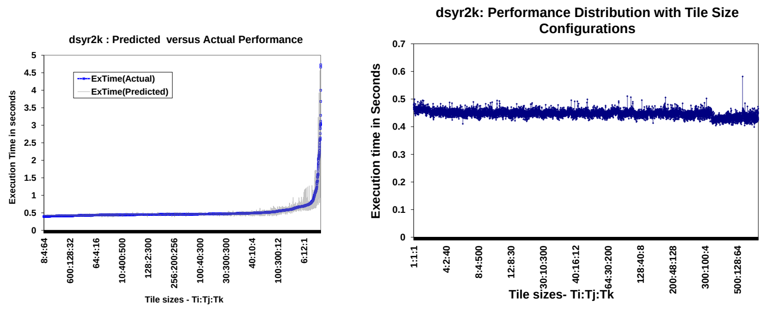

Fig. 7. dsyr2k Evaluation

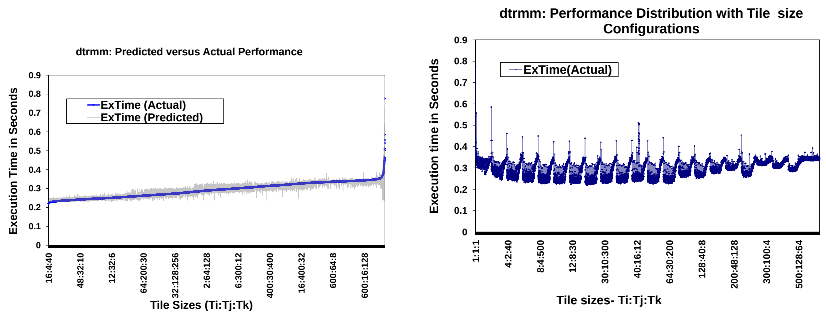

Fig. 8. dtrmm Evaluation

time. Although our results show in some cases a deviation rate which may seem unsatisfactory, care must be taken when interpreting these numbers. First, we are interested in finding the best tile size, so even if their execution time is predicted with a significant deviation, what really matters is only that they are predicted as the best size by the model.

Similarly, if the deviation occurs for average to low performing configurations, this does not impact the selection of the *best* tile size. We show in the following section that this overall error rate does not harm the efficiency of our approach to find the best tile size.

4 Searching for the Best Tile Size

We now present our strategy for selecting the best performing tile sizes. Our approach is aimed to substitute for a simple random search algorithm for the case of single-level tiling.

4.1 Neural Network Assisted Selection of the Best Tile Sizes

Our neural-network assisted search technique computes on the fly the performance distribution based on a first random sampling phase, for which each randomly selected candidate tile size in the search space is empirically evaluated. Then, instead of randomly evaluating more points, the performance distribution is used to isolate a fraction of the space where the best tile sizes are predicted to be good: these points are the one which are empirically evaluated. Technically, our two-stage approach for tile sizes selection is based on our observations of the performance distribution. We leverage the emergence of local minima in the performance distribution, and collect a bucket of the 50 predicted best tile sizes. We then empirically evaluate the performance of all tile sizes in this set, keeping only the best performing one. This is detailed in the algorithm in Figure 9.

```
TileSizeSelection: Select the best tile size
Input:
  program: a parametrically tiled program
  searchSpace: set of all tile sizes in the search space
  samplingRate: sampling rate for random search
  bucketSize: number of candidates selected from the ANN
              for the second empirical evaluation
Output:
  best: a tuple of best tile size

ActualTime ← emptyDatabase
Tuples ← randomlySelectTuplesInSearchSpace(samplingRate, searchSpace)
foreach t ∈ Tuples do
    executionTime ← executeProgramWithTileSize(program, t)
    addToDatabase(ActualTime, (t, executionTime))
end do
𝒜 ← buildAndTrainANNForPerfPrediction(ActualTime)
PredictedTime ← emptyDatabase
foreach t ∈ searchSpace do
    predictedExecTime ← getPredictedTimeFromANN(𝒜, t)
    addToDatabase(PredictedTime, (t, predictedExecTime))
end do
Bucket ← selectTuplesWithBestTimeInDatabase(PredictedTime, bucketSize)
foreach t ∈ Bucket do
    executionTime ← executeProgramWithTileSize(program, t)
    addToDatabase(ActualTime, (t, executionTime))
end do
best ← selectTuplesWithBestTimeInDatabase(ActualTime, 1)
return best
```

Fig. 9. Tile size selection algorithm

Procedure `randomlySelectTuplesInSearchSpace` selects randomly a fraction *samplingRate* (eg, $x\%$) of the search space of possible tile sizes. Procedure `getPredictedTimeFromANN` evaluates with the ANN a given tile size, to obtain the predicted execution time. Procedure `selectTuplesWithBestTimeInDatabase` selects the y best tile sizes sorted according to their execution time.

4.2 Experimental Results

We have conducted extensive experiments to validate the benefit of this decoupled approach over a standard random search. We use the Efficiency indicator to measure the performance of a point. The efficiency of a candidate tile sizes is defined as:

$$\text{efficiency} = \frac{\text{space optimal actual execution time}}{\text{candidate actual execution time}}$$

In other words, an efficiency of 100% means the candidate is the best performing tile sizes in the entire search space.

We report in Table 1 the efficiency of the best found tile sizes, for Random versus ANN (our decoupled search), for different random sampling rate (*samplingRate* goes from 1% to 4%). We have set $bucketSize = 50$. Because we use a random search strategy, there can be a significant variation between the performance of the best tile sizes found when doing two occurrences of a random draw of $x\%$ of the search space. To highlight this variation, we have repeated each experiment 100 times, and report the worst, average over the 100 runs, and best efficiency of the best tile sizes which was found by the two methods.

The execution time for the full process of finding the best tile size is totally dominated by the execution time of the candidate tile sizes on the machine. The complete process (empirical evaluation of candidates, training of the network and gathering the predicted performance) never exceeded 16 minutes on our test suite, for the highest sampling rate for 2d-jacobi. In average, using a sampling rate of 2% our process completed in less than 5 minutes for all benchmarks.

The random points found by the Random heuristic are those used to train the ANN. For a fair comparison, one should take into account the extra 50 points (about 0.5% of the space) that are evaluated with our decoupled approach: 1% sampling rate for the ANN leads to an empirical evaluation of a total 1.5% of the space, 2% sampling leads to evaluating 2.5% of the space, etc.

A key observation is that our approach systematically outperforms a pure random search, even when using a slightly higher sampling rate. For instance, evaluating 2% of randomly selected points in the space shows a lower efficiency than randomly evaluating 1% of the space combined with the evaluation of 50 additional points selected by the model – a total of 1.5% points are evaluated.

However, we believe the most significant advantage of our approach is not in pure performance improvement over a random search. It is in the efficiency of the tile size found in the worst case scenario. Because of the non-uniform distribution of the best tile size, a single random draw may totally fail to discover a good tile size. However, our decoupled approach is able to identify interesting regions based on the performance distribution that is computed on-the-fly, efficiently driving the search to a subspace of

		doitgen	gemm	trmm	syr2k	lu	2d-jacobi	fdtd-2d
1%	R-best	100%	99.86%	99.69%	98.15%	99.89%	99.91%	97.75%
	R-average	98.71%	96.29%	94.18%	94.80%	92.19%	94.10%	84.15%
	R-worst	**95.35%**	**69.64%**	**73.91%**	**89.81%**	**40.63%**	**17.69%**	**31.02%**
	ANN-best	100%	99.86%	100%	100%	100%	99.91%	100%
	ANN-average	98.89%	96.35%	97.89%	96.01%	92.62%	98.51%	84.50%
	ANN-worst	**97.26%**	**82.93%**	**92.07%**	**89.79%**	**79.68%**	**94.23%**	**66.53%**
2%	R-best	99.97%	99.86%	100%	98.71%	99.89%	100%	100%
	R-average	98.71%	96.42%	94.26%	94.80%	92.87%	97.60%	84.10%
	R-worst	**86.49%**	**67.89%**	**68.40%**	**88.20%**	**45.29%**	**55.98%**	**27.30%**
	ANN-best	100%	99.86%	100%	100%	100%	100%	100%
	ANN-average	98.89%	96.76%	98.08%	96.69%	95.34%	98.55%	88.61%
	ANN-worst	**97.26%**	**89.83%**	**84.93%**	**89.65%**	**85.80%**	**94.17%**	**60.65%**
3%	R-best	99.97%	99.86%	100%	98.71%	99.89%	100%	100%
	R-average	98.77%	96.47%	94.34%	94.80%	94.27%	98.39%	85.47%
	R-worst	**94.89%**	**63.58%**	**64.53%**	**87.99%**	**61.24%**	**84.54%**	**47.99%**
	ANN-best	99.97%	99.86%	100%	100%	100%	100%	100%
	ANN-average	98.93%	97.14%	98.24%	97.17%	95.34%	98.74%	91.45%
	ANN-worst	**97.64%**	**71.74%**	**84.93%**	**92.27%**	**85.80%**	**94.50%**	**63.34%**
4%	R-best	99.97%	99.86%	100%	98.71%	99.89%	100%	100%
	R-average	98.80%	96.65%	94.89%	94.93%	92.19%	98.41%	85.55%
	R-worst	**96.86%**	**69.73%**	**89.21%**	**88.57%**	**52.03%**	**82.47%**	**43.74%**
	ANN-best	100%	99.86%	100%	100%	100%	100%	100%
	ANN-average	98.99%	97.67%	98.46%	97.20%	95.79%	98.90%	93.55%
	ANN-worst	**98.28%**	**73.57%**	**96.19%**	**92.66%**	**85.80%**	**94.50%**	**79.26%**

Table 1. Efficiency w.r.t. space optimal tile sizes of the best point found using random (R) and Neural Network (ANN) search strategies, for different fraction of the space empirically evaluated (1% to 4%). Running each experiments 100 times and considering the **worst** one, the best tile size found by ANN is always significantly better than the one discovered by R alone.

good performing points. As an extreme example for 2d-jacobi a random sampling of 2% of the space may lead to discovering a best tile size which performs at 55% of the actual space optimal point, while our ANN approach has a worst case efficiency of 94% when using a sampling rate of only 1%: it leads to almost a 2× difference in execution time of the best found point between the two methods.

Finally, let us note that for the benchmarks which have a high density of good points, a simple random search performs well, as one could expect. We observe that by design, our ANN-based technique performs at least as well as a standard random search. However our technique can be used to quickly approximate the performance distribution for such cases: if the distribution is flat, one can safely stop the search.

4.3 Discussions and Future Work

The advantage of our technique is the ability to exhibit all local minima in the performance distribution. With such an approach, one can extract a set of candidates for empirical evaluation which have a higher probability of performing well.

It is possible to address the problem of defining a convergence bound for the empirical search using our technique. We show it is possible to feed a neural network during a random sampling of the space to characterize the performance distribution. As a result, one can quickly derive the shape of the performance distribution *as an additional information* that can be used to stop the tuning. This is typically profitable for simple distributions such as dsyr2k where the benefit of extensive empirical search is very limited. On the other hand, for more complex distributions such as fdtd-2d or lu a higher proportion of the space must be sampled to provide a statistical guarantee to discover the best tile sizes.

We are currently investigating several degrees of generalization of our technique. The first one is to generalize our ANN to be able to deal with other dataset sizes. We are currently conducting extensive experiments to validate the following hypothesis.

- Considering a problem size large enough to exceed L2 cache size, the best tile sizes for a larger dataset size is also part of the best tile sizes for the current one;
- there may be best tile sizes for the current dataset size which are not the best one for larger dataset size.

Considering this, by collecting local minima we can ensure that only those need to be evaluated for larger dataset sizes, dramatically reducing the search process for other dataset sizes. At the time of writing, we have observed this hypothesis holds true for the Nehalem Core i7 for the case of numerous benchmarks.

5 Related Work

Finding the optimal tile sizes for effective exploitation of data locality a key issue in achieving high performance with kernels involving loop nests on platforms with a deep multi-level memory hierarchy. This has been studied in the past through different approaches including search based techniques [29], analytical modeling of cache misses equations [7, 4, 21, 22], and through the use of meta-heuristics like genetic algorithm and simulated annealing [15]. However, these approaches have not been demonstrated to be general, robust and effective. Therefore empirical tuning based on extensive search is the approach most commonly used in practice.

In the absence of a strong analytical model, researchers in different disciplines have used regression and neural networks for creating predictive models. It has been widely used in the architecture and systems community as well. Ipek *et al.* [12] studied the impact of hardware parameters (e.g. cache size, memory latency etc.) on the performance characteristics of target platforms using an ANN model. Wang and O'Boyle [27] used a machine learning based approach to predict the optimal number of threads and scheduling policy to map an already parallelized program to a multi core processor.

Singh *et al.* [23] used a machine learning based approach for predicting the performance of parallel applications - SMG 2000, a semi coarsening multigrid solver, and HPL.

Their model could predict the performance within 2% to 7% of the actual application run time. Khan *et al.* [13] used predictive modeling for cross program design space exploration in multi-core systems. They proposed a concept of reaction based characterization. In reaction based characterization, a model trained on one application can not only predict the behavior against unseen parameters for the same application, but also predict behavior for an unseen application as well.

Regarding tile sizes selection, Li and Garzaran [16] proposed a classifier learning system for optimizing matrix multiplication. Their system determines the number of levels of tiling and tile sizes at each level depending on the target platforms. Yuki *et al.* proposed a framework for tile size optimization based on program features, built on training an ANN [33]. We address in the present paper a slightly different problem, focusing on developing an accurate performance model for variation of tile sizes for a *specific* benchmark, with emphasis on reducing the variance when randomly searching the space of tile sizes.

Epshteyn *et al.* addressed the problem of efficiently using feedback from an iterative search process to select the next candidate to evaluate [5]. They use regression curves to approximate the performance distribution, and active learning to select good candidates for empirical evaluation. We pursue a similar goal, using an ANN to estimate the performance distribution instead. However our approach allows the collection of points around all local minima in the performance distribution, and is not geared towards discovering only the global minimum. Our preliminary experiments show that collecting local minima may be an efficient pruning strategy when searching for the best tile size for large dataset sizes.

6 Conclusion

Loop tiling is a critical transformation for making effective use of the cache hierarchy on modern machines. Recent advances in the design of automatic frameworks for parametric loop tiling [3, 8], combined with the ever-increasing complexity of modern architectures, have made the tile size optimization problem a practically significant one.

Many approaches to tile sizes selection have been based on analytical models for performance [4, 7, 21, 22], but have not been demonstrated to be robust and effective over a range of benchmark kernels or applications. At the other end, auto-tuning libraries leverage extensive empirical search of numerous tile sizes for equally numerous problem sizes [28, 2, 24]. Extensive empirical search is not realistic for arbitrary user-defined kernels: as the benefit is near-optimal tile sizes, the downside is the compilation time the user have to pay to get this performance. Furthermore, random sampling is usually discarded for tile size selection because of the high difficulty to determine an good sampling rate that trades adequately the search time and the efficiency of the best found tile size.

We have proposed to address the problem of tuning the tile sizes by using a Neural Network approach for predicting the execution time of different tile sizes, for a given benchmark. Our method is based on learning the performance distribution of tile sizes, and can be successfully used to determine convergence bounds for random empirical search. We used this performance model to predict the best tile sizes for a benchmark, and we have proposed a strategy to fully substitute for a standard random search, based on a learning on-the-fly of the performance distribution to drive the search towards interesting sub-spaces. This technique achieves between 93% and 98% of the maximal possible performance, and dramatically limit the statistical variance of the random search.

Acknowledgements. 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.

References

1. M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *The International Symposium on Code Generation and Optimization (CGO)*, 2010.
2. J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC. In *Proc. ACM International Conference on Supercomputing*, pages 340–347, 1997.
3. U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2008.
4. S. Coleman and K. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *PLDI'95*, pages 279–290, 1995.
5. A. Epshteyn, M. Garzaran, G. Dejong, D. Padua, G. Ren, X. Li, K. Yotov, and K. Pingali. Analytic models and empirical search: A hybrid approach to code optimization. In *Proc. of the International Workshop on Languages and Compilers for Parallel Computing (LCPC'05)*, 2005.
6. P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Program.*, 21(5):389–420, 1992.
7. S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: an analytical representation of cache misses. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 317–324, New York, NY, USA, 1997. ACM Press.
8. A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *ACM International Conference on Supercomputing (ICS)*, 2009.
9. A. Hartono, M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Parametric tiled loop generation for effective parallel execution on multicore processors. In *IPDPS'10: Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing*, 2010.
10. HiTLoG: Hierarchical Tiled Loop Generator. `http://www.cs.colostate.edu/MMAlpha/tiling/`.
11. C.-h. Hsu and U. Kremer. A quantitative analysis of tile size selection algorithms. *J. Supercomput.*, 27(3):279–294, 2004.
12. E. Ipek, S. A. McKee, K. Singh, R. Caruana, B. R. d. Supinski, and M. Schulz. Efficient architectural design space exploration via predictive modeling. *ACM Trans. Archit. Code Optim.*, 4(4):1–34, 2008.
13. S. Khan, P. Xekalakis, J. Cavazos, and M. Cintra. Using predictive modeling for cross-program design space exploration in multicore systems. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 327–338, Washington, DC, USA, 2007. IEEE Computer Society.
14. D. Kim, L. Renganarayanan, M. Strout, and S. Rajopadhye. Multi-level tiling: 'm' for the price of one. In *SC*, 2007.
15. T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, F. Bodin, and H. A. G. Wijshoff. A feasibility study in iterative compilation. In *ISHPC'99: Proc. of the Second Intl. Symp. on High Performance Computing*, pages 121–132, London, UK, 1999.

16. X. Li and M. J. Garzaran. Optimizing matrix multiplication with a classifier learning system. 2008.
17. The Pluto automatic parallelizer. `sourceforge.net/projects/pluto-compiler`, 2010.
18. PrimeTile: A Parametric Multi-Level Tiler for Imperfect Loop Nests. `http://www.cse.ohio-state.edu/~hartonoa/primetile/`.
19. L. Renganarayana, D. Kim, S. Rajopadhye, and M. Strout. Parameterized tiled loops for free. In *PLDI'07*, pages 405–414, 2007.
20. L. Renganarayanan. *Scalable and Efficient Tools for Multi-level Tiling*. PhD thesis, Department of Computer Science, Colorado State University, 2008.
21. G. Rivera and C. Tseng. A comparison of compiler tiling algorithms. In *CC '99: Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, pages 168–182, London, UK, 1999. Springer-Verlag.
22. V. Sarkar and N. Megiddo. An analytical model for loop tiling and its solution. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2000)*, April 2000.
23. K. Singh, E. İpek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana. Predicting parallel application performance via machine learning approaches: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(17):2219–2235, 2007.
24. C. Tapus, I.-H. Chung, and J. K. Hollingsworth. Active harmony: towards automated performance tuning. In *SC*, pages 1–11, 2002.
25. A. Tiwari, C. Chen, J. Chame, M. Hall, and J. Hollingsworth. Scalable autotuning framework for compiler optimization. In *IPDPS '09*, May 2009.
26. TLoG: A Parametrized Tiled Loop Generator. `http://www.cs.colostate.edu/MMAlpha/tiling/`.
27. Z. Wang and M. F. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 75–84, New York, NY, USA, 2009. ACM.
28. R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the ACM/IEEE SC98 Conference*, pages 1–27. IEEE Computer Society, 1998.
29. R. C. Whaley and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005.
30. R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.
31. M. Wolfe. More iteration space tiling. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664, New York, NY, USA, 1989. ACM.
32. J. Xue. *Loop tiling for parallelism*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.
33. T. Yuki, L. Renganarayanan, S. Rajopadhye, C. Anderson, A. Eichenberger, and K. O'Brien. Automatic creation of tile size selection models. In *Symp. on Code Generation and Optimization (CGO'09)*, Apr. 2010. to appear.
34. A. Zell, N. Mache, R. Hbner, G. Mamier, M. Vogt, K. uwe Herrmann, M. Schmalzl, T. Sommer, A. Hatzigeorgiou, S. Dring, D. Posselt, and M. R. Martin. Snns - stuttgart neural network simulator, 1993.