Learning Models in Self-Optimizing Systems

Karan Singh Vincent Weaver
Cornell University
{karan,vince}@csl.cornell.edu

Abstract

Library generators like ATLAS generate high-performance BLAS by performing a global, empirical search over the space of parameter values. ATLAS generates programs based on the parameter values it finds, and evaluates the best one based on actual hardware performance. Yotov et al [8] replace the global search with an analytical model and get comparable performance. As such, the model-driven approach requires considerably less work to generate the BLAS. The tradeoff is that hand-built models require a deep understanding of both computer architectures and the algorithm involved, and can take years to conceive. We propose replacing the ATLAS search engine with a machine learning model. The advantages of this approach being: (1) unlike ATLAS, the machine learning model requires minimal domain knowledge, (2) for one of our approaches, we obtain a model based on the parameter space instead of an optimal string of parameter values similar to Yotov et al [8]. Our approach can be extended to other library generators (e.g., FFTW) as well as new problems, with minimal effort.

1. Introduction

Library generators like ATLAS [7] generate high-performance BLAS by performing a global, empirical search over the space of parameter values. ATLAS generates test programs that evaluate the performance of various linear algebra routines by varying a set on input parameters. In this paper we focus on the matrix-matrix multiply case, which has the parameters given in Table 1. ATLAS runs these tests on actual hardware, and builds a library around the best performing set of inputs.

Yotov et al [8] replace ATLAS's global search with an analytical model and obtain comparable performance. As such, the model-driven approach requires considerably less work to generate the BLAS. The tradeoff is that hand-built models require a deep understanding of both the computer architectures and the algorithms involved, and can take years to develop.

We propose replacing the ATLAS search engine with a machine learning model. We evaluate two different machine learning techniques *Artificial Neural Networks* and *Genetic Algorithms*. The advantages of our approach are: (1) unlike ATLAS, the machine learning model requires minimal domain knowledge, (2) for artificial neural networks, we get a model based on the parameter space instead of an optimal string of parameter values similar to Yotov et al [8]. Please note that unlike Yotov et al, this model is platform-specific.

Our approach can be extended to other library generators (e.g., FFTW) with minimal effort. The only domain knowledge needed is a parameterized space that needs to be explored for best performance, and some reasonable bounds in the case where unreasonable parameters choke the compiler.

This paper is organized as follows. In Section 2, we review some related work. In Sections 3, we present background, implementation, and results using the artificial neural network approach.

Name	Description
N_B	L1 data cache tile size
M_U, N_U	Register tile size
K_U	Unroll factor for k' loop
L_s	Latency for computation scheduling
FMA	1 if fused multiply-add available, 0 otherwise
F_F, I_F, N_F	Scheduling of loads

Table 1. ATLAS Optimization Parameters

Section 4 presents background, implementation, and results using genetic algorithms. Finally, we present our conclusions and future work in Section 5. The appendix presents configurations for the machines used and the actual performance numbers obtained using ATLAS CGw/S, artificial neural networks, and genetic algorithms.

2. Related Work

This work builds off of the research into ATLAS done by Yotov et al [8], where the search part of ATLAS is replaced by a handcrafted model.

Epshteyn et al [2] use machine learning in conjunction with ATLAS, using active sampling to try to reduce the search space by learning which points are good to sample. Our method is much more comprehensive and uses machine learning for the entire search process.

While genetic algorithms have been applied to many similar problems, no one has seriously attempted to apply them to the ATLAS search before. It had been suggested previously though, as can be seen in the ATLAS FAQ [6]. The author of ATLAS has a strong commitment to the current search mechanism, but he mentions genetic algorithms explicitly in the FAQ as a method suggested previously to him by outside contributers.

A last minute additional literature search has turned up a paper released just a week ago at UTK [10]. This work applies Genetic Algorithms to ATLAS, but claims the results for 4 architectures are not as good as ATLAS's search overall. Our work tests more architectures, and provides more details on what results were found.

3. Artificial Neural Networks

3.1 Background

Artificial Neural Networks (ANNs) are a class of machine learning models that can be used to model complex relationships between a set of inputs and outputs. They are a powerful method for performing non-linear regression and work well even with noisy data. A neural network consists of layers with sets of units in each layer. A unit takes an input and produces an output based on an activation function (e.g. linear, sigmoid etc). Each unit in a layer is connected to units in the next layer via a weight. When values are propagated through the network, each unit sees the sum of the products of all

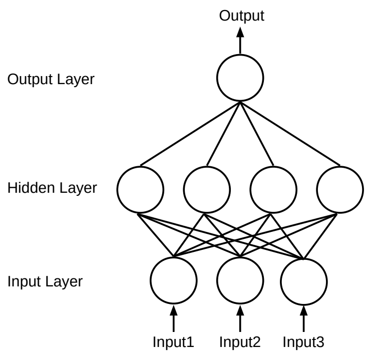

Figure 1. A feedforward neural network with one hidden layer [1]

the units and the weights connected to it. An ANN consists of an input layer, an output layer, and one or more hidden layers. Units receiving input values comprise the input layer, and units outputting the results form the output layer. Hidden layers lie between the input and output layers and help increase the representational power of the net. The network shown in Figure 1 represents a multilayer fully connected feedforward neural network, since every unit in a given layer receives values only from its immediately preceding layer.

Training an ANN consists of learning the edge weights in the net from a given training sample. Good learning implies that the net can generalize well and can make accurate predictions for inputs it has not been trained on. We use resilient backpropagation to train edge weights. Backpropagation, in general, uses gradient descent to minimize the error between the real and predicted values. When training, examples are repeatedly passed through the neural network and error is calculated. Then backpropagation takes a small step in the direction with minimal error using a small learning rate constant as the step size. Resilient backpropagation is adaptive and only propagates the sign of the error such that an evolving update rule is developed for each weight. Training with rprop tends to be faster than standard backpropagation.

In addition to rprop for training, there are some additional techniques that help enhance learning in ANNs.

3.1.1 Cross-validation

ANNs are prone to overfitting, in which case they tend to do extremely well on the train set but do not generalize well. As a result, they perform poorly on any new data they see. A common technique used to avoid overfitting is *early stopping*. We hold out part of the training set as an early stopping set. While training on the rest of the data, we check the performance on the early stopping set and save a copy if it has improved since the last run. If performance were to deteriorate, that would be an indicator of overfitting and a copy of the net would not be saved. In the end, we have copy of the net with the best performance (or least error) across all iterations. The drawback of this technique is that we end up losing some data we could have used for training an even better model. *Cross-validation* helps overcome this problem by allowing us to have an early stopping set but still get to train on all the data.

The training set of size N is divided up into k folds. Each fold contains N/k training cases. We train on folds 1 through $k-1$, and use fold k for early stopping. Then we train another neural net on folds 2 through k and use fold 1 for early stopping, and so on. This way we get k models that have seen all of the training data. We get the final predictions by averaging the predictions from each fold. Also, it has been shown that averaging multiple models tends to give better performance than a single model trained on all of the data. We can also keep one fold for error estimation, depending on our requirements.

3.1.2 Active learning

Another technique that helps reduce the number of points we need for getting a good model is *active learning*. The aim of this technique is to get the smallest sample that will give the most benefit when used for training. We use the active learning technique proposed by Ipek et al [1]. Whenever points are added to the training set, the point selection is based on the predictions of the nets trained so far. The top n points with most disagreement between the fold nets are chosen to be added to the training set. The idea here being that if the nets agree for a given data point, they have already learned what they need to know about it. However, if they disagree on it, then training on that data point might contribute to learning the target function better.

3.2 Implementation

We use the Stuttgart Neural Network Simulator (SNNS) [5] package for our neural network needs. Cross-validation and active learning are implemented using bash and python scripts. We use a neural network with 16 hidden units trained using min-max scaling and resilient backpropagation. We use N_B, M_U, N_U, K_U, L_s, FMA, F_F, I_F and N_F as our inputs to the neural network, and $MFLOPS$ as the output. We use 4-fold cross-validation and active learning. We use 3 folds for training and 1 fold for early stopping and error estimation. We start the algorithm with 16 points and keep adding to the training set in steps of 16 until it reaches 96 points. Once training is finished, we go back and query our models for the best prediction.

Since the compiler tends to slow down severely with unreasonable inputs, we place some constraints on the search space using the ATLAS constraints as guidelines.

Steps for optimizing BLAS using ANNs:

1. Take random sample of 16 points from the search space.
2. Run sets of parameters using ATLAS code generator and form initial training set.
3. Train on the training set.
4. Add another 16 points to the training set using feedback from active learning.
5. Train on the training set.
6. Repeat steps 4-5 till training set is 96 points.
7. Query models for best prediction and verify.

3.3 Results

Figures 2 through 8 show learning curves as more cases are added to the training set. The horizontal line represents performance acheived by ATLAS CGw/S. Figure 9 shows performance that the ANN approach gets compared to ATLAS CGw/S. The performance numbers are normalized to those of ATLAS CGw/S and vary between 85% to 111%. Figure 10 shows runtimes normalized to ATLAS CGw/S. The runtimes vary between 13% and 202% of those acheived by ATLAS. One thing to note is that the Opteron 240, Athlon MP, and Pentium III show the most slowdown using the ANN approach compared to ATLAS, but their optimum points are found much before training on the entire 96 points ends (Figures 5, 6, 7). On the Pentium 4, the ANN approach takes 25% of the time as ATLAS CGw/S and gets 111% of the performance. For the Itanium 2, we can get better performance by loosening the bound on NB. We intend to generate platform-specific bounds using a tool like X-Ray [9] so that we can have a better idea of the

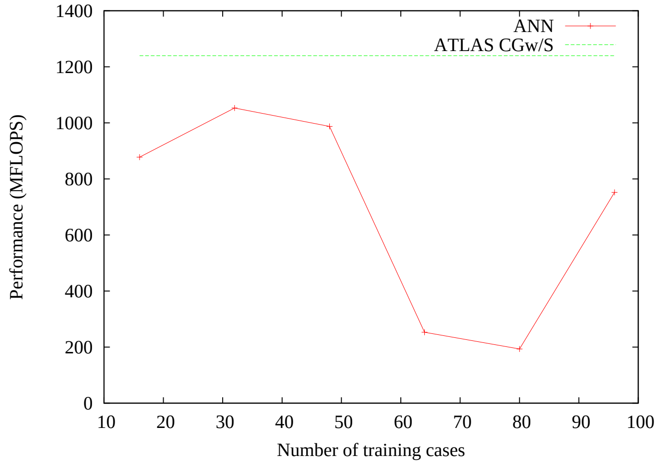

Figure 2. ANN: Power3

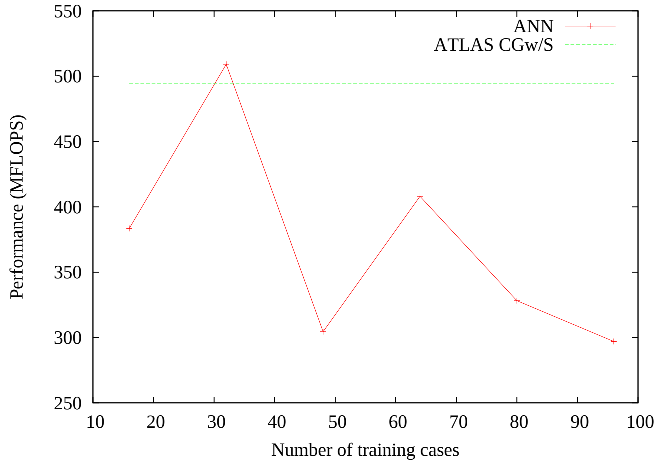

Figure 3. ANN: R12K

search space, as part of our future work. Overall speed can also be improved by coding the python and bash components in C, and also by using better active learning techniques.

There are no results for the Alpha 21264 and UltraSPARC III using the ANN approach because we were unable to get our software working on these machines by the time of this submission.

4. Genetic Algorithms

4.1 Background

Genetic algorithms are used to search large spaces in a reasonable amount of time, achieving "good enough" results. The algorithms were designed to mimic the behavior of biological systems. They were first proposed by John Holland [3] and since then have been used for a variety of purposes by many others. The chapter on genetic algorithms in Mitchell's book [4] provides a good overview of the topic.

Genetic algorithms are a form of randomized hill-climbing. Each cycle through the algorithm, called a generation, involves first creating a new set of inputs based on the previous generation, followed by an evaluation to find the best results.

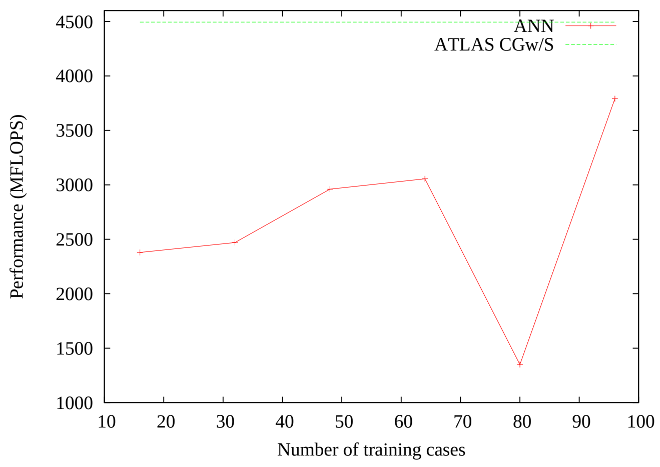

Figure 4. ANN: Itanium 2

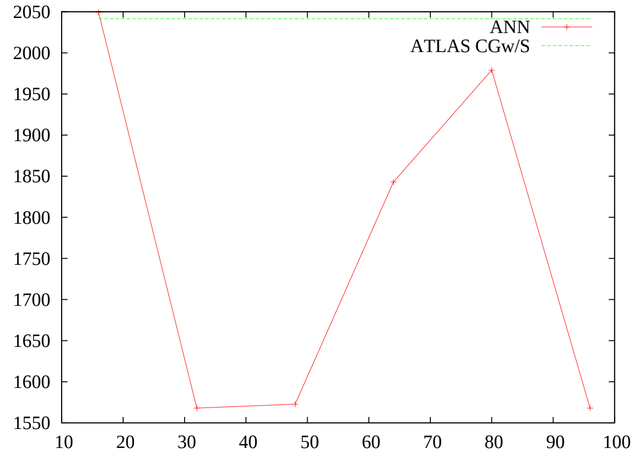

Figure 5. ANN: Opteron 240

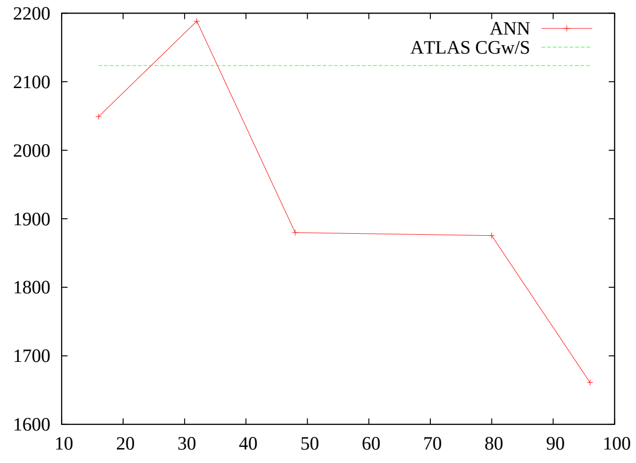

Figure 6. ANN: Athlon MP

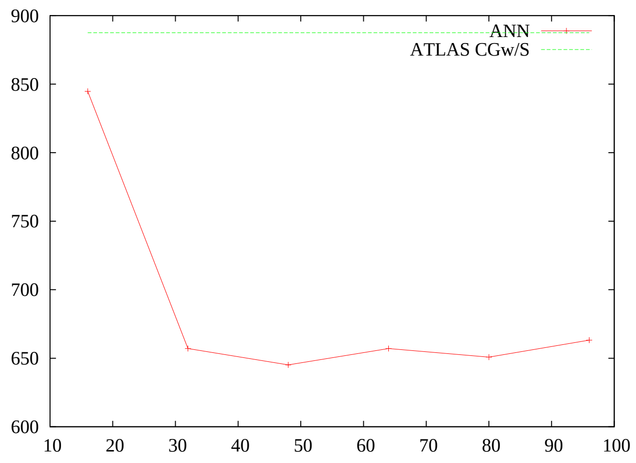

Figure 7. ANN: Pentium III

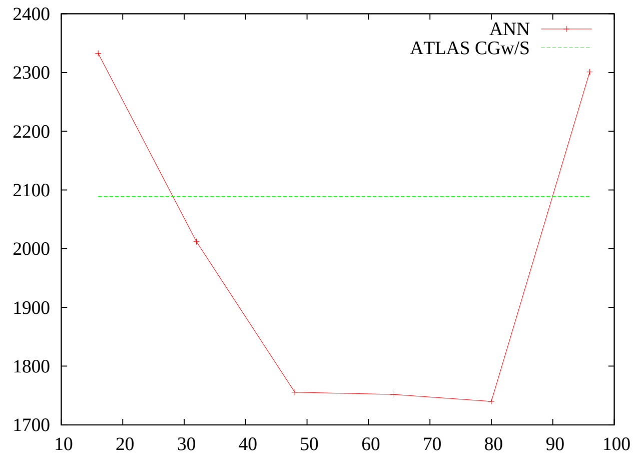

Figure 8. ANN: Pentium 4

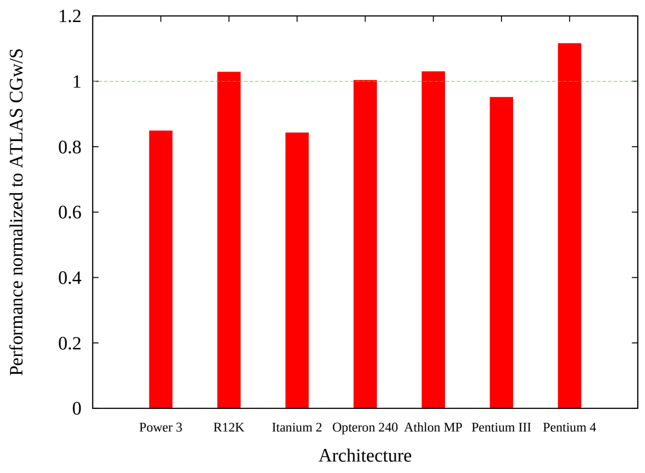

Figure 9. ANN: Performance compared to ATLAS CGw/S

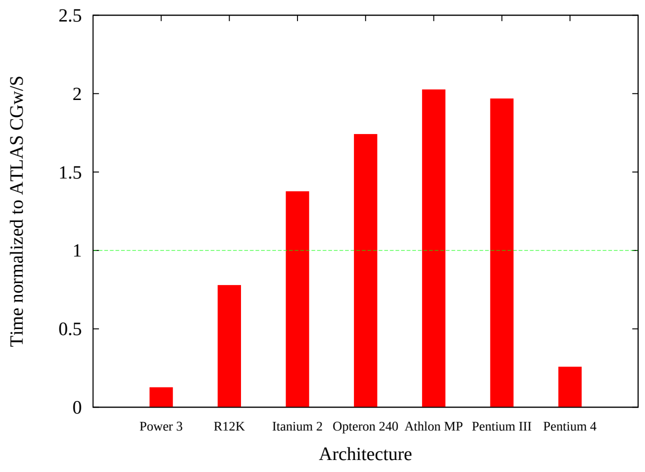

Figure 10. ANN: Runtime compared to ATLAS CGw/S

The initial state of the input sets is typically completely randomized. The evaluation routine is called a *fitness function* and is applied to all of the sets. Some number of the best sets are kept, and the remaining potentially have modifications applied to them in order to create the input sets for the next generation. The two most common modifications used are *mutation* and *crossover*. In the mutation operation, some of the inputs are changed by a random amount, thus adding new or modified values into the input pool. In crossover, two input sets are picked, and a random subset of the inputs is swapped from one set to the other. This is done in the hopes of taking desirable inputs from the two parent sets and combining them in a way that creates a superior child set.

After the mutation and crossover phases, the cycle is repeated and the fitness function is run again. The algorithm is stopped either after a fixed number of generations, or after some threshold value of fitness has been achieved.

Genetic algorithms have the following good qualities:

- Rapidly find a “good” (though maybe not the best) solution.
- Can be easily parallelized.
- Can search a large space.

Genetic algorithms have the following limitations:

- May require a large number of runs.
- The inherent randomness leads to some non-determinism in results.
- A large set of input sets is required to keep the algorithm from becoming stuck at local minima.
- No easy way to determine how to tune the various factors in the algorithm for best performance.

4.2 Implementation

Our genetic algorithm implementation is written in C. The C program keeps track of the input sets, performs all of the mutations and crossovers, and calls an outside program to measure performance. The actual fitness function used is the mini-mm infrastructure provided by the vogue framework and ATLAS. The inputs are passed to the mini-mm script, which creates the sample benchmark with the proper parameters, runs the benchmark, and returns the number of MegaFLOPS which is then read in by the C program.

The search space is kept as open as possible. Unfortunately the compiler tends to choke on unreasonable inputs (sometimes getting stuck using huge amounts of memory and CPU and taking hours to complete) so some limits were placed on the inputs using the ATLAS provided constraints as guides. The only non-boolean value with an upper limit set is L_s, kept less than or equal to 6. All of the inputs had lower limits. N_B could not go lower than 16, and was constrained to be a multiple of 4. M_U, N_U, and N_F cannot be less than 1, and K_U cannot be less than 4. I_F cannot be less than 2, and boolean values F_F and FMA are enforced as such.

The standard C library random number generator is used for all random decisions. It is seeded at run-time by the current time returned by the time() function.

Each generation evaluates 9 input sets. The top result is found, and is compared against the previous best which is stored separately. If the current result is better than the old best, the old best is replaced. To further keep the good values in circulation, each generation the worst performer is replaced by the current best result.

Mutation is handled next. In the current setup, 20 mutations happen per generation. The set and parameter is picked completely random. A value between -16 and 16 is added to the parameter chosen. To avoid getting stuck at a local minima, there is a 25% additional chance that this value added will be doubled. For boolean values, a separate mutation function is used that only results in 0 or 1.

For crossover we use *uniform crossover*, which means that each value inside a set undergoing this operation has a 50% chance of moving over. The other possible ways of crossing over involve moving over contiguous blocks of values, but we arbitrarily chose to use the uniform method. We have 5 crossovers happening per generation.

After undergoing mutation and crossover, the values are checked to be sure they meet the limits described earlier. Then the fitness function is applied, and the cycle repeats.

In order to keep the algorithm running in a timely manner, the C program monitors the fitness calculation, and if it takes more than 5 minutes to complete the result is reported as invalid and the test involved is killed.

4.3 Results

The genetic algorithm was run for 250 generations on each platform. The experiment was repeated at least twice if time allowed. The best result obtained was chosen for comparison purposes. In most cases the best overall result was within 5% of the worst result. The only exception was the Itanium 2 case where the worst run was 10% slower than the best.

The results on the Pentium III architecture proved to be invalid when independently tested. For some reason the ATLAS infrastructure occasionally reports invalid MegaFLOPS ratings on this particular machine, and this poisoned the genetic algorithm. At the time of writing this paper the reasons for this bug have not been determined.

The results from the runs are shown in Figures 11 through 18.

On the plots, the resulting performance in MegaFLOPS is shown versus number of generations required. The N_B parameter is also shown (potentially scaled to make it easier to see), as it is the one most closely related to performance. The horizontal line shows the performance found by ATLAS's search, and the vertical line show the approximate time taken by ATLAS to find that result.

In all cases, the genetic algorithm found good results. In 5 of the cases it found better results than ATLAS did, although in only 3 of those (Alpha, MIPS, and Pentium 4) did it do this faster than ATLAS did.

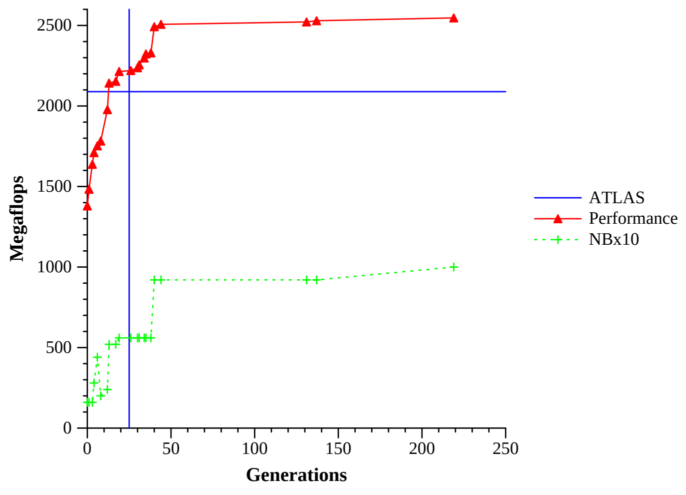

Figure 11. Genetic Algorithm: Pentium 4

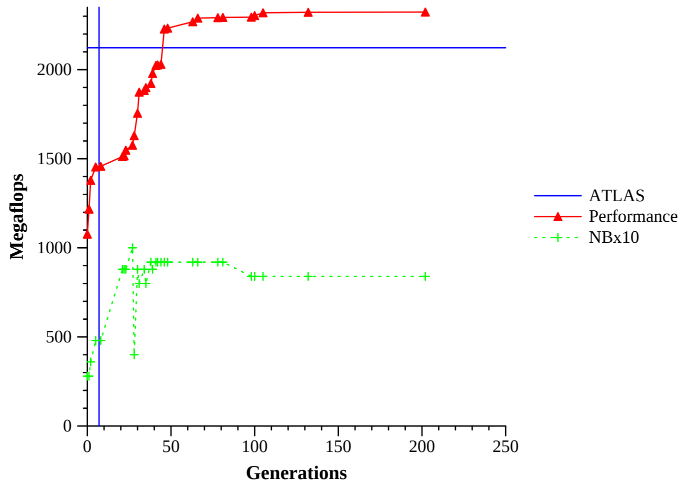

Figure 12. Genetic Algorithm: Athlon

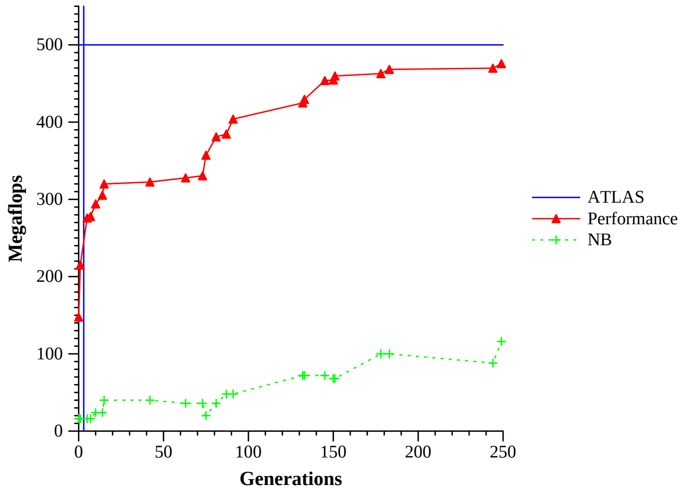

Figure 13. Genetic Algorithm: SPARC

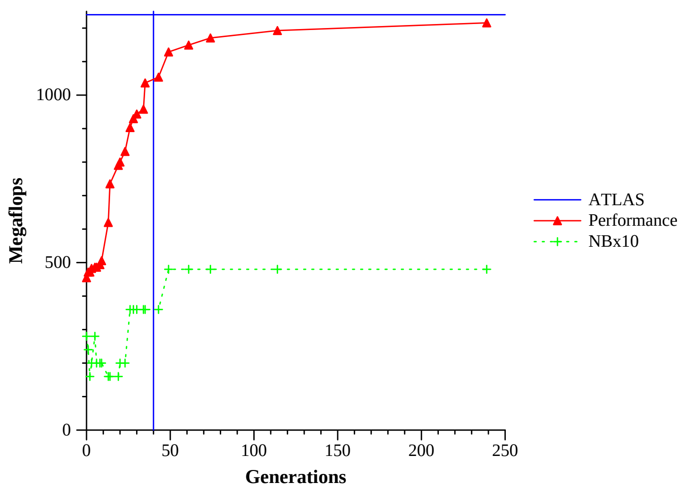

Figure 14. Genetic Algorithm: Power3

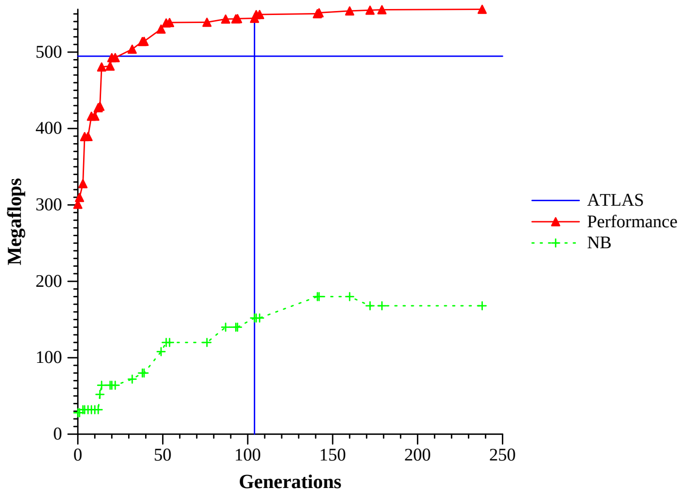

Figure 15. Genetic Algorithm: MIPS

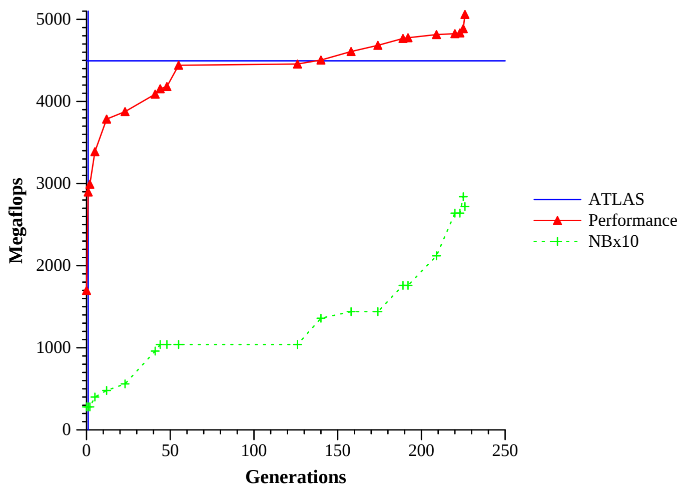

Figure 16. Genetic Algorithm: ia64

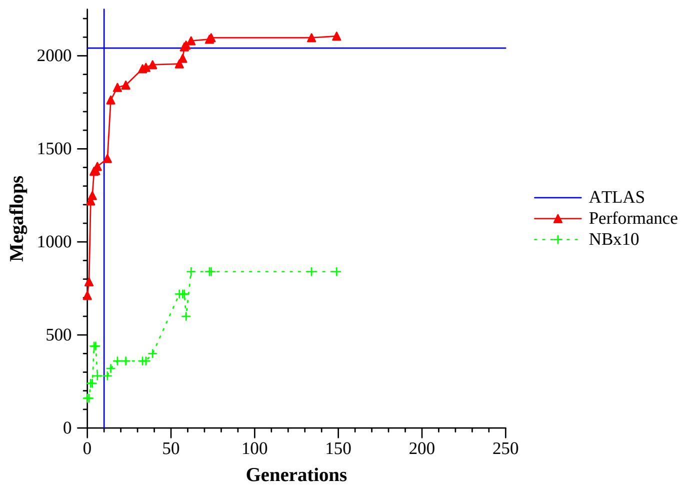

Figure 17. Genetic Algorithm: Opteron

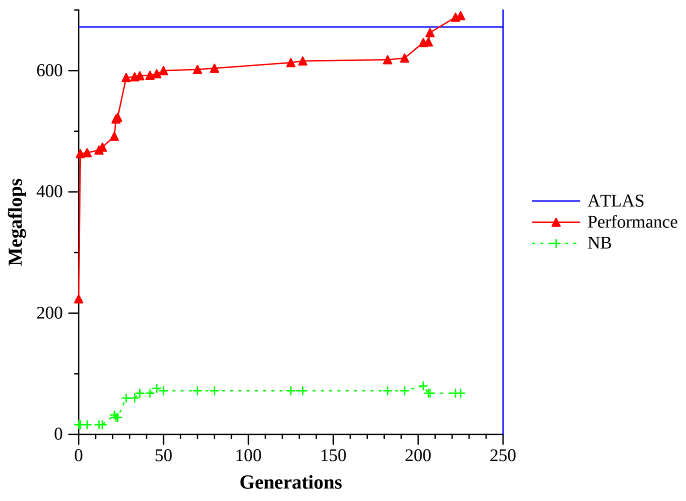

Figure 18. Genetic Algorithm: Alpha

5. Conclusions and Future Work

Using the ANN approach, we see that we can achieve results comparable to ATLAS for most cases (within 85% to 111%). Unlike ATLAS, which only gives a string of optimal values, the ANN approach gives us a model that can be queried for other points in the search space. This can be useful for obtaining a feel for the structure of the parameter space, especially when applied to new problems.

For future work, we intend to code the bash/python part of the software in C to improve portability and also to attain a faster runtime. It would be also be interesting to explore what benefits can be attained by using platform-specific bounds using a tool like X-Ray [9]. Another possible approach using ANNs could be to have a global model for all architectures, which would be exactly what Yotov et al [8] came up with, except it would not require a deep understanding of the application or the architecture it is running on.

For genetic algorithms, the results show that good results can be found with an unbounded search space. Given enough time, genetic algorithms can find results that are approaching that of ATLAS's, and in many cases even better. More interesting is that the results found are often outside of the narrow range ATLAS searches in.

For the future, more analysis needs to be done with the genetic algorithm results. Many of the best results have high values for N_B which Whalen warns might have negative impacts when the BLAS is run in actual high-performance applications [6]. If the values returned by the genetic algorithms turn out to have good behavior, it might be worth the trouble of having a tournament setup for determining optimal BLAS, choosing either the result of ATLAS or the genetic algorithm, whichever is better.

Overall, applying machine learning to self-optimizing systems is a net win and should be investigated more thoroughly. Our work can be extended to other library generators (e.g., FFTW) and other similar problems with minimal effort. Some work remains to be done to see if setting reasonable bounds can increase performance and reduce runtime, but overall our setup has been show to be practical and useful.

References

[1] Engin Ipek and Sally A. McKee and Martin Schulz and Bronis R. de Supinski and Rich Caruana. Efficiently exploring architectural design spaces via predictive modeling. In *Under Review*.

[2] 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 *LCPC*, 2005.

[3] J. H. Holland. *Adaption in natural and artificial systems*. MIT Press, Cambridge MA, 1975.

[4] T. M. Mitchell. *Machine Learning*, chapter 9, pages 249–273. The McGraw-Hill Companies, Inc., New York, 1997.

[5] U. of Stuttgart. Snns: Stuttgart neural network simulator.

[6] R. C. Whaley. Atlas faq, 2006.

[7] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998. CD-ROM Proceedings. **Winner, best paper in the systems category.**
URL:`http://www.cs.utsa.edu/~whaley/papers/atlas_sc98.ps`.

[8] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas? In *Proceedings of the IEEE, Special issue on Program Generation, Optimization, and Adaptation*, 2005.

[9] K. Yotov, K. Pingali, and P. Stodghill. X-ray: A tool for automatic measurement of hardware parameters. In *International Conference on Quantitative Evaluation of SysTems*, 2005.

[10] H. You, K. Seymour, and J. Dongarra. An effective empirical search method for automatic software tuning. Technical report, University of Tennessee, May 2006.

Feature	Value
CPU	Alpha 21264
Architecture	Out-of-Order, RISC
CPU Core Frequency	500MHz
L1 Data Cache	64KB
L1 Instruction Cache	64KB
L2 Unified Cache	4MB
Floating-Point Registers	32
Foating-Point Functional Units	2
Floating-Point Multiply Latency	4
Fused Multiply-Add	No
Operating System	Tru64 5.1
C Compiler	gcc 4.0

	N_B	M_U, N_U	L_s	FMA	F_F, I_F	MFLOPS
		K_U			N_F	
CGw/S	56	4, 2, 1	4	0	1, 4, 2	672
ANN	-	-, -, -	-	-	-, -, -	-
Genetic	68	2, 3, 4	2	0	1, 44, 42	691

Table 2. Alpha 21264: Parameters

Feature	Value
CPU	Power3
Architecture	Out-of-Order, RISC
CPU Core Frequency	375MHz
L1 Data Cache	64KB
L1 Instruction Cache	32KB
L2 Unified Cache	4MB
Floating-Point Registers	32
Foating-Point Functional Units	2
Floating-Point Multiply Latency	4
Fused Multiply-Add	Yes
Operating System	Linux 2.4.18
C Compiler	gcc 3.3

	N_B	M_U, N_U	L_s	FMA	F_F, I_F	MFLOPS
		K_U			N_F	
CGw/S	48	3, 4, 48	6	1	0, 6, 1	1240
ANN	80	8, 1, 1	5	1	0, 2, 1	1053
Genetic	48	26, 1, 50	1	1	1, 4, 1	1216

Table 3. Power3: Parameters

Feature	Value
CPU	MIPS R12K
Architecture	Out-of-Order, RISC
CPU Core Frequency	300MHz
L1 Data Cache	32KB
L1 Instruction Cache	32KB
L2 Unified Cache	4MB
Floating-Point Registers	32
Foating-Point Functional Units	2
Floating-Point Multiply Latency	2
Fused Multiply-Add	Yes
Operating System	IRIX 6.5.22
C Compiler	SGI MIPSPro 7.2.1

	N_B	M_U, N_U	L_s	FMA	F_F, I_F	MFLOPS
		K_U			N_F	
CGw/S	60	5, 5, 1	5	1	0, 25, 1	495
ANN	80	2, 7, 1	6	1	0, 2, 1	509
Genetic	168	3, 6, 39	1	1	1, 2, 1	556

Table 4. MIPS R12k: Parameters

Feature	Value
CPU	UltraSPARC II
Architecture	Out-of-Order, RISC
CPU Core Frequency	360MHz
L1 Data Cache	?
L1 Instruction Cache	?
L2 Unified Cache	?
Floating-Point Registers	?
Foating-Point Functional Units	?
Floating-Point Multiply Latency	?
Fused Multiply-Add	?
Operating System	Solaris 9
C Compiler	Sun Studio 11

	N_B	M_U, N_U	L_s	FMA	F_F, I_F	MFLOPS
		K_U			N_F	
CGw/S	40	4, 4, 1	4	0	0, 8, 2	500
ANN	-	-, -, -	-	-	-, -, -	-
Genetic	116	2, 7, 4	4	0	1, 52, 13	476

Table 5. UltraSPARC II: Optimization Parameters

Feature	Value
CPU	Itanium 2
Architecture	In-Order EPIC IA-64
CPU Core Frequency	1.5GHz
L1 Data Cache	16KB
L1 Instruction Cache	16KB
L2 Unified Cache	256KB
L3 Unified Cache	3MB
Floating-Point Registers	128
Foating-Point Functional Units	2
Floating-Point Multiply Latency	4
Fused Multiply-Add	Yes
Operating System	Linux 2.6.9-5.0.5.EL
C Compiler	icc 9.0

	N_B	M_U, N_U	L_s	FMA	F_F, I_F	MFLOPS
		K_U			N_F	
CGw/S	120	6, 6, 1	6	1	1, 36, 1	4495
ANN	80	8, 2, 1	3	1	0, 2, 1	3791
Genetic	272	6, 5, 4	1	0	0, 53, 45	5060

Table 6. Itanium 2: Parameters

Feature	Value
CPU	Opteron 240
Architecture	Out-of-Order, CISC, x86-64
CPU Core Frequency	1.4GHz
L1 Data Cache	64KB
L1 Instruction Cache	64KB
L2 Unified Cache	1024KB
Floating-Point Registers	8 x87
Foating-Point Functional Units	ADD + MUL+ MEM
Floating-Point Multiply Latency	4
Fused Multiply-Add	No
Operating System	Linux 2.6.9-perfctr
C Compiler	gcc 3.4.4

	N_B	M_U, N_U	L_s	FMA	F_F, I_F	MFLOPS
		K_U			N_F	
CGw/S	60	6, 1, 60	2	1	0, 5, 1	2041
ANN	60	6, 1, 80	1	1	0, 2, 1	2049
Genetic	84	6, 1, 43	1	1	1, 55, 1	2106

Table 7. Opteron 240: Parameters

Feature	Value
CPU	Athlon MP
Architecture	Out-of-Order, CISC, x86
CPU Core Frequency	2.1GHz
L1 Data Cache	64KB
L1 Instruction Cache	64KB
L2 Unified Cache	256KB
Floating-Point Registers	8
Foating-Point Functional Units	ADD + MUL+ MEM
Floating-Point Multiply Latency	4
Fused Multiply-Add	No
Operating System	Linux 2.6.9-perfctr
C Compiler	gcc 3.4.5

	N_B	M_U, N_U K_U	L_s	FMA	F_F, I_F N_F	MFLOPS
CGw/S	60	5, 1, 20	6	1	0, 5, 1	2124
ANN	80	6, 1, 80	6	1	0, 2, 1	2188
Genetic	84	5, 1, 64	5	1	0, 45, 16	2323

Table 8. Athlon MP: Parameters

Feature	Value
CPU	Pentium III
Architecture	Out-of-Order, CISC, x86
CPU Core Frequency	1.266GHz
L1 Data Cache	16KB
L1 Instruction Cache	16KB
L2 Unified Cache	512KB
Floating-Point Registers	8
Foating-Point Functional Units	1
Floating-Point Multiply Latency	5
Fused Multiply-Add	No
Operating System	Linux 2.4.21-40.ELsmp
C Compiler	gcc 3.2.3

	N_B	M_U, N_U K_U	L_s	FMA	F_F, I_F N_F	MFLOPS
CGw/S	40	2, 1, 40	5	0	0, 2, 1	888
ANN	80	3, 1, 80	1	1	0, 2, 1	845
Genetic	-	-	-	-	-	-

Table 9. Pentium 3: Parameters

Feature	Value
CPU	Pentium 4
Architecture	Out-of-Order, CISC, x86
CPU Core Frequency	2.8GHz
L1 Data Cache	8KB
L1 Instruction Cache	12K μOPs
L2 Unified Cache	512KB
Floating-Point Registers	8
Foating-Point Functional Units	1
Floating-Point Multiply Latency	7
Fused Multiply-Add	No
Operating System	Linux 2.6.11.4-21.10-smp
C Compiler	gcc 3.3.5

	N_B	M_U, N_U K_U	L_s	FMA	F_F, I_F N_F	MFLOPS
CGw/S	72	1, 6, 72	1	0	0, 7, 1	2089
ANN	80	3, 1, 80	1	1	0, 2, 1	2332
Genetic	100	4, 1, 126	1	1	0, 36, 7	2547

Table 10. Pentium 4: Parameters