JIT Compilation of Common Kernels in Sequoia using Supervised Learning

Michael Bauer Kushal Tayal

Department of Computer Science
Stanford University
{mebauer, ktayal }@stanford.edu

1 Introduction

Sequoia is a portable, locality-aware parallel programming language designed to make it easier for scientists and engineers to write high-performance codes that can easily be transported across different super computers. Sequoia is able to achieve this goal by directly programming the memory hierarchy in a machine agnostic way. In order to make code independent of any one architecture, Sequoia code is parametrized with tunable variables that are specified at compile-time for a given architecture. Whenever a piece of code is to be transferred to a new machine, it is the responsibility of the programmer to fill in the compile-time tunables in a mapping file that is designed to map the code onto a given architecture. This currently requires that the problem sizes, and therefore the tunable variables are known statically.

One of the future goals of Sequoia is to become more dynamic in its ability to execute programs by off-loading some of the work of the compiler onto the runtime for the cases where information is unknown at compile-time. The need to off-load work onto the runtime could be either a result of unknown problem sizes or changing machine conditions. In either case, the runtime will need to be able to dynamically just-in-time (JIT) compile code based on a problem size and a target architecture. This will require the runtime to be able to compute the tunable constants to be passed to the compiler very efficiently. The need to do this with minimal overhead will be paramount to the ability of the runtime to achieve good performance.

In order to facilitate the ability of the runtime to efficiently JIT code, we propose to build models of some commonly occurring kernels in Sequoia programs so that these models will enable the runtime to efficiently generate a mapping file for a given architecture. These models will be functions that will take as arguments the data size and the attributes of the target architecture, and quickly compute the tunable variables to be passed to the compiler. Since these are commonly occurring kernels we can build the models offline using machine learning techniques and then embed the models in the runtime. The goal of this work is to discover whether or not it is feasible to build such models using various machine learning techniques.

2 Problem Definition

The goal of this project was to be able to easily build models of common Sequoia kernels so we can automatically generate tunable variables given a data size and target architecture. We therefore stated our problem to be: given a feature vector consisting of the input problem size and the parameters defining the target architecture (number of levels, number of children per level, memory sizes), generate the tunable variables for the associated machine.

3 Application Kernels

Before we began our study of various machine learning techniques to apply to our problem, we first fixed our set of input kernels[1]. Our goal in this paper was to investigate the ability to machine learning techniques to build models for commonly occurring kernels in the Sequoia programming language. We chose four kernels that tend to appear in many scientific computing applications and therefore would be good case studies for this project:

- SAXPY - vector times a scalar plus another vector
- SGEMV - dense matrix-vector multiplication
- Matrix Multiplication - dense matrix-matrix multiplication
- CONV2D - two-dimensional convolution

These kernels appear in many different Sequoia applications and also correspond to different points in the space of computation-to-communication ratios[2]. This therefore makes them a good set of kernels for our experiments.

4 Data Collection

The first step in our project was to generate a data set corresponding to various problem sizes and target architectures for our chosen kernels. We constructed a test suite by generating mapping files corresponding to 5 different problem sizes across 7 different machine architectures.

- SMP2 - A 2 node SMP machine
- SMP32 - A 32 node SMP machine
- CMP4 - A 4 core CMP machine
- Cluster32 - A 32 node MPI cluster
- Cluster+SMP16/2 - A 16 node MPI Cluster with 2 cores on each node
- GPU1 - A machine with a single NVIDIA Tesla C2050
- GPU2 - A machine with a pair of NVIDIA GTX 480s

[1]Here we are referring to computational kernels and not kernels in the machine learning sense as applied to SVMs.

[2]Computation-to-communication ratios are in important factor in determining the performance of parallel codes

In some cases the machines have multiple levels to the memory hierarchy. We modeled each level as a different input feature vector as each level has to generate its own output mapping parameters. For each kernel we currently have 60 different mappings in our training set. We have constructed this data set to demonstrate several important mapping techniques across different machines and problem sizes with the goal that the machine learning algorithms will be able to duplicate these techniques.

We also constructed a series of three test problems for each of the different kernels that we were attempting to model. Each kernel has a different test problem for the following architectures: SMP, Cluster, and GPU. By testing on different architectures we are able to observe whether or not the machine learning algorithms are capable of detecting the subtle differences in mapping techniques that can result in varying levels of performance.

5 Model Construction

Based on our previous experience using an autotuner with Sequoia, we felt that it would be necessary to have a separate model for each of the different kernels. The kernels themselves have vastly different performance characteristics and should therefore be modeled separately. However, we felt that a model for a given kernel should be able to represent all the potential target architectures. We therefore explored several different techniques for constructing models for our given kernels.

5.1 Linear Regression

In order to get something running, the first technique we tested is basic linear regression which fits a linear curve to the set of input vectors. Normalizing the input vector increased the training error and thus we applied linear regression to unnormalized data. We used the normal equation for running linear regression.

$$\theta = (X^T X)^{-1} X^T y \tag{1}$$

Our experiments (see section 6) indicated that linear regression was insufficient for capturing the interesting mapping techniques to achieve good performance.

5.2 Logistic Regression

After looking at the results of linear regression, we noticed that the features of the input vector have a varied range. We determined that fitting a linear regression to such an input may not be the optimal solution. The second method we tried was applying logistic regression. Logistic regression fits a logistic curve to the input vectors and outputs a value in the range $(0, 1)$. For this purpose, we scaled our input and output training vectors by dividing each feature space by a maximum value specific to that feature. The test vectors were also scaled by the same maximums and the output vector was re-scaled back to original range. We used the batch gradient descent method for minimizing the LMS expression.

```
Repeat until convergence {

    $\theta_j := \theta_j + \alpha \sum_{i=1}^{m} (y_{(i)} - g(\theta^T x^{(i)}))(g(\theta^T x^{(i)}))(1 - g(\theta^T x^{(i)}))x_j^{(i)}$

}
```

The function $g(z)$ here is the traditional logistic equation. We used a brute force method to compute the optimum value of the learning rate(α) and the number of iterations based on the training error values.

5.3 SVM-Like Regression

Looking at the output of the previous two regression algorithms, another thing that we noticed was that the algorithms were having trouble dealing with the extreme variations in parameter ranges that existed across the various features. We then decided that it might be useful to map our vectors into a higher dimensional space where the higher dimensional space would include applying functions to the initial feature vectors. Our hope was that this would enable the algorithm to deal better with the extreme discrepancy in the ranges of different features. Unfortunately, our restricted training data set size prevented us from actually mapping the kernels into a higher dimensional space as we only had 60 vectors for each kernel[3], and we would have been unable to avoid under-fitting in a higher dimensional space.

Since we couldn't directly map the features into a higher dimensional space using some kernel, we did the next best thing: apply different functions to each feature in order to scale each feature. We chose five different functions to apply to each individual feature:

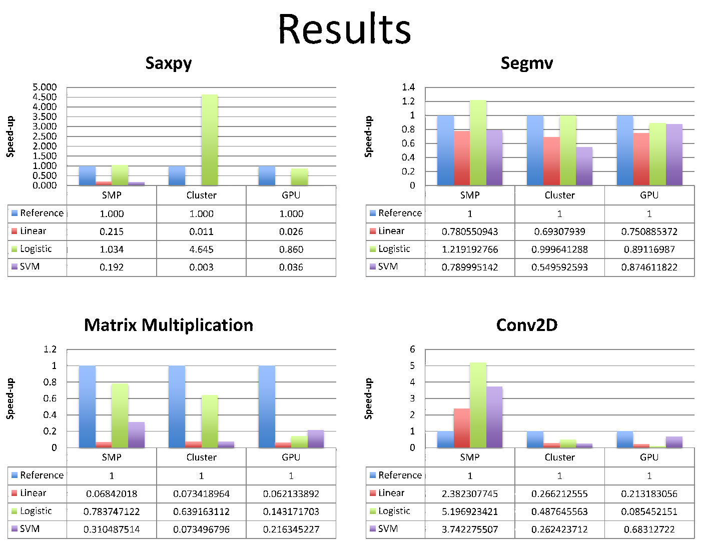
- $f(x) = x$
- $f(x) = log_2(x)$
- $f(x) = exp(x)$
- $f(x) = sqrt(x)$
- $f(x) = x^2$

Rather than attempt to be intelligent about our choice of function to apply to each feature, we simply tried all combinations of different functions applied to each feature. For matrix multiplication which has 10-dimensional input vectors we then ended up testing 5^{10} different combinations. We then applied linear regression to each of the modified training sets and selected the set of function applications that minimized the difference between the output vectors.

6 Performance Results

Due to the fact that our learning problem was a regression algorithm and not a classification problem we determined that the actual performance numbers attained by our generated mapping files would be a better metric than absolute difference between the expected output vectors and our computed vectors. We ran each of our three test vectors for each kernel through each of our learning algorithms and generated the equivalent mapping files. We then used Sequoia to compile the generated code for each of the three target architectures: an MPI cluster, a 32-core SMP, and an NVIDIA Tesla C2050 GPU. The performance speedups attained relative to a hand coded implementation can be seen in figure 6.

[3]Generating training problems is very time consuming as they have to be done by hand currently and require expert knowledge.



Saxpy

	SMP	Cluster	GPU
🟦 Reference	1.000	1.000	1.000
🟥 Linear	0.215	0.011	0.026
🟩 Logistic	1.034	4.645	0.860
🟪 SVM	0.192	0.003	0.036

Segmv

	SMP	Cluster	GPU
🟦 Reference	1	1	1
🟥 Linear	0.780550943	0.69307939	0.750885372
🟩 Logistic	1.219192766	0.999641288	0.89116987
🟪 SVM	0.789995142	0.549592593	0.874611822

Matrix Multiplication

	SMP	Cluster	GPU
🟦 Reference	1	1	1
🟥 Linear	0.06842018	0.073418964	0.062133892
🟩 Logistic	0.783747122	0.639163112	0.143171703
🟪 SVM	0.310487514	0.073496796	0.216345227

Conv2D

	SMP	Cluster	GPU
🟦 Reference	1	1	1
🟥 Linear	2.382307745	0.266212555	0.213183056
🟩 Logistic	5.196923421	0.487645563	0.085452151
🟪 SVM	3.742275507	0.262423712	0.68312722

Figure 1: Performance results of learning algorithms on four different Sequoia kernels across three different architectures.

From these performance numbers we noticed several interesting facts. The logistic regression technique performed the best. It was able to attain similar performance to the hand coded version for both SMP and cluster architectures across all four kernels. In some cases logistic was even able to exceed the performance of a hand coded implementation by discovering that the problem size was small enough to not make use of additional parallel resources due to runtime overheads. The logistic regression did not however perform as well on the GPU architecture. After looking at the generated mappings, we concluded that this is due to the fact that a GPU architecture is a four-level tree of memories (as opposed to two levels for the SMP and cluster) and the algorithm wasn't able to capture the interesting cross level mapping techniques.

Both the linear and SVM-like algorithms did not perform as well as the hand-coded implementations. In some cases they were able to achieve comparable performance, but weren't consistent enough to be able to be useful in a real-world application.

7 Conclusion and Future Work

From the results of this project we've determined that using offline machine learning to build models of commons scientific computing kernels has the potential to be useful in a JIT environment. However, before we implement this technology we must first discover a way to capture the interesting mapping techniques applied for deep (more than two level) memory hierarchies. This will be crucial to our success as most super computers that are targeted by Sequoia consist of at least three memory levels.

In the future we plan to continue this work by building larger training sets that should enable us to use SVM techniques to map our problems into higher dimensional spaces. Our hope is that this technique will capture the interesting mapping techniques for deep memory hierarchies.