Optimizing Matrix Multiplication with a Classifier Learning System *

Xiaoming Li and María Jesús Garzarán

Department of Computer Science
University of Illinois at Urbana-Champaign
{xli15, garzaran}@cs.uiuc.edu
http://polaris.cs.uiuc.edu

Abstract. Compilers have been very successful on automating the process of program optimization, but there is still a significant difference in performance between the code generated by the compiler and the hand-optimized code. Library generators such as ATLAS, SPIRAL, and FFTW address this problem by using empirical search to find the parameter values of certain optimization such as degree of unroll. We have recently developed a generator of sorting routines. Sorting differs from the algorithms implemented by other library generators in that performance of sorting depends not only on the target platform but also on the characteristics of the input data. In our work we used a classifier learning system to generate sorting routines that are capable of adapting to the input data. In this paper we follow a similar approach and use a classifier learning system to generate high performance libraries for matrix-matrix multiplication. Our library generator produces matrix multiplication routines that use recursive layouts and several levels of tiling. Our approach is to use a classifier learning system to search in the space of the different ways to partition the input matrices the one that performs the best. As a result, our system will determine the number of levels of tiling and tile size for each level depending on the target platform and the dimensions of the input matrices.

1 Introduction

Compilers have been very successful on automating the process of program optimization, but there is still a significant difference in performance between the code generated by the compiler and the hand-optimized code. The growing complexity of the architectural features of modern processors makes it very difficult to optimize performance. An approach that some researchers have followed is to use library generators to generate high performance code for some specific problem domains.

Examples of well-known library generators are ATLAS [30], PHiPAC [4], FFTW [11] and SPIRAL [33]. ATLAS and PHiPAC generate linear algebra routines and focus the

* This work was supported in part by the National Science Foundation under grant CCR 01-21401 ITR; by DARPA under contract NBCH30390004; and by gifts from INTEL and IBM. This work is not necessarily representative of the positions or policies of the Army or Government.

optimization process on the matrix multiplication routine. During installation, the parameter values of a matrix multiplication implementation, such as tile size and amount of loop unrolling, that deliver the best performance are identified using empirical search. This search proceeds by generating different versions of matrix multiplication that only differ in the parameter value that is being sought. An almost exhaustive search is used to find the best parameter values. The other two systems mentioned above, SPIRAL and FFTW, generate signal processing libraries.

Recently we have built a library generator for sorting [17, 18]. Sorting is different from the algorithms implemented by the previous library generators in that performance of sorting depends not only on the target platform but also on characteristics of the input data, which are only known at runtime. In the work presented in [18] we used a classifier learning system to generate algorithms capable of adapting to the input data. In the work discussed herein, we follow a similar approach and use a classifier learning system to generate high performance libraries for matrix-matrix multiplication (MMM). Our library generator generates MMM routines that use recursive layouts [7, 8] and several levels of tiling. Our approach is to use a classifier learning system to search among all the different ways to partition the input matrices, the one that performs the best. The MMM routine generated with our classifier learning system uses different levels of tiling and tile sizes based on the dimensions of the matrices and the architectural features of the target machine.

ATLAS is a library generator that also produces a MMM routine. The difference between our approach and the one followed by ATLAS is that we use recursive layouts to place the blocks in consecutive memory locations and focus the search on levels of tiling and size of each tile. ATLAS does not search for the number of levels of tiling. In fact, ATLAS only searches for the tile size for a single level of tiling, although a second level of tiling can be implemented [1]. Also, notice that the performance delivered by ATLAS in some platforms is still far from the one delivered by the vendor provided libraries [36], mainly because ATLAS does not take into account all the levels of the memory hierarchy and does not take advantage of some optimizations like prefetching. Our objective is to reduce the performance gap between the hand-optimized code and the automatic generated code by extending the search to consider parameters ignored by ATLAS.

When using a single level of tiling, it has been shown that a model can predict the best value of the tile size almost as well as the empirical search of ATLAS by simply taking into account certain cache parameters [35, 36]. However, when tiling for the different levels of the memory hierarchy, the size of the matrices becomes important. If the matrices are not a multiple of the tile sizes, we need to use padding or cleanup code. With padding, the size of the matrices is increased with additional rows or columns of zeros. Arithmetic operations are usually blindly performed on them. With cleanup, additional code (which is usually suboptimal) is executed to multiply the remainder rows or columns. With recursive layouts, padding is the method usually preferred. Given the large sizes of the second and third level of caches of current machines (6 to 8 MB), padding can represent a significant overhead if the tile sizes are computed without taking into account the matrix sizes. On the other hand, choosing the tile sizes based on the matrix sizes and disregarding the cache sizes will result in poor cache utilization.

In addition, choosing the number of levels of tiling based on the number of caches of the machine may result in slow-downs. In some platforms it is better to use a single level of tiling because additional levels of tiling introduce additional instructions such as branches that may execute slowly.

We compared the MMM routine generated using a classifier learning system with the MMM routine generated by ATLAS when multiplying matrices of sizes 1000 to 5000. Our preliminary results show that the MMM routine generated using the approach we follow in this paper runs always faster than ATLAS in a Sun UltraSparc III by an average 18%. In the case of Intel Pentium Xeon, our routine is almost always faster than ATLAS by an average 5%. However, ATLAS runs on average 14% faster than our routine in Intel Itanium II. Our experiments also show that padding is important to obtain high performance, and we plan to implement more sophisticated padding strategies to improve the performance of the generated library.

The paper is organized as follows. Section 2 revises some of the compiler optimizations that are applied to MMM. Section 3 presents the partition primitives that will be used by the classifier learning system, which is presented in Section 4. Section 5 presents our experimental setup and preliminary results. Section 6 presents related work, and finally, Section 7 concludes.

2 Matrix-Matrix Multiplication

In this Section we present an overview of an automatic tiling and discuss copying and recursive layouts in the context of matrix-matrix multiplication.

A naïve implementation of matrix-matrix multiplication is shown in Figure 1-(a). Usually this code runs slowly because of the poor utilization of cache memories. A transformation used to increase cache locality is loop tiling. This transformation was first introduced by McKellar and Coffman [19] and discussed in the context of compilers by Abu-Sufah [3] and later by Wolfe [32]. Figure 1-(b) shows the code for a tiled matrix-matrix multiplication using a square tile of size $NB \times NB$. This tile size is a parameter that must be chosen to minimize capacity misses. However, when the matrices are large each row (in a row major layout) can be in a different physical page and then TLB misses can occur. This problem can be avoided if the tile selection considers the number of entries in the TLB in conjunction with the cache size [20]. In any case, to reduce conflict and TLB misses, tiling is usually used in combination with copying [16, 28] where the elements of each $NB \times NB$ submatrix are copied into contiguous memory locations.

Tiling has been extensively considered in the literature when applied to a single cache level [9, 16, 21, 25, 35]. However, when tiling for a single level of cache, we do not exploit all the cache levels. For example, Figure 2-(a), shows the order in which the submatrices of A, B and C are accessed when executing the code of Figure 1-(b). Each iteration of the outermost loop (j) will traverse the 16 blocks of matrix A. Unfortunately, if matrix A is large, it will not fit in the second level cache. Therefore each j iteration will have to bring all the A blocks back to the second and first cache level. A solution to this problem is to apply another level of tiling [25, 34].

Suppose that we apply another level of tiling to the code in Figure 1-(b) by adding three additional loops with the same order JIK. The outer loops would operate on

```
for (j = 0; j < M; j++)
  for (i = 0; i < N; i++)
    for (k = 0; k < K; k++)
      C[i][j] += A[i][k] * B[k][j]
```

(a) Naïve implementation

```
for (j = 0; j < M; j+= NB)
  for (i = 0; i < N; i+= NB)
    for (k = 0; k < K; k = +NB)
      for (jj = 0; jj < j + NB; jj++)
        for (ii = 0; ii < i + NB; ii++)
          for (kk = 0; kk < k + NB; kk++)
            C[ii][jj] += A[ii][kk] * B[kk][jj]
```

(b) Tiled implementation

Fig. 1. Matrix Multiplication Code.

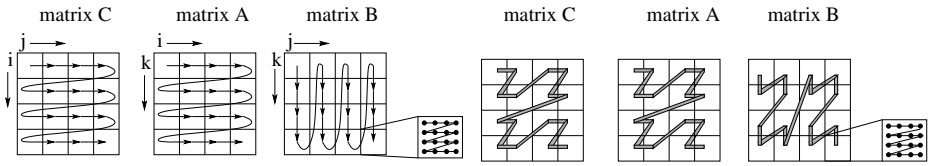

Fig. 2. Memory layouts for tiled matrix-matrix multiplication. (a)- One level of tiling and block data layout. (b)- Two levels of tiling and recursive layout.

blocks consisting of 2×2 tiles so that the blocks of matrix A will be traversed in the order shown in Figure 2-(b). The blocks of the second level of tiling are no longer consecutive in memory and, as a result, these accesses can result in cache conflicts and TLB misses [22]. To avoid this problem, nonlinear array layout or recursive layouts together with tiling have been used [7, 8]. The idea is to copy these blocks into consecutive memory locations. These array layouts are described as based on quadtrees [10] or on space-filling-curves[15, 23, 27]. Instances of this family are familiar in parallel computing under the names Morton ordering and Hilbert Ordering. The layout shown in Figure 2-(b) for matrix A is known as Z-Morton. These recursive layouts were shown to deliver high performance [7, 8], but some considerations need to be taken into account in their implementation:

- These nonlinear layouts can be applied recursively down to the level of individual matrix elements [10]. However, Chatterjee et al. [8] showed that this was counterproductive, and that it is better to follow a recursive layout only until the tile fits in the cache.
- These recursive layouts require that for a matrix of size $M \times N$ and a tile of size $tm \times tn$, the following equations be satisfied: $\frac{M}{tm} = \frac{N}{tn} = 2^d$. Sometimes it is necessary to add padding to the matrix in order to satisfy this equation. The general idea is to select the appropriate tile $tm \times tn$ for the cache of the machine, insert a zero padding and perform the arithmetic operations on the zero padding.

3 Partition Primitives

The library generator used in this study produces a matrix-matrix multiplication (MMM) routine that computes $C = \alpha AB + \beta C$, where A, B and C are matrices of dimensions

$M \times K$, $K \times N$ and $M \times N$ respectively. The generated MMM routine uses multilevel tiling and recursive layouts as discussed above. The routine first copies the original matrices from row or column major layout to the recursive layout. Then, it multiplies the matrices and transforms the resulting C matrix back to the row or column major layout. The copy and multiplication procedures are determined by the number of levels of tiling and tile sizes. These values will be selected using empirical search as discussed below. This Section describes the partition primitives which will be used by the search procedure to determine the best number of levels of tiles and tile sizes for the dimensions of the input matrices and target architecture. Before explaining the primitive partitions, we briefly describe the procedures for copying and padding.

We denote the matrix dimensions at level i as M_i, N_i and K_i, where i ranges from 1 to the $number\_of\_levels\_of\_tiling$. If the matrices at level i are partitioned with factors pm_i, pn_i and pk_i, the dimensions of each submatrix in the next recursion level will be $M_{i-1} = \frac{M_i}{pm_i}$, $N_{i-1} = \frac{N_i}{pn_i}$ and $K_{i-1} = \frac{K_i}{pk_i}$ respectively. The partition factors determine how the sub-blocks must be copied from row (or column) major layout to the recursive layout. An example of these recursive layouts has been shown in Figure 2-(b).

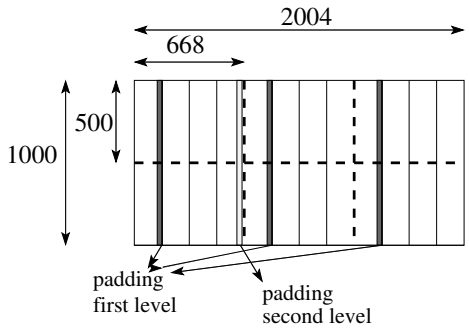

Fig. 3. Example of padding

When the factors in the partition vector are not a divisor of the matrix dimensions we need to use padding. For example suppose A is a matrix of 2000×1000, and we divide it first by (3,2) and then (4,1). Since 3 is not a divisor of 2000, we need to add padding so that we can divide the matrix in exactly 3 pieces. Each resulting submatrix will be of size 667×500. Now, the 667 elements of the X dimension need to be divided by 4. Since 4 is not a divisor of 667, we need to pad each submatrix, and make them to be 668. Thus, we end up with a matrix of size 2004×500. We have 4 additional columns of zeroes which will be blindly multiplied. The example is shown in Figure 3.

Next, we describe the partition primitives that we use in this work.

1. Partition by Block (PB)

This primitive specifies the tile or block size. It has three parameters, which are the block size for each M, N and K dimension. So, consider $M = 100, N = 100, K =$

40. If we want tiles of sizes 50, 50, and 20 for the dimensions M, N, and K, respectively, we would specify this as follows `Partition By Block (50,50,20)`. The `Partition By Block` primitive will compute the partition factors (pm,pn,pk) as follows:

$$pm = \lfloor \frac{m}{bm} \rfloor, pn = \lfloor \frac{n}{bn} \rfloor, pk = \lfloor \frac{k}{bk} \rfloor$$

The `Partition by Block` primitive allows to specify tiles of any size, not only square tiles.

2. `Partition by Size` (PS)
This partition primitive specifies the size of a block and partitions the different dimensions of the matrix until the resulting submatrices are equal or smaller than the size of the specified block. The primitive guarantees that the ratio between the dimensions is kept constant. The primitive allows the specification of the dimensions to be partitioned. It has four parameters. The first three parameters specify if a given dimension M, N and K needs to be partitioned. The fourth parameter specifies the block size. The algorithm used by this primitive is shown below.

```
Input Parameters:
    m,n,k: input matrix dimensions
    muse, nuse, kuse: boolean variables indicating the
                      dimensions to be partitioned
    size: the block size

begin
    maxratio = MIN(m,n,k)/2
    for(ratio=maxratio; ratio >= 2;ratio--) {
        if(muse) tmpm= ⌈m/ratio⌉
        if(nuse) tmpn= ⌈n/ratio⌉
        if(kuse) tmpk= ⌈k/ratio⌉
        tmpsize= tmpm*tmpk + tmpk*tmpn + tmpm*tmpn
        if(tmpsize <= size)
            break;
    }
    if(muse) pm = ratio;
    if(nuse) pn = ratio;
    if(kuse) pk = ratio;
end
```

Notice that most of the previous research on recursive layouts works by dividing each dimension by half. The `Partition by Size` primitive is a generalization of the `divide by half` strategy which can be implemented by setting $muse = nuse = kuse = true$ and $size = \frac{m*k+k*n+m*n}{4}$. In some studies, the recursion is carried down all the way to the individual elements [10, 12]. The work in [12] showed that

this strategy resulted in minimum number of cache misses. Unfortunately in this case minimizing the number of misses does not necessarily results in better performance, because of the additional instructions that need to be executed. In fact, the work by Chaterjee et al. [7, 8] showed that stopping the recursion at tiles of the appropriate size returned better performance. In this paper, when generating the kernel routine for the MMM we will follow the approach of Chatterjee et al.(Section 5).

4 Classifier Learning System

To build a high performance library we need to determine how the input matrices should be partitioned along the M, N and K dimensions. The best partitioning is a function of architectural features such as number of caches and size of each cache and the dimensions of the input matrices. Choosing the correct partition is hard. For some machines, we need to apply a single level of tiling, since the overhead of the additional instructions executed when more levels of tiling are applied results in lower performance. Even when tiling for a single level of cache we need to decide whether to tile for L1 or L2 [2, 35]. When tiling for L2 and L3, it is important to take into account the dimensions of the matrices. Since L2 and L3 tend to be large (sometimes 6 or 8 MB), when a dimension of the matrix is not a multiple of the tile size, the amount of padding can be substantial.

We plan to use the partition primitives described in the previous Section as the building blocks to generate a MMM library. By combining the different primitives and selecting different parameter values, the space of the different algorithms that we can generate is very large. As a result, exhaustive search is unfeasible. Our approach is to use a classifier learning system [6, 24, 31] to search the space of possible algorithms. The main reason to use a classifier learning system is that with this mechanism input characteristics can be used to create a table with the best partitioning parameters. This table can be used at runtime to enable dynamic adaptation.

A classifier system consists of a set of rules. Each rule has two parts, a condition and an action. A condition is a string that encodes certain characteristics of the input, where each element of the string can have three possible values: "0", "1", and "*" (don't care). Similarly, the input characteristics are encoded with a bit string of the same length. If i and c are the input bit string and the condition string respectively, we can define the function $match(i, c)$ as follows:

$$match(i,c) = \begin{cases} true, \ \forall(j) i_j = c_j \vee c_j =' *', where j = length\_of\_the\_bit\_string \\ false, otherwise \end{cases}$$

If there is only one $match(i, c)$ which is $true$, the action corresponding to the condition bit string c is selected. However, for a given input several matches are possible. In this case, we will choose one action among all the rules that match. The mechanism for the selection is explained below (in Section 4.3).

Next we explain how the classifier learning system is tuned for each platform and input

4.1 Representation.

Encoding of the Rule Condition. The input characteristic that will determine the parameter values of the partition primitives is the dimension of the matrices. Thus, we will

encode possible values of the dimensions of the matrices A, B and C in the condition of the rules.

Action of the Rule. The action part will be a list of the partition primitives `partition by size` (PS) or `partition by block` (PB) with their corresponding parameter values. For example, an action will have the shape `(PS param-list (PB param-list))`, where `param-list` is the list of parameters. This action will return a *single function* that will decompose the input matrices of size $M \times N \times K$ into submatrices of size $M' \times N' \times K'$, that result from applying first the PS primitive and them the PB primitive.

Notice that each action, even if it contains several partition primitives correspond to a single level of tiling. To apply several levels of tiling, we can recursively invoke the rule set of the classifier system with the size of the resulting submatrices. The recursion will finish when the number of levels of tiling has already reach the maximum number of levels allowed, or when the size of the submatrices is within a predefined range.

4.2 Training

During the training process we generate matrices of different sizes. Given a training input, we have a match rule set, which are the set of rules where the condition matches the bit string that encodes the input characteristics. We use a XCS classifier learning system as the one in [6, 31]. In this type of classifier systems, each rule has two attributes. The first attribute is the fitness. The fitness is an estimation of the performance of this rule on the inputs that match the associated condition. The second attribute is the accuracy. The accuracy measures the confidence of the fitness attribute in predicting the correct performance.

In our approach we use a multi-step classifier system, since the output of an invocation can be used as the input for the next invocation. This system works as follows. The first time we invoke the rule set with a training input we have a match rule set. All the actions in the matching rules are the set of strategies that can be used to partition the input matrices. During the training process, all the actions in the matching rules are applied. Thus, given an input of size $M \times \mathbb{N} \times K$, the result will be submatrices of sizes $M'_i \times \mathbb{N}'_i \times K'_i$, where $i = 1..number\_of\_matching\_rules$. Each of the $M'_i \times \mathbb{N}'_i \times K'_i$ generated outputs can be used as the input to the next invocation to the learning classifier system. The system, as explained above, will stop when the maximum level of calls is reached or when the size of the submatrices is within a specified range. At the end, we have many different partition strategies, each of them blocking the matrices with tiles of different sizes, and possibly different levels of tiling. We generate the MMM routine for each partition strategy and measure the execution time. Based on the results obtained, we update the fitness and accuracy of all matching rules used to generate each of the MMM routines. The algorithm is shown in Figure 4.

To generate new conditions and actions, transformations such as mutation and crossover applied in genetic algorithms [13, 18] are also used here. MOre details about the XCS classifier learning system that we use in this work can be found in [6, 31].

4.3 Runtime

At the end of the training phase we have a tuned rule set. At runtime, the bit string encoding the input characteristics will be used to extract all the rules whose condition matches the input. Among all these rules, the one selected will depend on a function that rewards low execution time and penalizes low accuracy. The runtime overhead includes the computation of the input bit string, and the scan of the rule set to select the best one.

We train the classifier system to learn a set of rules that cover the space of the possible input parameter values, discover the conditions that better divide the input space and tune the actions to learn the best partition scheme based on the input characteristics.

Multi_Step_Classifier_Learning
Inputs:
M,N,K: dimensions of the input matrices
l: current level of recursion
Outputs:
pm_i, pn_i, pk_i, i=[0..max-num-levels]: partition factors
$exec$: execution time
begin
P= variable that contains the partition factors —pm_i, pn_i, pk_i, i=[0..max-num-levels]
Encode M,N,K into the bit string $\overrightarrow{in}$
$mset = \varnothing$
for each rule r
$\overrightarrow{rcond}$ = condition of r
if $match(\overrightarrow{in}, \overrightarrow{rcond})$
add r to $mset$
while ($mset \neq \varnothing$)
extract r from $mset$
act= action part in r
pm_i, pn_i, pk_i= result of applying act on M, N, K
Update P with the new pm_i, pn_i, pk_i
M', N', K'= result of applying pm_i, pn_i, pk_i on M, N, K
if $notend$ then
call $Multi\_Step\_Classifier\_Learning$ (M', N', K',$l + 1$)
else
Run matrix multiply with M, N, K using P
Measure execution time $exec$
Use $exec$ to update $fitness$ and $accuracy$ of r
return exec
end

Fig. 4. Classifier learning algorithm

5 Experiments

In this section we evaluate our approach of using a classifier learning system to optimize a MMM routine. In Section 5.1 we discuss the environmental setup that we use for the evaluation and in Section 5.2 we present performance results.

5.1 Environmental Setup

We evaluated our approach on three different platforms: Sun UltraSparc III, Intel Itanium 2, and Intel Xeon. Table 1 lists for each platform the main architectural parameters, the operating system, the compiler and the compiler options used for the experiments.

	Sun	Intel	Intel
CPU	UltraSparcIII	Itanium 2	P4 Intel Xeon
Frequency	750MHz	1.5GHz	3GHz
L1d/L1i Cache	64KB/32KB	16KB/16KB	8KB/12KB (1)
L2 Cache	1MB	256KB (2)	512KB
Memory	4GB	8GB	2GB
OS	SunOS5.8	RedHat7.2	RedHat3.2.3
Compiler	Workshop cc 5.0	gcc3.3.2	gcc3.4.1
Options	-native -xO5	-O3	-O3

Table 1. Test Platforms. (1) Intel Xeon has a 8KB trace cache instead of a L1 instruction cache. (2) Intel Itanium2 has a L3 cache of 6MB.

To generate the MMM library we used the classifier learning system. We trained the classifier with the algorithm of Figure 4. The classifier determines the number of levels of tiling and the tile size for each matrix size. For the implementation of the MMM at the last level of tiling we used the kernel generated by ATLAS. ATLAS generates a MMM routine and uses empirical search to look for the best parameter values of certain compiler transformations such as tile size, loop unrolling and software pipelining [30, 35, 36]. The kernel in ATLAS produces code for a MMM routine with a single level of tiling and square tiles. Thus, in our MMM library the submatrices in the last level of tiling must also be square. We allow these submatrices to be in the range of 40 - 120, since this range cover most of the different values that ATLAS finds for current platforms [36]. ATLAS generates a single MMM routine and searches for the tile size that obtains the best performance results. In our system, the tile size of the last level is determined by the classifier learning system, but we use ATLAS to search for the rest of the other parameters for each tile size in the range 40 - 120. We limited the maximum number of levels of tiling to be 3, since current architectures have three or less caches, and our experiments showed that increasing the level of tiling beyond 3 resulted in less performance. Apart from this, after we determine the partitioning strategy, we need to copy the tiles to the corresponding recursive layout. In this work we use the Z-Morton layout, although in a longer study we could also search for the best layout. When the matrix is not a multiple of the tiling we insert padding, as shown in Figure 3. Padding can also be necessary to obtain a square tile at the last level of tiling.

To encode the size of the matrices, we used 13 bits per dimension. Since we have 3 dimensions $M \times N \times K$, we used a total of 39 bits. Initially we generated 1000 rules, and we randomly generated the condition and the action part of each rule. For the training we randomly generated matrices whose sizes were between 1000 and 5000. We

did not specify any condition to end the training process. Instead, we let the training run for a certain amount of time. In the experiments reported here, we let it run for 1 week.

We compare the MMM routine generated by our classifier learning system with three different approaches:

- L1, where the MMM routine has a single level of tiling.
- L2, where the MMM routine has two levels of tiling.
- ATLAS.

To make a fair comparison with L1 and L2 approaches we used ATLAS to generate the kernel of the MMM routine. In both cases we used the same copying strategy and padding as the one used in the MMM routine generated using the classifier. For the L1 approach we used the tile size that ATLAS found to be the best. For the L2 approach we used the value found by ATLAS for the first level of tiling. For the second level of tiling we chose the size so that $Tile2 = K \times Tile1$. We selected K so that $Tile2$ is multiple of $Tile1$, and smaller than the value that results from resolving the inequality $3{*}Tile2^2 \leq CacheSize$. The exception is Sun UltraSparc III. This machine has a large L2 cache (1 MB) and selecting the $Tile2$ using the previous formula resulted in low performance, since padding represented a large overhead in some cases. We decided to select for the Sun UltraSparc a tile of size 1/3 of the computed value using the previous formulas. Table 2 shows the values used for each $Tile1$ and $Tile2$. In both L1 and L2 we allowed the $Tile1$ to vary within the value reported in the Table and $+/-10$. We varied the size of the $Tile1$ based on the matrix size to minimize the amount of padding.

	UltraSparcIII	Itanium 2	P4 Intel Xeon
L1_Tile	68	120	60
L2_Tile	380	240	240

Table 2. Tile Sizes.

For ATLAS we used the code produced by the ATLAS Code Generator using empirical search. ATLAS can also use hand tuned BLAS routines. When ATLAS is installed these hand-coded routines are also executed and evaluated. However, since in this work we are only interested on the comparison on the MMM routine generated by ATLAS, we only used the code generator, without hand-coded code. Notice, that ATLAS can have a L2 Cache Blocking parameter by setting a variable called CacheEdge. For the ATLAS experiments, we set this variable to the appropriate value as reported in [1].

5.2 Experimental Results

Figure 5 presents the performance results of the four MMM routines described in the previous Section: L1, L2, Classifier and ATLAS. For the experiments we multiplied square matrices whose sizes vary from 1000 to 5000, in steps of 100.

The results vary from platform to platform. In the case of the Sun UltraSparc, Classifier is always the best. For this platform L2 is also better than ATLAS and L1. For Itanium 2, the code generated by ATLAS performs better than any of the other routines.

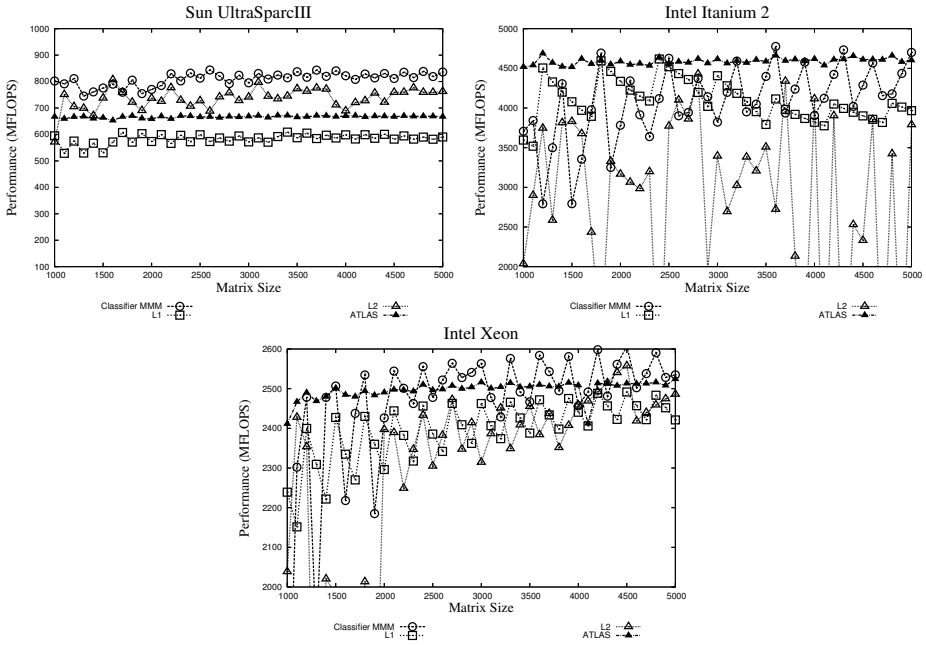

Fig. 5. Performance Results

Only in a few points the code generated by the Classifier is equal or better. For Intel Xeon, the code generated by the Classifier is usually the fastest, followed by that of ATLAS.

It has been stated [26] that tiling for L1 was enough and that multi-level tiling was not necessary. However, our results for Sun UltraSparc III show that multi-level tiling can improve performance over one level of tiling, since L2 and Classifier are always the best approaches for this platform. For the other two platforms it is not clear if multilevel tiling is better.

The performance results for the Intel platforms Itanium 2 and Xeon shows high variability in performance for the code generated by Classifier, L1 and L2. Since these 3 approaches use padding when the dimensions of the matrices are not multiple of the tile sizes, while ATLAS (whose performance is very stable) uses cleanup code, we think that the variability is due to the fact that the amount of padding changes for the different matrices being multiplied. We need to conduct further experiments to verify this. Also, in the future we plan to study different strategies to pad the matrices more efficiently. For example, we can concentrate all the padding at the end of the matrix, instead of distribute it in each tile, as we have done in the routine in this paper. We will also study the possibility of combining cleanup code with recursive layouts. If we find out that performance is highly dependent on the padding or clean up strategies, we can also search in this space.

Overall, our results, still preliminary, show that the MMM routine generated using the approach we follow in this paper runs always faster than the code generated by ATLAS in a Sun UltraSparc III by an average of 18%. In the case of an Intel Pentium Xeon, our routine runs almost always faster than ATLAS by an average of 5%. However, ATLAS runs 14% faster than our routine in Intel Itanium II. In the future, we will also add more platforms to this study.

6 Related Work

As mentioned in Section 2 the use of loop tiling to increase cache locality has been extensively studied in the literature. Lam et al. [16], Coleman and McKinley [9] and others have developed algorithms to compute the optimal tile sizes when a single level of tiling is applied. Lam et al. [16] present an algorithm that selects the largest square tile that does not cause self interference misses. Coleman and McKinley [9]'s technique uses the Euclidean G.C.D. to generate a set of tiles without self-interference misses and from those tiles select the one that maximizes cache utilization and minimizes cross-interference misses.

Recursive matrix multiplication has been studied by Frens and Wise [10], Gustavson [14], Chatterjee et al. [8] and Frigo et al [12]. Chaterjee et al [8] shows that recursive layouts can significantly outperform traditional layouts for standard matrix-matrix multiplication. They also show that stopping the recursion when the tile fits into the cache results in better performance because it avoid some of the overheads due to recursive calls. Our approach is different than that of Chaterjee et al. [8]. We use machine learning techniques to search for the appropriate number of levels of tiling and tile sizes based on the dimensions of the input matrices and the architectural platform.

The ATLAS [30] generator uses empirical search to find the optimal tile size for a single level of tiling. However, the ATLAS' search problem is simpler than that of our system because ATLAS only considers the case where the same tile size is used for all the matrix sizes.

Finally, the approach that we present in this paper is also related to the problem of selecting from a set of candidate algorithms the one that performs best for a particular input and system. Systems that follow this approach are described by Li et al. [17, 18], Brewer [5] and Thomas et al. [29]. In [17] we used the Winnow algorithm to select from three sequential sorting algorithms the one that performs best for a target system based on the entropy and number of keys of the input data, while in [18] we used a learning classifier system to generate composite sorting algorithms. Brewer [5] and Thomas et al. [29] use a framework for algorithm selection to generate parallel operations that adapt to the input and platform. In particular Thomas et al. [29] describe a general framework that can be easily extended with new operations and different empirical learning approaches.

7 Conclusions

In this paper we have generated a MMM routine using a classifier learning system. The MMM routine generated with our classifier learning system uses different levels of tiling and tile sizes based on the dimensions of the matrices and the architectural features of the machine where it is installed.

We compared the MMM routine generated using a classifier learning system with the MMM routine generated by ATLAS when multiplying matrices of sizes 1000 to 5000. Our preliminary results show that the MMM routine generated using the classifier runs always faster than ATLAS in a Sun UltraSparc III by an average of 18%. In the case of an Intel Pentium Xeon, our routine runs almost always faster than ATLAS by an average of 5%. However, ATLAS runs on average 14% faster than our routine in Intel Itanium II. Our experiments also show that padding is important to obtain high performance, and we plan to implement more sophisticated padding strategies to improve the performance of the generated library.

References

1. ATLAS home page. [Online]. http://math-atlas.sourceforge.net/errata.html#tuneCE.
2. ATLAS home page. [Online]. http://math-atlas.sourceforge.net/faq.html#NB80.
3. W. Abu-Sufah, D. Kuck, and D. Lawrie. On the Performance Enhancememt of Paging Systems through Program Analysis and Transformations. *IEEE Transactions on Computers*, 30(5):341–356, May 1981.
4. J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing Matrix Multiply using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *Proc.of the 11th ACM International Conference on Supercomputing (ICS)*, July 1997.
5. E. A. Brewer. High-level Optimization via Automated Statistical Modeling. In *Proc. of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 80–91, New York, NY, USA, 1995. ACM Press.
6. M. V. Butz and S. W. Wilson. An Algorithmic Description of XCS. *Lecture Notes in Computer Science*, 1996:253–272, 2001.
7. S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear Array Layouts for Hierarchical Memory Systems. In *International Conference on Supercomputing*, pages 444–453, 1999.
8. S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thotterhodi. Recursive array layouts and fast matrix multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 13:1105–1123, 2002.
9. S. Coleman and K. s. McKinley. Tile Selection Using Cache Organization and Data Layout. In *Proc. of Int. Conference Programming Language Design and Implementation*, pages 279–290, June 1995.
10. J. Frens and D. Wise. Auto-blocking Matrix-Multiplication or Tracking BLAS3 Performance with Source Code. In *Proc. of the Intenational Symp. on Principles and Practice of Parallel programming (PPoPP)*, pages 206–216, June 1997.
11. M. Frigo. A Fast Fourier Transform Compiler. In *Proc. of Programing Language Design and Implementation*, 1999.
12. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. In *Proc. of the Intenational Symp. on Foundations of Computer Science (FOCS)*, October 1999.
13. D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.
14. F. G. Gustavson. Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.
15. D. Hilbert. Über Stetige Abbildung einer Linie auf ein Flächenstrück. *Mathematische Annalen*, 38:459–60, 1891.
16. M. Lam, E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proc. of the Int. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 63–74, October 1991.

17. X. Li, M. J. Garzarán, and D. Padua. A Dynamically Tuned Sorting Library. In *In Proc. of the Int. Symp. on Code Generation and Optimization*, pages 111–124, 2004.
18. X. Li, M. J. Garzarán, and D. Padua. Optimizing Sorting with Genetic Algorithms. In *In Proc. of the Int. Symp. on Code Generation and Optimization*, pages 99–110, March 2005.
19. A. McKellar and E. Coffman. Organizing Matrices and Matrix Operations for Paged Memory Systems. *In Communications of the ACM*, 12(3):153–165, March 1969.
20. N. Mitchell, K. Hogstedt, L. Carter, and J. Ferrante. Quantifying the Multi-Level Nature of Tiling Interactions. *Int. Journal of Parallel Programming*, 26(6):641–670, June 1998.
21. P. Panda, H. Nakamura, N. Dutt, and A. Nicolau. Augmenting Loop Tiling with Data Alignment for Improved Cache Performance. *IEEE Trans. on Computers*, 48(2):142–149, February 1999.
22. N. Park, B. Hong, and V. Prasanna. Tiling, Block Data Layout, and Memory Hierarchy Performance. *IEEE Trans. on Parallel and Distributed Systems*, 14(7):640–654, July 2003.
23. G. Peano. Sur Une Curbe qui Remplit Toute une Aire Plaine. *Mathematische Annalen*, 36:157–160, 1890.
24. W. S. Pier Luca Lanzi and S. W. Wilson. *Learning Classifier Systems, From Foundations to Applications*. Springer-Verlag, 2000.
25. G. Rivera and C. Tseng. Data Transformations for Eliminating conflict Misses. In *Proc. of Int. Conference Programming Language Design and Implementation*, pages 38–49, June 1998.
26. G. Rivera and C. Tseng. Locality Optimizations for Multi-Level Caches. In *Proc. of IEEE Supercomputing*, November 1999.
27. H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994.
28. O. Temam, E. Granston, and W. Jalby. To Copy or Not to Copy: A Compile–Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts. In *Proc. of the ACM/IEEE Supercomputing Conference*, November 1993.
29. N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A Framework for Adaptive Algorithm Selection in STAPL. In *Proc. of Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 277–288, New York, NY, USA, 2005. ACM Press.
30. R. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Sofware and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, 2001.
31. S. W. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.
32. M. Wolfe. Iteration Space Tiling for Memory Hierarchies. In *Third SIAM Conference on Parallel Processing for Scientific Computing*, December 1987.
33. J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A Language and a Compiler for DSP Algorithms. In *Proc. of the International Conference on Programming Language Design and Implementation*, pages 298–308, 2001.
34. Q. Yi, V. Adve, and K. Kennedy. Transforming Loops To Recursion for Multi-Level Memory Hierarchies. In *Proc. of the Int. Conf. on Programming Language Design and Implementation (PLDI)*, pages 169–181, June 2000.
35. K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzarán, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A Comparison of Empirical and Model-driven Optimization. In *Proc. of Programing Language Design and Implementation*, pages 63–76, June 2003.
36. K. Yotov, X. Li, G. Ren, M. J. Garzarán, D. Padua, K. Pingali, and P. Stodghill. Is Search Really Necessary to Generate a High Performance Blas? *In Proc. of the IEEE, special issue on Program Generation, Optimization, and Platform Adaptation*, 23:358–386, February 2005.