Capturing the Expert: Generating Fast Matrix-Multiply Kernels with Spiral

Richard Veras and Franz Franchetti

Carnegie Mellon University,
5000 Forbes Avenue, Pittsburgh, PA

Abstract. Matrix-Matrix Multiplication (*MMM*) is a fundamental operation in scientific computing. Achieving the floating point peak with this operation requires expert knowledge of linear algebra and computer architecture to craft a tuned implementation, for a given microarchitecture. The expert follows a mechanical process for implementing *MMM*, by deriving the algorithm models from the literature. Then, by hand, applying optimizations which are well suited for that architecture. Lastly, the experts codes that implementation at the assembly level. In this paper, we argue that this process is mechanical and can be captured in an autotuning and program generation system such as *Spiral*. We then show that given this machinery, *Spiral* can produce code for large size *MMM* implementations that are competitive with hand tuned code.

1 Introduction

Implementing a high performance implementation of *MMM* is difficult. A great body of work provides the mathematical machinery for determining the optimal blocking strategies analytically [2, 1]. However, once this is determined, there is still the issue of mapping the implementation to the hardware. There are microarchitecture specific issues that must be addressed in order to reach this level of high performance.

The Compiler: Compilers fall short of reaching the peak performance for *MMM*. If a programmer provides a straightforward implementation of *MMM* with a triple nested loop (Fig 3.1), the code generated by the compiler will not achieve the same high performance as the code an expert produces. One reason for this, is that compilers are general purpose and trade off potential performance gains, in the code, for shorter compile times. As a result, the compiler contains optimizations which are well suited for the general case, but are not necessarily the optimizations needed for the specific case of *MMM*.

The Expert: The domain expert acts as a specialized compiler. She uses domain knowledge to optimize for both the operation and the hardware.. How do we automate the expert? How do we capture this knowledge in an auto-generation framework? The goal of this paper is to formalize the necessary components that an expert would use, for generating high performance implementations *MMM*, in a rule based framework for automatic program generation.

In this paper, we exploit the fact that the expert uses a mechanical process for creating high performance *MMM* kernels and we formalize this process as a

set of rules. We then feed these rules through the *Spiral* framework [7], which in turn, produces an autotuned high performance implementation of *MMM*. This implementation is comparable performance to an expert produced kernel. We base our work on the success of existing Automatic Program Generation Systems which replaced expert programmers in the fields of Digital Signal Processing (*DSP*) and Sparse Matrix-Vector Multiplication.

2 Related Work

The goal of our project is to capture the expert knowledge, behind implementing a high performance *MMM*, as rules in a rule in a system like *Spiral*, which in turn can generate and tune an implementation from those rules. The *Spiral* project [7] automates the expert programmer in the *DSP* field. They do this through a layering of Domain Specific Languages (*DSL*) and a database of rules for each *DSL* which captures the knowledge of the domain experts. In the case of *DSP*, they show that high performance code can be automatically generated for fixed and general sized operations. In [5], the authors extended *Spiral* for operations, like *MMM*, through a new language called OL. Their generated *MMM* kernels performed significantly faster than those produced by optimizing compilers, but still fell short of the expert programmer. The authors of [3] improved upon the ideas of OL and produced a domain specific language for linear algebra inside *Spiral*. Their work targets small vectorized kernels whereas our project focuses on large kernels typically associated with Level-3 *BLAS* routines.

The *ATLAS* project [10], provides a framework for empirically determining the blocking dimensions for cache and registers for *MMM*. We differ from their project because we are generating code from rules given as inputs to a system, as opposed to using a system that is hard coded for a certain class of problems. The authors of [1] demonstrated that the algorithm implemented by *ATLAS* was sub-optimal and presented a novel blocking strategy that achieves – both in theory and practice – near peak performance. This work was extended by the BLIS Framework [2] for all Level-3 *BLAS*, by providing a software architecture for minimizing the actual amount of code that an expert needs to produce for a given architecture. The AUGEMM project [4] took this last piece of the expert and automated the process of generating the kernels via templates in a *MMM* specific framework. We differ from their project because we generate our implementation from high level rules that are selected via search, instead of templates. The rules that we will describe in the next section capture the insight from these works. In the *DxT* project [8], the authors propose a system for performing high level optimizations of linear algebra algorithms. Our work differs from theirs, because we are trying to define a language that allows the description of such transformations. Our purpose is to capture the expert knowledge that is needed to produce high performance large size *MMM* kernels in the general rule based framework *Spiral*, with the hopes of exploiting this knowledge in problems outside of the domain of linear algebra.

3 A New Spiral Operator Language

In this section, We illustrate our language for capturing the knowledge used by the expert in developing a high performance implementation of matrix-matrix

multiplication (*MMM*). A matrix matrix multiplication can be mathematically described as $C = AB$ where A, B and C are general matrices of conformal sizes. In our operator language, the operation is $\text{MMM}_{m,k,n}$ where the subscripts describe the sizes of the inputs (i.e. A is $m \times k$ and B is a $k \times n$ matrix.

3.1 A Naïve MMM Implementation

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

$$
\begin{aligned}
&\text{MMM}_{M,K,N} \rightarrow \\
&\quad \text{PtA}(h_{0,[1]}^{N\rightarrow N} \otimes h_{0,[1]}^{M\rightarrow M}, \\
&\quad\quad \text{Pt}(h_{0,[1]}^{N\rightarrow N} \otimes h_{0,[1,1]}^{1\rightarrow M}, \\
&\quad\quad\quad \text{Pt}(h_{0,[M,1]}^{1\rightarrow N} \otimes h_{0,[1]}^{1\rightarrow 1}, \\
&\quad\quad\quad\quad P_1 \\
&\quad\quad\quad\quad h_{0,[M]}^{1\rightarrow 1} \otimes h_{0,[1]}^{1\rightarrow 1} \times h_{0,[K,1]}^{1\rightarrow N} \otimes h_{0,[1]}^{1\rightarrow 1}), \\
&\quad\quad\quad h_{0,[1]}^{1\rightarrow 1} \otimes h_{0,[1,1]}^{1\rightarrow M} \times h_{0,[K]}^{N\rightarrow N} \otimes h_{0,[1]}^{1\rightarrow 1}), \\
&\quad\quad h_{0,[M,1]}^{1\rightarrow K} \otimes h_{0,[1]}^{M\rightarrow M} \times h_{0,[1]}^{N\rightarrow N} \otimes h_{0,[1,1]}^{1\rightarrow K})
\end{aligned}
,
$$

Fig. 1. On the left we have a naïve implementation of *MMM* using a triply nested loop. The inputs, A and B, are stored in column major ordering. On the right we have a representation of that implementation of *MMM* in our language.

At the heart of our representation in Fig 3.1 is the operator $\text{Pt}(s, A, g)$ which is our non-overlapping loop operator (PtA is our overlapping loop, which accumulates the output). The result of each iteration of computation that is performed is written to a unique spot in memory. This operator is parameterized by three major components. The first is an *index mapping* function, g, for describing how elements are read from memory at each iteration. The second parameter is an operation, A, that is performed at each iteration. And the last parameter, is a second index mapping function, s, which describes how the result of A will be written to memory.

For s and g, we use a *stride* operator [6] which captures the structure of the input or output. For example, $h_{0,[1,1]}^{n_b\rightarrow N} \otimes h_{0,[1]}^{M\rightarrow M}$ describes an $M \times N$ matrix which is linearized in memory in column major ordering (M is the leading dimension), and we are selecting out $m_b \times N$ adjacent sub-blocks from this matrix. The key point behind these stride functions is that they capture the structure of the dataset in memory.

3.2 What an Expert Really Does

The Goto/*BLIS* algorithm [1, 2], which we will implement in our language, achieves high performance by exploiting several key factors: 1. blocking for last level cache reuse, 2. packing to minimize Translation Lookahead Buffer (TLB) misses, 3. data layout transformations to minimize Level 1 cache misses, 4. and efficient use of vector instructions [4] . We will take a step by step approach to show how we capture these insights in our language.

Blocking *Blocking*, or tiling, is a method for maximizing cache reuse by operating on blocks of an input rather than operating on scalar elements of the

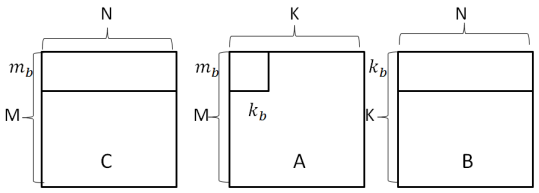

Fig. 2. The Goto/*BLIS* blocking structure.

input. Because *MMM* is a computationally bounded problem, we want to insure that we maximize the cache reuse of our inputs so our performance is not restricted by the cost of moving data from memory into the processor [1]. We achieve blocking by adding an additional nesting of loop operators, Pt, around a smaller MMM and by using the stride operators to capture how the sub-blocks fit inside of the input matrices.

$$
\begin{aligned}
&\mathrm{MMM}_{M,K,N} \rightarrow \\
&\quad \mathrm{PtA}(h^{N\rightarrow N}_{0,[1]} \otimes h^{M\rightarrow M}_{0,[1]}, \\
&\quad\quad \mathrm{Pt}(h^{N\rightarrow N}_{0,[1]} \otimes h^{m_b\rightarrow M}_{0,[1,1]}, \\
&\quad\quad\quad \mathrm{Pt}(h^{n_r\rightarrow N}_{0,[M/m_b,1]} \otimes h^{m_b\rightarrow m_b}_{0,[1]}, \\
&\quad\quad\quad\quad \mathrm{Pt}(h^{n_r\rightarrow n_r}_{0,[M/m_b]} \otimes h^{m_r\rightarrow m_b}_{0,[1,1]}, \\
&\quad\quad\quad\quad\quad \mathrm{MMM}_{m_r,k_b,n_r} \\
&\quad\quad\quad\quad\quad h^{k_b\rightarrow k_b}_{0,[M/m_b]} \otimes h^{m_r\rightarrow m_b}_{0,[1,1]} \times h^{n_r\rightarrow n_r}_{0,[K/k_b]} \otimes h^{k_b\rightarrow k_b}_{0,[1]}), \\
&\quad\quad\quad\quad h^{k_b\rightarrow k_b}_{0,[M/m_b]} \otimes h^{m_b\rightarrow m_b}_{0,[1]} \times h^{n_r\rightarrow N}_{0,[K/k_b,1]} \otimes h^{k_b\rightarrow k_b}_{0,[1]}), \\
&\quad\quad\quad h^{k_b\rightarrow k_b}_{0,[1]} \otimes h^{m_b\rightarrow M}_{0,[1,1]} \times h^{N\rightarrow N}_{0,[K/k_b]} \otimes h^{k_b\rightarrow k_b}_{0,[1]}), \\
&\quad\quad h^{k_b\rightarrow K}_{0,[1,1]} \otimes h^{M\rightarrow M}_{0,[1]} \times h^{N\rightarrow N}_{0,[1]} \otimes h^{k_b\rightarrow K}_{0,[1,1]})
\end{aligned}
$$

Using the formula above, we can express the blocking strategy used in the Goto/*BLIS* approach. For the sake of consistency we follow the same variable naming conventions for tile sizes (k_b, m_b, n_r, m_r) as those used in [1]. The essence of this formula, is that we can use the stride functions to express the data layout of the blocks in memory and use the loop operators to manipulate those blocks.

Packing Blocks Into Contiguous Buffers *Packing* takes blocking one step further by copying the sub-blocks into a contiguous buffer in memory. By placing the working set in a contiguous buffer, fewer TLB entries – a cache for address translation – are needed to address the working set and therefore fewer costly TLB misses occur which can severely hinder performance [1]. If we want to pack a $m_b \times k_b$ subblock of matrix A which is $M \times K$ we can use the following formula:

$$
\begin{aligned}
&\mathrm{PackA}_{M,m_b,k_b} \rightarrow \\
&\quad\quad \mathrm{Pt}(h^{1\rightarrow k_b}_{0,[1,1]} \otimes h^{m_b\rightarrow m_b}_{0,[1]}, \\
&\quad\quad\quad \mathrm{Pt}(h^{1\rightarrow 1}_{0,[1]} \otimes h^{1\rightarrow m_b}_{0,[1,1]}, \\
&\quad\quad\quad\quad I_1, \\
&\quad\quad\quad\quad h^{1\rightarrow 1}_{0,[M/m_b]} \otimes h^{1\rightarrow m_b}_{0,[1,1]}), \\
&\quad\quad\quad h^{1\rightarrow k_b}_{0,[M/m_b,1]} \otimes h^{m_b\rightarrow m_b}_{0,[1]})
\end{aligned}
$$

The formula above uses two nested loop operators to copy a sub-block of A into a contiguous buffer. Following this same construction we can pack an $k_b \times N$ subblock of matrix B which has the dimensions $K \times N$. We can then construct a new rule:

$$\begin{aligned}\text{MMM}_{m_b,k_b,N} &\rightarrow \text{MMM}_{m_b,k_b,N} \circ (\text{PackA}_{M,m_b,k_b} \times I_{Nk_b}) \\ &\rightarrow \text{MMM}_{m_b,k_b,N} \circ (I_{m_b k_b} \times \text{PackB}_{K,k_b,N})\end{aligned}$$

If you read these rules from right to left they express the behavior of packing a sub-block of one input (PackA and PackB) while leaving the other input unmodified ($I_{m_b k_b}$). After that they are passed onto the next step of computation, the MMM operator.

Data Layout Transformation of Packed Blocks A *data layout transformation* is a reindexing of the inputs such that every memory access to that input is performed in unit stride. We can improves temporal locality in the cache by rearranging the elements of our sub-blocks of A and B as we pack them. The following formula captures the packing and data layout transformation where we read our input with one data layout and write our output in a different layout:

$$\begin{aligned}\text{PackA}_{M,m_b,k_b} \rightarrow \\ &\text{Pt}(h_{0,[1,1]}^{1\rightarrow m_b/m_r} \otimes h_{0,[1]}^{k_b\rightarrow k_b} \otimes h_{0,[1]}^{m_r\rightarrow m_r}, \\ &\quad \text{Pt}(h_{0,[1]}^{1\rightarrow 1} \otimes h_{0,[1,1]}^{1\rightarrow k_b} \otimes h_{0,[1]}^{m_r\rightarrow m_r}, \\ &\quad\quad \text{Pt}(h_{0,[1]}^{1\rightarrow 1} \otimes h_{0,[1]}^{1\rightarrow 1} \otimes h_{0,[1]}^{m_r\rightarrow 1}, \\ &\quad\quad\quad I_1, \\ &\quad\quad\quad h_{0,[M/m_r]}^{1\rightarrow 1} \otimes h_{0,[1,1]}^{1\rightarrow m_r}), \\ &\quad\quad h_{0,[M/m_r,1]}^{1\rightarrow k_b} \otimes h_{0,[1]}^{m_r\rightarrow m_r}), \\ &\quad h_{0,[M/m_b]}^{k_b\rightarrow k_b} \otimes h_{0,[1,1]}^{m_r\rightarrow m_b})\end{aligned}$$

We achieve our data layout transformation by modifying our method for packing through the introduction of a new index striding function for the sub-blocks. For example, our sub-block A originally has the initial layout described by this index mapping function, $h_{0,[1]}^{k_b\rightarrow k_b} \otimes h_{0,[1]}^{m_b\rightarrow m_b}$. Our goal is to reorder the elements into this index mapping function $h_{0,[1,1]}^{1\rightarrow m_b/m_r} \otimes h_{0,[1]}^{k_b\rightarrow k_b} \otimes h_{0,[1]}^{m_r\rightarrow m_r}$ which will allow us to access the elements in unit stride, in the innermost loop of our kernel Fig 3.1.

Minimal Instruction SIMD Vector Broadcast In order to achieve high performance on a processor that allows vector operations, we must use those vector operations. However, in the innermost loop of our *MMM* we need to multiply every element from the A matrix with every element from the B matrix. If we perform this task with vector operations, we would need a way to broadcast each element of one of the inputs into a vector. On some architectures this is sufficient, but on others this operation is expensive. An alternative to the broadcast instruction is to reorder the elements in the vector register for one input, in such a way that we are ultimately able to multiply each element from

one input with the other. This is discussed in further detail in [4]. We capture this optimization through the addition of two special permutation operations: a permuted broadcast on a vector of length v, Y_v and a reordering operator that reverses the permutation, U_v.

$$
\begin{aligned}
\mathrm{MMM}_{M,K,N} \rightarrow\ & U_v \circ \\
& \mathrm{PtA}(h^{N\rightarrow N}_{0,[1]} \otimes h^{M\rightarrow M}_{0,[1]}, \\
& \quad \mathrm{Pt}(h^{N\rightarrow N}_{0,[1]} \otimes h^{1\rightarrow M}_{0,[1,1]}, \\
& \quad\quad \mathrm{Pt}(h^{v\rightarrow N}_{0,[M,1]} \otimes h^{1\rightarrow 1}_{0,[1]}, \\
& \quad\quad\quad P_v \circ (Y_v \times I_v) \\
& \quad\quad\quad h^{1\rightarrow 1}_{0,[M]} \otimes h^{1\rightarrow 1}_{0,[1]} \times h^{v\rightarrow N}_{0,[K,1]} \otimes h^{1\rightarrow 1}_{0,[1]}), \\
& \quad\quad h^{1\rightarrow 1}_{0,[1]} \otimes h^{1\rightarrow M}_{0,[1,1]} \times h^{N\rightarrow N}_{0,[K]} \otimes h^{1\rightarrow 1}_{0,[1]}), \\
& \quad h^{1\rightarrow K}_{0,[M]} \otimes h^{M\rightarrow M}_{0,[1]} \times h^{N\rightarrow N}_{0,[1]} \otimes h^{1\rightarrow K}_{0,[1,1]})
\end{aligned}
,
$$

The permuted broadcast, Y_v, is applied at every iteration of the innermost loop, which would be the case with a normal vector broadcast, however the reordering operator, U_v, is applied outside of the inner loop. Depending on the architecture, the implementation of these operators may vary, however, they capture the essence that one of the inputs must be broadcasted and that the order in which it happens need not be preserved, because the results will be reordered before they are written to memory.

4 Code Generation

Spiral [7] is a rule based system for generating high performance code. Given a set of breakdown rules, which we described in the previous section, *Spiral* will recursively apply those rules until it has a family of fully expanded rule trees, which describe an implementation. Then, given a set of translation rules for converting those rule trees into code (Table 4). Lastly, via exhaustive empirical search, *Spiral* will find the best implementation in its search space.

Operator	**C Code Translation**
Code $(A, \mathbf{y}, \mathbf{x}, [i])$	`A(y,x,[i]);`
Code $(A \circ B, \mathbf{z}, \mathbf{x}, [i])$	Code $(A, \mathbf{z}, \mathbf{y}, [i])$; Code $(B, \mathbf{y}, \mathbf{x}, [i])$;
Code $(A \times B, \mathbf{y}, \mathbf{x}, [i])$	Code $(A, \mathbf{y_0}, \mathbf{x_0}, [i])$; Code $(B, \mathbf{y_1}, \mathbf{x_1}, [i])$;
Code $(I_1, \mathbf{y}, \mathbf{x}, [i])$	`*y = *x;`
Code $(P_1, \mathbf{y}, \mathbf{x}, [i])$	`*y = *x0 * *x1;`
Code $(Pt(s, A, g), \mathbf{y}, \mathbf{x}, [i])$	`for( j=1; j < rng(s)/dmn(s); ++j)` Code $(s, \hat{\mathbf{y}}, \mathbf{y}, [j])$ Code $(g, \hat{\mathbf{x}}, \mathbf{x}, [j])$ Code $(A, \hat{\mathbf{y}}, \hat{\mathbf{x}}, [j])$
Code $(f, \mathbf{y}, \mathbf{x}, [j])$	`y = &x[f(j)];`

Table 1. Here we have listed the translations between our scalar operators and C code. Given a formula B, we recursively apply Code $(B, \mathbf{y}, \mathbf{x}, [j])$, until we have a final C coded implementation. The $\mathbf{y}$ and $\mathbf{x}$ refer to our output and input, and the variable j refers to loop variable in the current scope.

5 Results

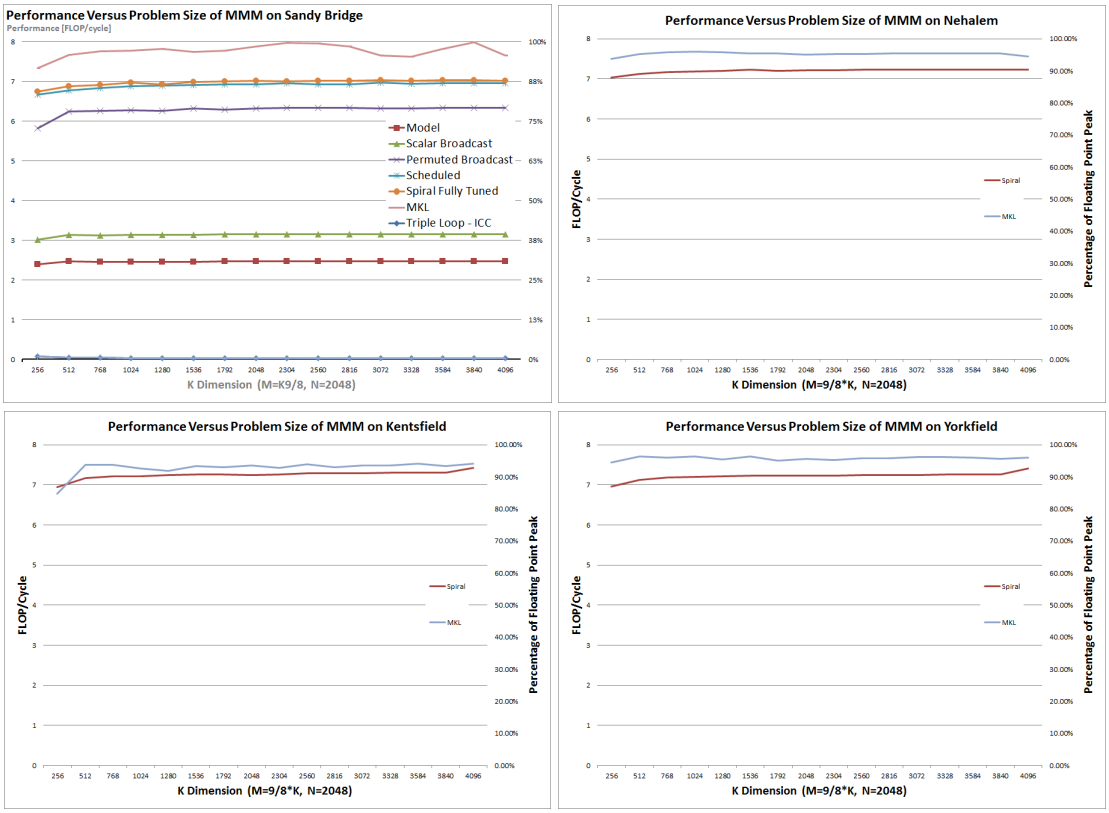

Fig. 3. Performance comparison of *Spiral* generated code versus expert written code on various architectures. In each of the charts, the top line represents that maximum floating point peak. **Top Left:** In this chart we show how the addition of a few low level transformations at code generation time can add the additional performance. Additionally, we show the performance of a triply nested implementation of *MMM*, described in Section 3, compiled with the Intel Compiler.

In this section we use *Spiral* to generate and optimize our implementation of *MMM* from the rules that we have defined. We then compare the performance achieved by the best generated code, found by *Spiral*'s exhaustive search, against Intel's Math Kernel Libraries (MKL) on several Intel architectures. In order to achieve high performance, there are several additional optimizations that must be performed when *Spiral* is generating the code. This includes: *memory prefetching* to minimize the number of last level cache misses, *software pipelining* [9] to hide the latency of bringing data elements from the cache, and *loop unrolling* which reduces the number of branch instructions and simplifies address computation. For our experiments, we compare the performance of our generated code

against Intel MKL on the following microarchitectures: Intel Sandy Bridge, Nehalem, Kentsfield and Yorkfield. On the Yorkfield and Nehalem the performance of our code is within 5% of the expert code and on the other architectures it is within 10%. Further optimizations are necessary in order to match the expert produced code. Lastly, on the Sandy Bridge, we generated multiple implementations of *MMM* by incrementally adding additional optimizations. Starting from the best unoptimized implementation with a permuted broadcast, we achieve 80% of the machine's peak. The addition or prefetching and software pipelining brings us closer to the expert.

6 Conclusion

In this paper we suggest that an expert produces an efficient implementation of *MMM* via a mechanical process. We demonstrate that given a formal language and rule based program generation system one can capture this process in the form of rules. These rules help express the space of algorithms, which *Spiral* uses to generate *MMM* code. On a representative sample of modern systems, the code generated is competitive with expert programmers. We have shown that our language can capture high level optimization, which are described in section 3. The next step is to capture these transformations as rewrite rules, which are not tied to a specific operation. In this way, we can extend these techniques on other operations such as stencil codes or signal processing codes.

References

1. Goto, K., van de Geijn, R.: Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw. **34** (2008) 12:1–12:25
2. Van Zee, F., van de Geijn, R.: BLIS: A Framework for Rapidly Instantiating BLAS Functionality. ACM Trans. Math. Softw. (2013)
3. Spampinato, D., Püschel, M. A Basic Linear Algebra Compiler. ACM CGO. **23** (2014)
4. Qian, W., Xianyi, Z., Yunquan, Z., Yi, Q. AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs. Intl. Conf. High Perf. Comp. (2013)
5. Franchetti, F., de Mesmay, F, McFarlin, D., Püschel, M. Operator Language: A Program Generation Framework for Fast Kernels Proc. IFIP Conf. DSL, LNCS, Springer, **5658** (2009) 385-410
6. Franchetti, F., Püschel, M. Formal Loop Merging for Signal Transforms. PLDI, (2005) 315–326
7. Püschel, M., Moura, J., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R., Rizzolo, N. SPIRAL: Code Generation for DSP Transforms, Proc. IEEE, "Prog. Gen. Opt. Adap." **93** (2005) 232– 275
8. Marker, B., Batory, B., van de Geijn, R. Code Generation and Optimization of Distributed-Memory Dense Linear Algebra Kernels. iWAPT, (2013).
9. Lam, M. Software pipelining: an effective scheduling technique for VLIW machines. PLDI, (2008) 318–328
10. R. Clint Whaley and Jack Dongarra, "Automatically Tuned Linear Algebra Software", SIAM Conf. Par. Proc. Sci. Comp. (1999)