Iterative Compilation with Kernel Exploration

D.Barthou[2], S.Donadio[12], A.Duchateau[2], W.Jalby[2], and E. Courtois[3]

[1] Bull SA Company, France
[2] Université de Versailles, France
[3] CAPS Entreprise, France

Abstract. The increasing complexity of hardware mechanisms for recent processors makes high performance code generation very challenging. One of the main issue for high performance is the optimization of memory accesses. General purpose compilers, with no knowledge of the application context and approximate memory model, seem inappropriate for this task. Combining application-dependent optimizations on the source code and exploration of optimization parameters as it is achieved with ATLAS, has been shown as one way to improve performance. Yet, hand-tuned codes such as in the MKL library still outperform ATLAS with an important speed-up and some effort has to be done in order to bridge the gap between performance obtained by automatic and manual optimizations.
In this paper, a new iterative compilation approach for the generation of high performance codes is proposed. This approach is not application-dependent, compared to ATLAS. The idea is to separate the memory optimization phase from the computation optimization phase. The first step automatically finds all possible decompositions of the code into kernels. With datasets that fit into the cache and simplified memory accesses, these kernels are simpler to optimize, either with the compiler, at source level, or with a dedicated code generator. The best decomposition is then found by a model-guided approach, performing on the source code the required memory optimizations.
Exploration of optimization sequences and their parameters is achieved with a meta-compilation language, X language. The first results on linear algebra codes for Itanium show that the performance obtained reduce the gap with those of highly optimized hand-tuned codes.

1 Introduction

The increasing complexity of hardware mechanisms incorporated in modern processors makes high performance code generation very challenging. One of the key difficulty in the code optimization process is that several issues have to be simultaneously addressed/optimized: for example maximizing instruction level parallelism (ILP) and optimizing data reuse across multilevel memory hierarchies. Moreover, very often, a code transformation will be beneficial to one aspect while it will be detrimental for the other one. The whole problem worsens because the issues are tackled by different levels of the compiler chain: most of the ILP is optimized by the backend while data locality optimization is performed at a higher level.

A good example for highlighting all of these problems is the simple matrix multiply operation. Although the code is fairly simple, none of the recent compilers is really able to generate performance close to hand coded routines. For dealing with this problem, Dongarra et. al.[18] have developed a specialized code generator (ATLAS) combining iterative techniques and experimentation. ATLAS is a very good progress in the right direction (it outperforms most of the compilers) but very often it still lags behind hand coded routines. Recently, ATLAS has been improved by replacing the iterative search by an adapted cost model enable to generate code with nearly the same performance [21]. But even with these recent improvements, vendor [8,16] or hand-tuned BLAS3 [11] still outperforms ATLAS compiled codes and,

up to now, such libraries are the only ones capable of reaching near-peak performance on linear algebra kernels. So what is ATLAS and more generally compilers still missing in order to reach this level of performance ?

In this paper we propose an automated process (i.e. no hand coding) which allows to close this gap. The starting point is to decouple the two issues (ILP and data locality optimizations) and then to solve them separately. For the matrix multiply operation, blocking is performed to produce a primitive which will operate on subarrays fitting in cache. This blocking does not provide us with a single solution but rather with constraints on block sizes. Then for optimizing the primitive which is still a matrix multiply, we use a bottom up approach combined with systematic exploration. In general, the triply nested loop will be too complex to be correctly optimized by a compiler even if the operands are in cache (i.e. no blocking for cache has to be performed). Therefore, from the triply nested loop, several kernels (using interchange, strip mine, partial unroll) are generated. These kernels are one up to three dimensional loops, loop bodies containing several statements resulting from the unrolling. Additionally, to simplify compiler's task, loop trip count is set to a constant. The rationale for such kernels is to be simple enough so that a decent compiler can generate optimal code. Then all of these kernels are systematically tested to determine their performance. As a result of this process, a set of kernels reaching close to peak performance is produced. From these kernels, the primitives can be easily rebuilt. And finally, taking into account block size constraints, a primitive is selected and the whole code is produced.

As we will demonstrate in this paper, such an approach offers several key advantages: first it generates very high performance codes competitive with existing libraries and hand tuned codes, second it relies on existing compilers, and third it is extremely flexible i.e. capable of accommodating arbitrary rectangular matrices (not only the classical square matrix operation).

The approach proposed is demonstrated on BLAS kernels but it does not depend upon any specifics of the matrix multiply and it can be applied to other codes. In contrast to ATLAS, we did not a priori select a given primitive which is further tuned. On the contrary, a large number of primitives which are automatically produced, is considered and analyzed. Each of these primitives correspond to the application of a given set of transformations/optimization. Generation and exploration of these optimization sequences and their parameters is achieved with a meta-compilation language, X language.

The approach described in this paper applies to regular linear algebra codes. More specifically, the codes considered are static control programs[9]: loop bounds can only depend linearly on other loop iteration counters or on variables that are readonly in the code. Array indices depend linearly on loop indices.

The paper is organized as follows: Section 2 describes the iterative kernel decomposition and their optimization, Section 3 briefly gives the main features of the X Language, Section 4 gives experimental results performed on various matrix shapes comparing our approach with ATLAS and MKL, Section 5 describes related work and Section 5 gives some future directions.

Motivating Example

Consider the standard code for matrix vector product given in Figure 1. We will assume that the matrix sizes M, N are such that both matrix and vector fit in cache.

The same code is transformed into the code of Figure 2 by unrolling two times the outer loop (tail code is not presented). The inner loop is no longer a simple daxpy but a variant called daxpy2. The transformation process can be applied with different unrolling degree values 3, 4,... resulting in similar inner loops called daxpy3, daxpy4,... Therefore, for the same original matrix vector routine, different decompositions can be obtained using different kernels daxpy2, daxpy3,...

```
for (i = 0; i < M; i++)
  ML = &M[i][0]; b = B[i];
  // DAXPY code
  for( j = 0 ; j < N ; j++)
    A[j] += b * ML[j];
```

Fig. 1. Dgemv with daxpy

```
for (i = 0; i < M; i+=2)
  ML1 = &M[i][0]  ; b1 = B[i];
  ML2 = &M[i+1][0]; b2 = B[i+1];
  // DAXPY2 code
  for( j = 0 ; j < N ; j++)
    A[j] += b1 * ML1[j];
    A[j] += b2 * ML2[j];
```

Fig. 2. Dgemv with daxpy2

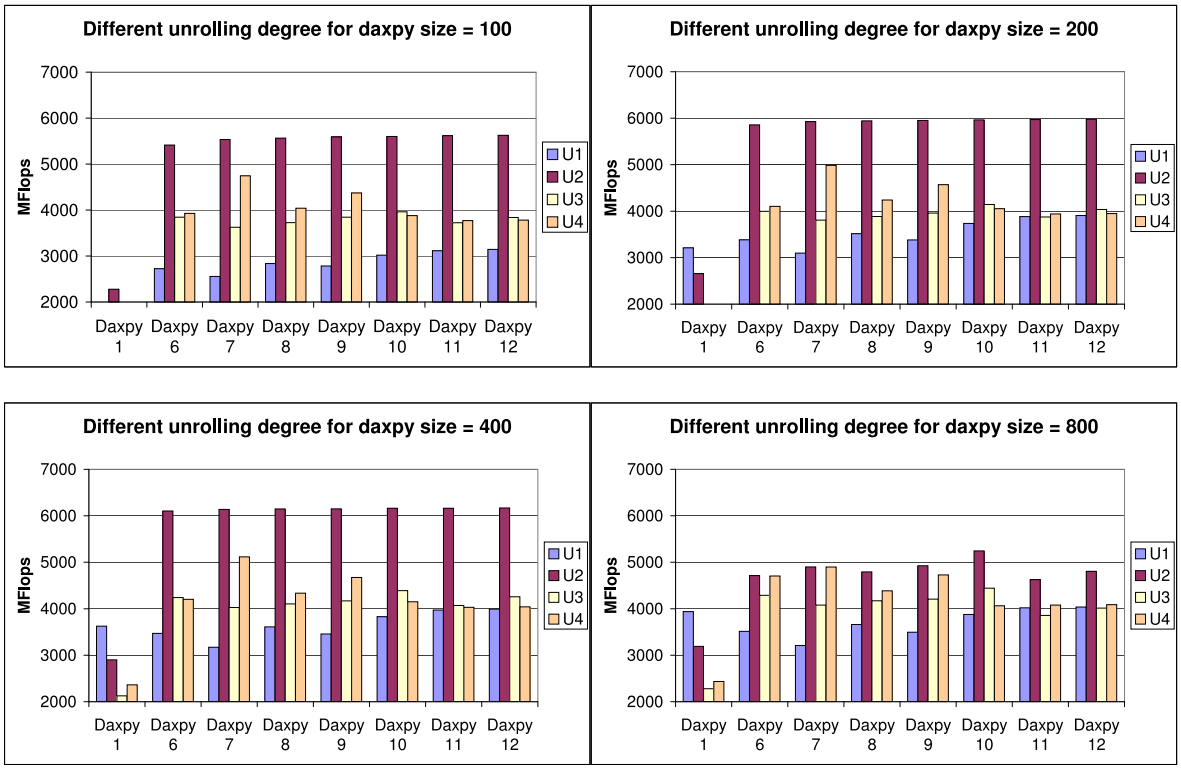

Fig. 3. Performance of daxpy with different sizes and unrolling degrees.

For this experiment, as for all others presented in this paper, the platform used is a Novascale Bull server featuring two 1.6Ghz Itanium 2 processors, with a 256KB level 2 cache and a 9MB level 3 cache. All codes are compiled using the Intel C compiler (`icc`) version 9.0 with `-O3 -fno-aliases` flags. For each of the unrolling factor of the outer loop resulting in the different kernels daxpy2, daxpy3, ..., the j loop was also unrolled 1, 2, 3 and 4 times (resp. U1, U2, U3 and U4). Performance of the resulting kernels (i.e. the j loop only) for $N = 100, 200, 400, 800$ is displayed in Figure 3. Results below 2000MFlops do not appear in the figures. Due to lack of space, performance numbers of the whole matrix vector primitive

are not shown but they have been generated and they are strictly equivalent to the performance number of the primitive they are using.

This shows several points:

- The compiler is able to generate near peak performance with a daxpy12 unrolled 2 times, of size 400 (peak performance for this machine is 6300MFlops). Even if this is obtained through a vendor compiler, this shows that compiler technology is able to produce high performance codes, without the help of hand-tuned assembly code;
- There is a speed-up of 3.13 between the slowest version (daxpy 1, unroll factor of 3) and the fastest one, among the versions that are displayed. This speedup is worth the effort but the optimization parameters to use in order to obtain the fastest version are far from obvious. This advocates for an iterative method relying on search;
- Selecting the right vector length is essential to reach peak performance. Vector length 100 is too short, while vector length 400 is optimal. Vector length 800 examplifies a typical performance problem of Itanium L2 cache banking system: since $800 = 25x32$ all of the vectors ML1, ML2, etc ... start in the same L2 bank.

This illustrates the fact that micro-optimization of loop can bring substantial performance improvement, even when resorting only to compilers. The rest of the paper shows how to use this idea in order to generate automatically high-performance linear algebra codes.

2 Iterative Kernel Decomposition

The principle of the approach described in this paper is the following: the code is first tiled for memory reuse. Then each tile is tiled again to create computational kernels. The performance of these kernels is evaluated, for different tile sizes and for different memory alignments, independently of the application context. The most efficient kernels are then put back into the memory reuse tile, according to the different possible sizes of the tile and the performance of the tile is evaluated. The data layout transformations (copies, transpositions) required by the computational kernels are also evaluated by microbenchmarking. From there, a decision tree builds up, for all loop sizes, the best decomposition into memory reuse tiles and computational tiles, according to the cost previously evaluated. All the steps concerning kernel optimization are new contributions for high performance compilation approach and are illustrated in Figure 4.

Each step is presented in details in the following sections. Figure 5 represents a matrix multiplication that will illustrate each step of the approach.

2.1 Loop Tiling

The goal of loop tiling[19,4,14,13] is to reduce memory traffic by enhancing data reuse. Tiling also enables memory layout optimizations, such as copies or scalar promotions to exhibit locality for instance. Given the cache size, the tile sizes must be such that the working set used in the tile fits into the cache. Moreover, we impose that tiles are rectangular. Indeed, even for a non rectangular initial iteration domain, it is always possible to tile most of the computation with rectangular bounds. The remaining iterations represent then a negligible amount of computation. Then, rectangular tiles are easier to optimize (possible to unroll with no remainder loop, more regular address streams for prefetching,...). Due to the fact that

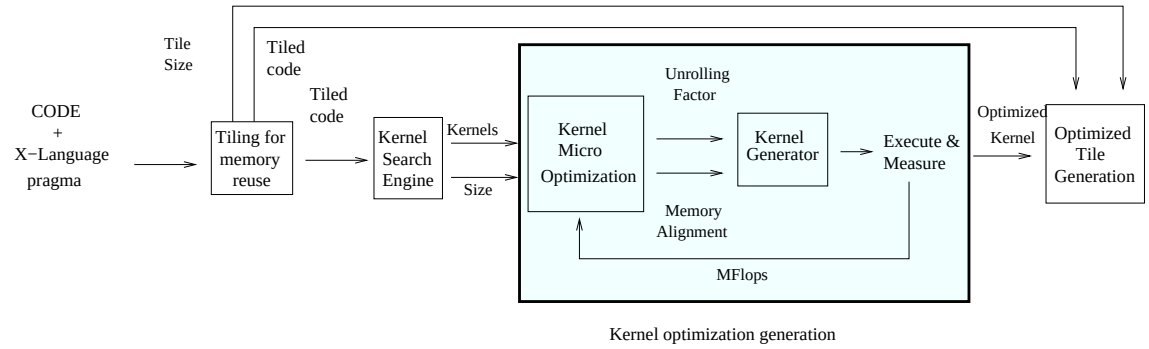

Fig. 4. Steps of iterative kernel decomposition

```
for (i = 0; i < N ; i++)
  for (j = 0; j < N ; j++)
    for (k = 0; k < N ; k++)
      c[i][j] += a[i][k] * b[k][j];
```

Fig. 5. Naive DGEMM.

```
// copy B into b by blocks of width NJ
for (i = 0; i < N ; i += NI)
  // copy A into a by block of width NK
  for (j = 0; j < N ; j += NJ)
    // copy C into c
    for (k = 0; k < N ; k += NK)
      // Tile for memory reuse
      for (ii = 0; ii < NI; ii++)
        for (jj = 0; jj < NJ; jj++)
          for (kk = 0; kk < NK; kk++ )
            c[ii][jj] += a[ii][kk] * b[kk][jj];
```

Fig. 6. Tiled DGEMM.

the global iteration space may not permit all tile sizes, a set of tile sizes that enable the partition of any outer iteration domain is considered.

The tiling is obtained through strip mine of all loops, followed by a search using loop permutation, skewing, reversal and loop distribution. Tile sizes are parametric and will be determined later in the method. The exact evaluation of the working set may lead in general to complex results, difficult to handle[3]. We choose to use for each array the min, max interval index value used in the tile as an over-approximation of the array elements accessed. Other methods, more sophisticated, can later be used.

The tiling applyied on DGEMM is presented Figure 6. The tiled code corresponds to a mini-MMM according to ATLAS terminology. The copy-out of `c` is not included.

2.2 Tiling for Computation Kernels

The code of the previously obtained tiles is then micro-optimized. It is important to stress the fact that a part of the overall code performance can only be obtained at this level. Further optimizations with scheduling among tiles or with higher level in memory hierarchy optimization may only degrade performance obtained in this level. The optimization of the tile code is in two steps: (i) we create inside the tile a simpler computation tile. Note that usually, this level of tiling corresponds to a level of blocking

for register file. Here we create instead a new tile containing one loop (1D tile) up to the dimension of the surrounding tile loops. (ii) the computation tile is optimized and evaluated in an iterative process.

The goal is to partition data reuse tiles into tiles that are simpler for a code generator (basically the compiler) to optimize. Compiler technology has a long history on low-level, low-complexity optimizations. Even if affine schedules and complex array data dependence analysis have existed since a long time, few are really implemented in vendor compilers and a large part of performance, when all the dataset is in cache, comes from the backend optimizations anyway. Simplifying source code by giving simple kernels once at a time is a method to take advantage of a code generator high quality backend (constant loop bounds enable accurate prefetching distances, opportunities for better unrolling or software pipelining).

The search for computation kernels relies on application of stripmine and loop permutations. The resulting kernels come from a selection of this inner tile. After these simple transformations, partial unroll is applied to the loops not in the kernel, generating many variations of the same family of kernel. In order to bound the search, range of unrolling factor is defined by the user with a pragma annotation in the source code. The data structures are then simplified, such that all iteration counters not in the kernel are considered as constants and projected out. A memory copy, transposition or another data layout transformation may be necessary to simplify the data layout.

The search is exhaustive, therefore a bound of the search space has to be given: for perfect loop nests, of depth n, there are $\binom{p}{n}$ possible kernels with p loops (considering any loop order, less if some dependences prevent some permutations). For each kernel with p loops, there are at most $n-p$ loops to unroll, therefore an upper bound of the number of kernels, including all versions obtained by unrolling, is $\mathcal{O}(n.2^n)$ where n is the depth of the initial loop nest. This not an issue since the maximum loop depth is usually lower than 4.

Concerning the mini-MMM code for DGEMM, searching for kernels leads to 5 different kernels, 4 of them are presented in Figures 7, 8, 9, 10. The remaining 3D kernel is the DGEMM itself. The values n, m correspond to the unrolling factor of the surrounding loops.

```
for( i = 0 ; i < ni ; i++)
  c11 += V1[i] * W1[i];
  ...
  c1n += V1[i] * Wn[i];
  c21 += V2[i] * W1[i];
  ...
  cmn += Vm[i] * Wn[i];
```

Fig. 7. 1D Kernel: dot product nm

```
for( i = 0 ; i < ni ; i++){
  V1[i] += a11 * W1[i];
  ...
  V1[i] += a1n * Wn[i];
  V2[i] += a2n * W1[i];
  ...
  Vm[i] += amn * Wn[i];
```

Fig. 8. 1D Kernel: daxpy mn

```
for (i = 0; i < ni ; i++)
  for (j = 0; j < nj ; j++)
    c1[j] += a1[j] * b[i][j];
    ...
    cn[j] += an[j] * b[i][j];
```

Fig. 9. 2D Kernel: dgemv

```
for (i = 0; i < ni ; i++)
  for (j = 0; j < nj ; j++)
    c[i][j] += a1[i] * b1[j];
    ...
    c[i][j] += an[i] * bn[j];
```

Fig. 10. 2D Kernel: outer product n

2.3 Kernel Micro-Optimization

Once kernels have been selected, their optimization is achieved. As stated before, this step mainly relies on the optimization capacity of some code generator. However, some optimizations and optimization parameters are still searched for:

- Loop bound sampling: different values of loop bounds are tested. The reason is that the loop bound impact directly the working set, using other levels of cache that the outer data-reuse tile. Moreover, mechanisms such as prefetching may be influenced by the actual value of the bound and loop overheads, pipelines with large MAKESPAN or large unrolling factors can take advantage of larger iteration counts. The span of the sampling can be user-defined through X language pragmas.
- Array alignments: the code generated may be unstable w.r.t. the alignment of the arrays starting addresses. Important performance gains can be obtained by finding the best alignment[12]. Testing the different possible alignments reveals performance stability. If stability is an issue, it is then possible to copy part of the arrays necessary for the tile with the specific alignments that enable the best performance.
- Loop transformations: interchange for kernels that have more than one loop, and unrolling (sometimes taken care of by the compiler) generate new versions of the kernel and increase parallelism. Optimizations such as software pipeline are performed by some compilers.

All experimental results are presented in Section 4. Note that as the data structures have been simplified and do no longer depend on the surrounding loops, it is quite possible to optimize the kernels in-vitro: out of the application context. The advantage of such approach is that kernel optimizations and micro-benchmarkings can be easily reused from one code to the other. The idea of using a database of highly optimized kernels is already used by CAPS with codes generated by XLG [20].

This suggests another method for kernel micro-optimization: with high performance libraries or kernels, matching the source kernel with an existing library function (source interface) would avoid completely the iterative optimization step. The kernel can then be replaced by the assembly version of the library function. Pattern-matching based techniques have been applied for instance by Bodin[2] for vectorized kernels. In general, recognizing codes even after data structure and loop transformations boils down to algorithm recognition techniques[1,15].

Finally, exploration space can be limited by static evaluation and comparison of the assembly codes. Tools such as MAQAO[6] potentially detects inefficient codes from the assembly and compare different versions. Indeed, the compiler sometimes generates the same assembly code from two different source codes.

2.4 Putting Kernels to Work

The final step consists of reassembling the code from the available kernels. This bottom-up phase first builds the data reuse tiles with kernels, according to the tile size. For a sampling of tile sizes, each decomposition in kernels is evaluated. In particular very thin tiles are studied because they necessitate special computation kernels to tile them. Then copies and other data layout transformations necessary for the kernels to work are added.

For the matrix multiplication, tiles considered are denoted by formula such as $(k \times N)X(N \times k)$: this denotes the multiplication of a matrix of size $k \times N$ where $k << N$, by a matrix $N \times k$. In these formula, k denotes an integer much smaller than N. The tile size studied are: $(k \times N)X(N \times N)$,$(N \times k)X(k \times N)$, $(k \times N)X(N \times k)$ and $(N \times N)X(N \times N)$.

Figures 11 and 12 present two mini-MMM tiled with a kernel of daxpy 10,1 unrolled 2 times (this unrolling factor concerns the loop inside the daxpy) and a kernel of dot product 1,1 unrolled 10 times. The later requires that the block `b` is transposed. The code of Figure 11 would be a good kernel for a matrix product of the form $(k \times N)X(N \times N)$ but is not adequate for a matrix product $(N \times k)X(k \times N)$ with $k < 10$ for instance. Another kernel decomposition is then needed.

```
for (ii = 0; ii < NI; ii++)
  for (jj = 0; jj < NJ; jj+=nj)
    for (kk = 0; kk < NK; kk+=10 )
      daxpy_10_u2(nj,c[ii],
        a[ii][kk],..,a[ii][kk+9],
        b[kk],..,b[kk+9]);
```

Fig. 11. Tile using daxpy 10,1 unrolled 2 times.

```
// transpose b into bt
for( ii = 0 ; ii < NI; ii++)
  for( jj = 0 ; jj < NJ; jj++)
    for (kk = 0; kk < NK; kk+=nk)
      dotproduct_u10(nk,c[ii][jj],a[ii],bt[jj]);
```

Fig. 12. Tile using dotproduct 1,1 unrolled 10 times.

Finally, according to the external loop sizes, the best performing combination of tile and memory copies is selected. Evaluation of the fastest combination requires that the memory operations are also evaluated. As a matter of fact, they are considered as kernels and are micro-optimized as well. A decision tree selects the right version.

3 X Language

X Language[7] is a language of pragmas used for meta-compilation: with the help of pragmas, a user can:

- Specify fragments of codes for which X Language transformations apply, using `#pragma xlang begin` and `#pragma xlang end` directives around selected code;
- Trigger some source to source transformations on the specified code using specific pragma directives, such as

  ```
  #pragma xlang transform tile(i,II,STRIDE)
  #pragma xlang transform unroll(i,UNROLL)
  ```

  to first tile the loop `i` with a stride `STRIDE` into a new loop `II` and then unroll this new loop by a factor of `UNROLL`. Available transformations include unrolling, tiling, fission, fusion, interchange, scalar promote,... The transformation engine is in Prolog and transformations can easily be added to the language.
- Generate multiple versions by defining search intervals, such as

  ```
  #pragma xlang parameter STRIDE [16:128:32]
  #pragma xlang parameter UNROLL [1:8:1]
  ```

  These directives define that `STRIDE` can take any value multiple of 32 between 16 and 128. X Language then generates automatically all versions of the code fragment with these optimization parameters.
- Trigger a search for the decomposition of a code fragment into kernels:

`#pragma xlang decompose i`

This directive decomposes loop `i` into kernels. This step corresponds to the tiling into computation kernels. X Language generates as many files as different kernels found.

Compared to the version presented in [7], this version of X Language is based on a C99 front-end parser (tiny C compiler), relies on a Prolog engine for the source to source transformations and finds kernels that compose a code fragment. Micro optimization of these kernels still requires now another compilation step using X Language. Testing stability w.r.t. array alignment is achieved by another tool, Kerbe, which is not yet linked to X language. Further automation of the method presented in this paper is planned for future work.

4 Experimental Results

We study in this section different kernels to do a matrix-matrix multiplication (DGEMM) and a convolution function. We compare these results with those of the functions of library like Atlas and the Intel library MKL. As for all experiments, after the kernel decomposition, each kernel is evaluated separately and then inside the data reuse tile.

4.1 Micro-Optimization of DGEMM Kernels

All kernels are presented in previous sections.

For 1D kernels, daxpy $n, 1$ kernels ($m = 1$) have been evaluated and the results of the experiments are presented in Figure 3. All experiments have measured the impact of array alignment. Only the best results are presented. For 2D kernels, performance of matrix-vector product are presented in the following table: The Figure 13 sums up performances of the outer product kernel. The 3D kernel represents a complete

N	cycles	cycles/fma	MFlops
96	50166,7	0,544	5863,970
128	88332,9	0,539	5918,367
160	137900,5	0,538	5929,368
192	226383,5	0,614	5195,439

Table 1. Matrix-Vector multiply

matrix matrix multiplication. Its performance are shown in Figure 18 and prove that this is not an adequate kernel.

4.2 Results on DGEMM Operation

We present the best results of dgemm according to the size of the tile or of the matrices. The decomposition is automatically performed by our tool, given the detected kernels. The limit sizes (values of $k << N$) are determined by the user in X language. Here, we choose $k = 1, 2, 4, 8$ and N ranges from 100 to 1500. Results for Atlas, MKL and our method are presented for the same tile sizes.

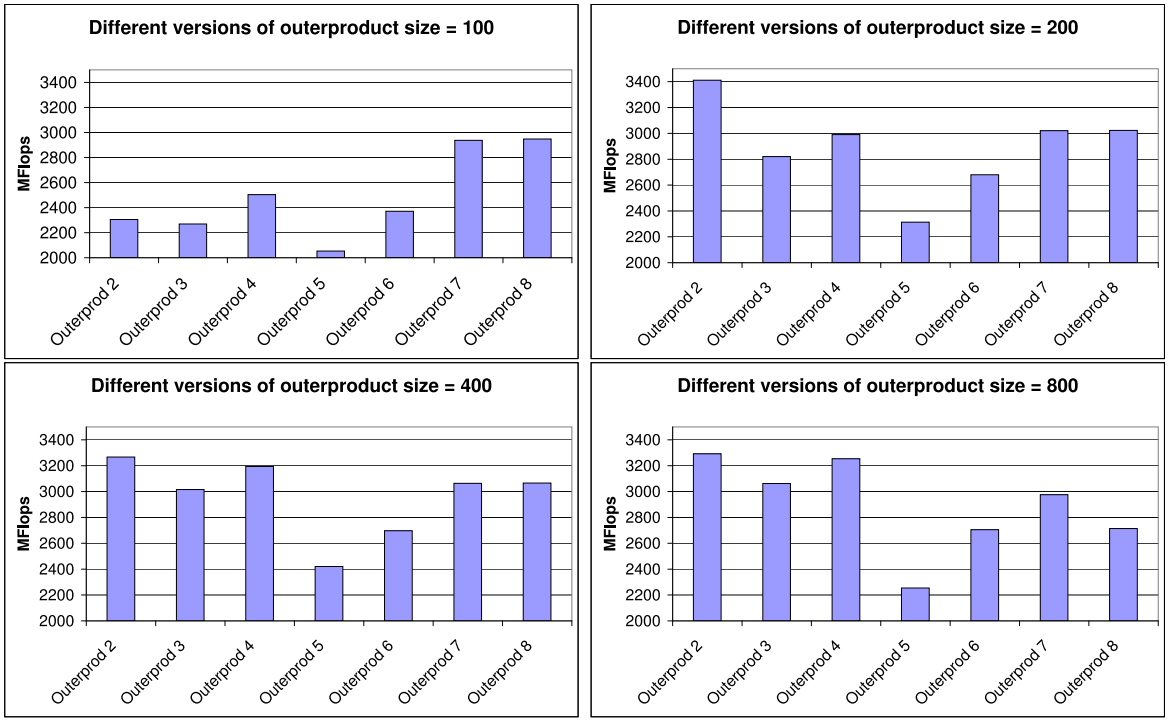

Fig. 13. Outer product performance for different versions and sizes.

Type $(k \times N)X(N \times N)$: This type of tile corresponds to a 2D kernel, performing k vector-matrix products (named Dgemv). For this type of tile, the fastest dgemm uses a kernel of dotproduct 1,1 unrolled 10 times, requiring a matrix transposition of `b`. Performance results are displayed in Figure 14 and following. A 50% speedup is obtained w.r.t. ATLAS and performance follows those of the MKL library. Performance drops around $N = 800$ because the tile size exceeds the cache size. At this point the outer tiling or the use of another kernel of our library can correct this degradation.

Type $(N \times k)X(k \times N)$: This type of tile corresponds to a 1D kernel, performing k outer products. For this type of tile where the common dimension if much lower than the others, the fastest dgemm uses the kernels of daxpy $k, 1$. Therefore each value of k requires a different kernel. Performance results are displayed in Figure 15 and following. The results outperform those of ATLAS by more than a factor of 2 and MKL is better by 50%. For $N > 900$, performance drops since the tile size exceeds the cache size, which is out of bounds for the kernel execution.

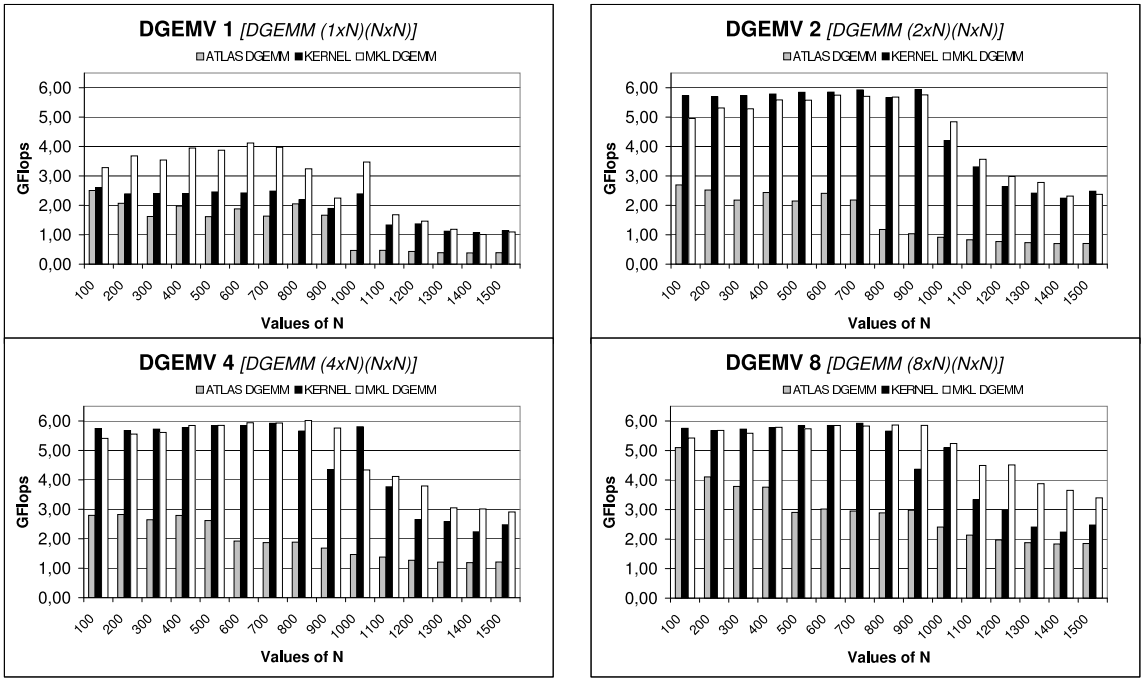

Fig. 14. DGEMM (kxN) X (NxN) with k=1,2,4,8.

Type $(k \times N)X(N \times k)$: This type of tile corresponds to a 1D kernel, performing k independent dot products. For this type of tile, the fastest dgemm uses the kernel of dot product $6, 6$. Performance results are displayed in Figure 16 and following. The results outperforms those of ATLAS by a factor of at least 3 and of MKL by a factor of at least 2. The dataset still fits into the cache for large values of N, since the resulting matrix is very small. Note that the performance of our product is very unstable w.r.t. the array alignment. Array copies when entering inside the tile prevents such unpredictability.

Type $(N \times N)X(N \times N)$: Finally, we build the code of a complete matrix product. As this step is not yet automated (construction of the decision tree), we consider only square matrices. Taking only into consideration the previous experimental results for various tile sizes, we choose to tile the matrices with rectangular matrices of the type $(k \times N)X(N \times k)$ with $k = 6$ (best performance) resorting to a kernel of daxpy $10, 1$ unrolled twice. Performance surprisingly enough matches those of the MKL, using only the compiler and source to source transformations. The performance of the naive code are shown in Figure 18 for comparison.

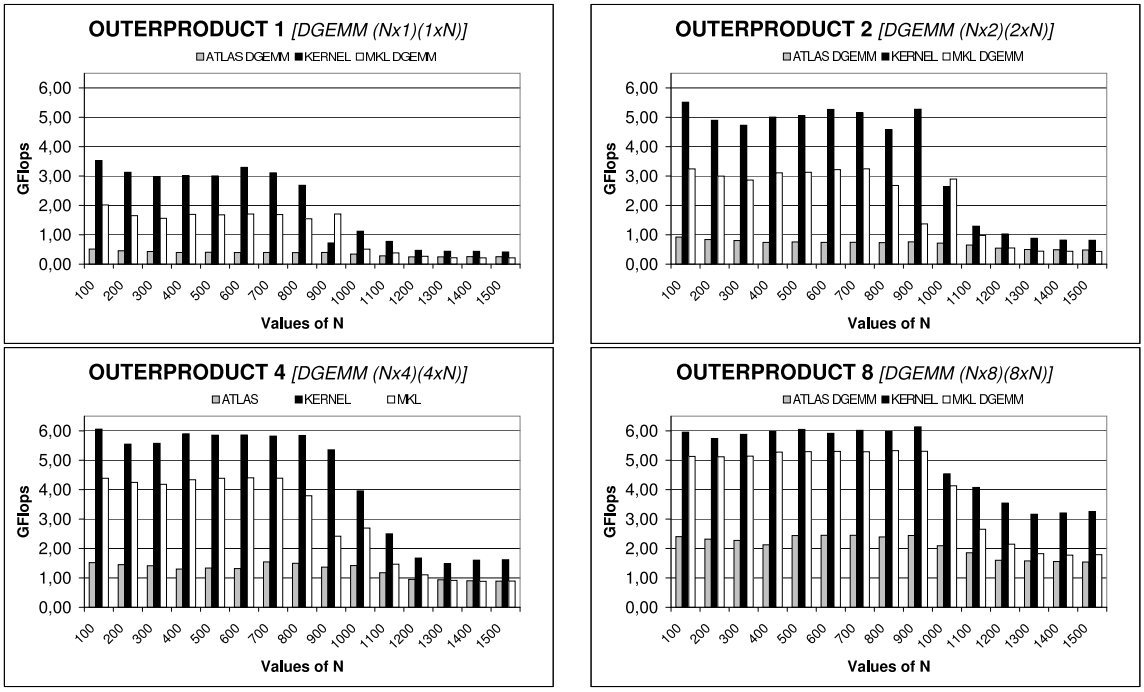

Fig. 15. DGEMM (Nxk) X (kxN) with k=1,2,4,8.

4.3 1D Convolution

This code presented in Figure 19 is an example of how to reuse kernel micro-optimization for other codes. Indeed, this code can be decomposed, after tiling, into daxpy and dot product kernels again. Using one of the previously optimized kernels leads to a 66% performance improvement.

5 Related Works

Among related works, many works have been dedicated to iteration exploration of optimization search:

Atlas[18] explores tile sizes and performs some simple micro-optimization (software pipeline, scalar promotion,...), but it relies mainly on only one kernel. This kernel was chosen according to its good ratio memory accesses/computations, not according to its performance on the target architecture. It is however possible to introduce new high performance kernels into Atlas, since there is an add-on mechanism that enable Atlas to use external, possibly hand-tuned assembly, codes. Compared with Atlas, the approach described in the paper is not limited to specific application and performs quite extensive search for the micro-optimizations, having the opportunity to find better kernels. This shows up on the performance results of previous section, where our approach compares to vendor library performance and outperforms Atlas. On the other hand, our method does not resort to exhaustive search and poor performance may result from the selected parameters. For example the exploration of tile sizes might generate unexpected results such as the poor performance numbers reported for vector length 800 in the motivating example.

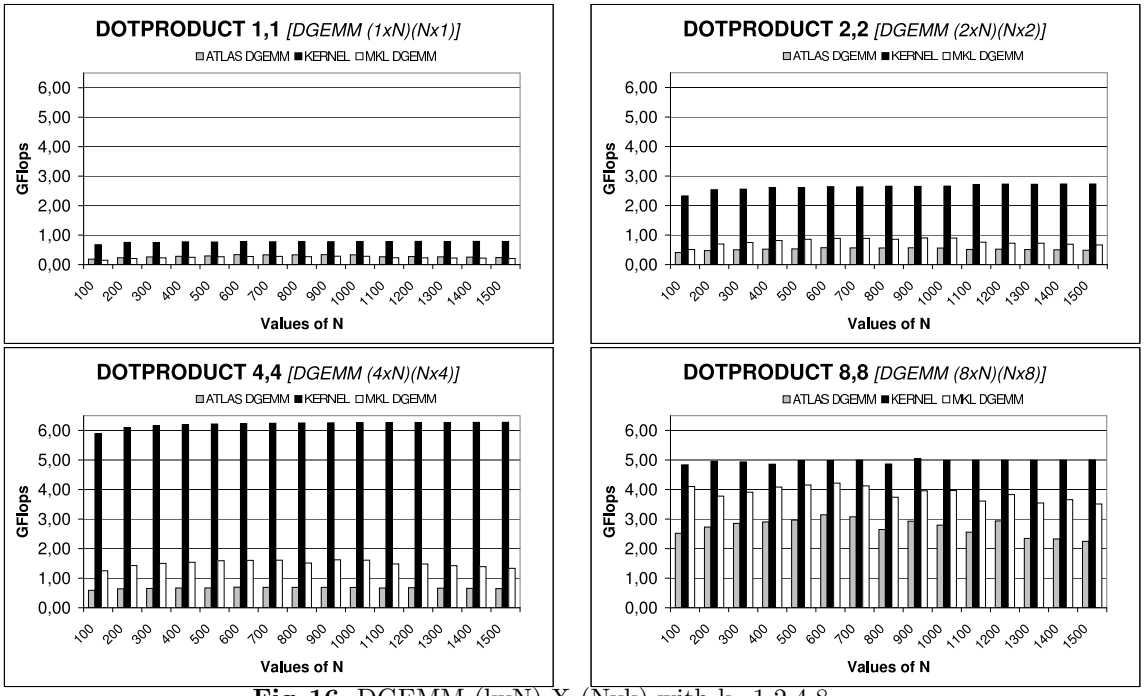

Fig. 16. DGEMM (kxN) X (Nxk) with k=1,2,4,8.

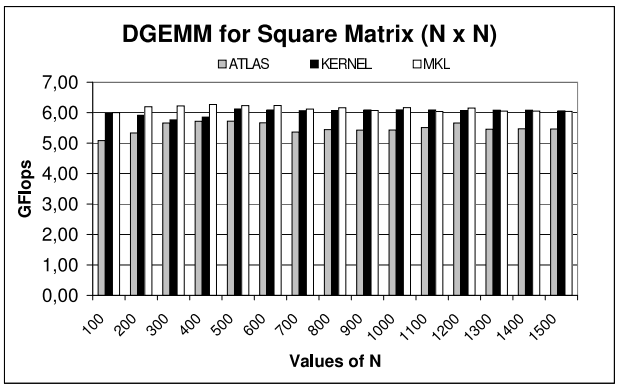

Fig. 17. Optimized DGEMM.

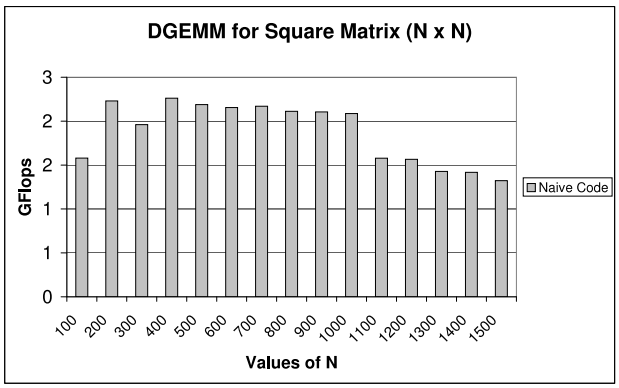

Fig. 18. Naive DGEMM

For model-based Atlas[21], the model targets essentially cache behavior. Our approach focuses more on micro-mmm optimization, and resorts to simple model based tiling and then iterative search for finding tile sizes, guided by the user. The use of more complex models ([10] for instance) is still possible.

Extensive search among optimizations[5] shows that it is difficult to understand the links between optimization parameters, optimization sequence and performance. The exploration proposed by the authors is very time consuming and yet does not include many optimizations. In comparison, our method

```
for(i=0;i<N-n;i++)
  for(j=0;j<2*n;j++)
    a[i] += b[j] * c[i-j+n];
```

Fig. 19. Code of 1D convolution.

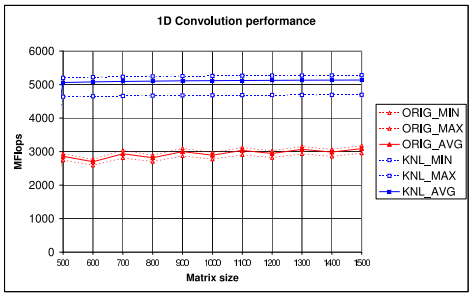

Fig. 20. Results of 1D convolution with daxpy 5,1 unrolled twice.

resorts to a very small number of transformations and relies on existing compiler to perform adapted optimizations.

The compiler optimization space exploration proposed by [17] changes the heuristic guiding optimizations by a search. This search is not exhaustive and is guided by some cost function. The goal is mostly to improve the optimization step of the compiler but does not seem to be aggressive enough to apply to library optimization.

Finally, [11] describes a methodology for hand-tuned optimization, applied to BLAS optimization. The authors propose a decomposition of micro kernel similar to ours, according to different tile sizes. The main focus of this work is to compact the data layout (making copies or transpositions of arrays) in order to improve TLB hit ratio. All the fine-tuning of micro kernels is however performed by hand. In comparison, our approach is automatic, at the expense of a small performance degradation, and is not specific to matrix matrix multiplication.

6 Conclusions/Future Directions

In this paper, we introduced a new automated approach for generating highly optimized code addressing simultaneously ILP issues as well as data locality issues. This approach relies on state of the art compiler and does not require any hand coding. This approach has been successfully validated on Itanium and BLAS3/BLAS2 routines, outperforming ATLAS and being very competitive with MKL highly tuned routines.

To be successful, this approach requires a state of the art compiler capable of generating kernels with performance close to peak. We performed experiments replacing icc with gcc; unfortunately gcc is far from being able to generate good code even on simple DAXPY like kernels and the overall performance results were pretty low. Now, when looking further at the exact requirements of our approach, what is essential is the ability to compile simple code structure, i.e. one dimensionnal loops with a loop body containing regular array access. Such capabilities are provided for example by XLG[20] code Tuner developped at CAPS Entreprise which is using specific code optimization techniques for well structured vector loops. Experiments were also performed replacing icc by XLG code Tuner: the results in terms of overall performance were similar at least for medium and large matrix sizes. However, XLG Caps Tuner was much easier to drive than icc (requiring less tuning parameters) and for small matrix sizes, XLG Caps Tuner also is capable of generating better code.

Finally, two directions are the main focus for future works: (i) More codes and libraries need to be tested with this approach, (ii) More architectures need to be tested besides Itanium. It includes not only testing other uniprocessor but also tackling the multicore/multithread case.

References

1. C. Alias and D. Barthou. On Domain Specific Languages Re-Engineering. In *ACM Int. Conf. on Generative Programming and Component Engineering*, pages 63–77, Tallinn, Estonia, September 2005. LNCS 3676, Springer-Verlag.
2. F. Bodin, Y. Mevel, and R. Quiniou. A user level program transformation tool. In *ACM Int. Conf. on Supercomputing*, pages 180–187, New York, NY, USA, 1998. ACM Press.
3. P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *ACM Int. Conf. on Supercomputing*, pages 278–295. ACM Press, 1996.
4. S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *ACM Conf. on Programming Language Design and Implementation*, pages 279–290, New York, NY, USA, 1995. ACM Press.
5. K. D. Cooper and T. Waterman. Investigating Adaptive Compilation using the MIPSPro Compiler. In *Symp. of the Los Alamos Computer Science Institute*, October 2003.
6. L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, and W.Jalby. Exploring application performance: a new tool for a static/dynamic approach. In *Symp. of the Los Alamos Computer Science Institute*, Santa Fe, NM, October 2005.
7. S. Donadio, J. Brodman, K.Yotov, T. Roeder, D. Barthou, A. Cohen, M. Garzaran, D. Padua, and K. Pingali. A language for the Compact Representation of Multiple Program Versions. In *Languages and Compilers for Parallel Computing*, Hawthorne, New York, October 2005.
8. Engineering and scientific subroutine library. Guide and Reference. IBM.
9. P. Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, February 1991.
10. B. Fraguela, R. Doallo, and E. Zapata. Automatic analytical modeling for the estimation of cache misses. In *Int. Conf. on Parallel Architectures and Compilation Techniques*, page 221, Washington, DC, USA, 1999. IEEE Computer Society.
11. K. Goto and R. van de Geijn. On reducing tlb misses in matrix multiplication. Technical report, The University of Texas at Austin, Department of Computer Sciences, 2002.
12. W. Jalby, C. Lemuet, and X. Le Pasteur. Wbtk: a new set of microbenchmarks to explore memory system performance for scientific computing. *Int. J. High Perform. Comput. Appl.*, 18(2):211–224, 2004.
13. I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *ACM Conf. on Programming Language Design and Implementation*, pages 346–357, 1997.
14. I. Kodukula and K. Pingali. Transformations for imperfectly nested loops. In *ACM Int. Conf. on Supercomputing*, page 12, Washington, DC, USA, 1996. IEEE Computer Society.
15. R. Metzger and Z. Wen. *Automatic Algorithm Recognition: A New Approach to Program Optimization*. MIT Press, 2000.
16. Intel math kernel library (intel mkl). Intel.
17. S. Triantafyllis, M. Vachharajani, and D. I. August. Compiler Optimization-Space Exploration. *Journal of Instruction-level Parallelism*, 2005.
18. R. Whaley and J. Dongarra. Automatically tuned linear algebra software, 1997.
19. M. Wolfe. Iteration space tiling for memory hierarchies. In *Conf. on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics.
20. Caps entreprise. http://www.caps-entreprise.com.
21. K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas, 2005.