Loop Optimization using Hierarchical Compilation and Kernel Decomposition

Denis Barthou [1] Sebastien Donadio [2 3] Patrick Carribault [2 3] Alexandre Duchateau [3]
William Jalby [1 3]

[1] PR*i*SM Laboratory, Université de Versailles Saint-Quentin, France
[2] Bull SA, Les Clayes sous Bois, France
[3] LRC ITACA, CEA/DAM and Université de Versailles Saint-Quentin, France

Abstract

The increasing complexity of hardware features for recent processors makes high performance code generation very challenging. In particular, several optimization targets have to be pursued simultaneously (minimizing L1/L2/L3/TLB misses and maximizing instruction level parallelism). Very often, these optimization goals impose different and contradictory constraints on the transformations to be applied.

We propose a new hierarchical compilation approach for the generation of high performance code relying on the use of state-of-the-art compilers. This approach is not application-dependent and do not require any assembly hand-coding. It relies on the decomposition of the original loop nest into simpler kernels, typically 1D to 2D loops, much simpler to optimize.

We successfully applied this approach to optimize dense matrix muliply primitives (not only for the square case but to the more general rectangular cases) and convolution. The performance of the optimized codes on Itanium 2 and Pentium 4 architectures outperforms ATLAS and in most cases, matches hand-tuned vendor libraries (e.g. MKL).

1. Introduction

The increasing complexity of hardware features incorporated in modern processors makes high performance code generation very challenging. One of the key difficulties in the code optimization process is that several issues have to be simultaneously addressed: 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 to 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 dense matrix multiply operation. Although the code is fairly simple, none of the recent production compilers is really able to generate performance close to hand-coded routines.

To deal with this problem, Whaley *et al.* [20] have developed a specialized and iterative code generator (ATLAS). ATLAS generated libraries outperform codes produced by most of the compilers. Recently, the cost of the iterative compilation in ATLAS has been reduced by replacing the iterative search by a cost model, while still generating codes with nearly the same performance[23]. But even with these recent improvements, vendor [9, 17] or hand-tuned BLAS3 [12] still outperforms ATLAS generated codes. So what are ATLAS and compilers still missing in order to reach this level of performance?

In this paper, we propose an automatic method to close this performance gap. The starting point is to decouple the two issues: locality optimization and ILP optimization. Tiling is first performed to produce a tile code operating on subarrays that fit in cache. Tiling constraints, ensuring that the subarrays accessed within the tile fit in cache, do not define a single tile size but rather a set of tile sizes. We use this degree of freedom to search for the best performing tile code and choose the tile size according to the constraints on this code. Then, to optimize the tile code, we use a bottom up approach combined with systematic exploration. In general, the multiply-nested loop structure of the tile code will still be too complex to be correctly optimized by a compiler even if the operands are in cache. Therefore, from the multiply-nested loop, we systematically generate several *kernels*, using interchange, strip mining, and partial unrolling. These kernels have a simpler control structure (1 or 2 dimensional loops), the loop body containing several statements resulting from unrolling surrounding loops. Additionally, to simplify the compiler task, loop trip counts are set to constants, and multidimensional array accesses are simplified. The main constraint on these kernels is to

be simple enough so that a decent compiler can generate high performance code. Then, the performance of all of these kernels is systematically measured. From these kernels, different variants of the original tile code can be easily rebuilt. And finally, taking into account tile size constraints, a specific version of the tile code 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 source-to-source transformations. Third, it is extremely flexible, *i.e.* capable of accommodating arbitrary rectangular iteration space.

The proposed approach is demonstrated on BLAS3 codes but can be applied to other codes. Unlike ATLAS, we did not *a priori* select a given code that is further tuned. On the contrary, we consider a large number of variants, which are automatically produced. Each of these variants correspond to the application of a given set of transformations/optimizations to the original tile code. Generation and exploration of these optimization sequences and their parameters are 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[10]: loop bounds have to depend linearly on other loop iteration counters or on read-only variables. Array indices also have to depend linearly on loop indices.

The main contributions of the proposed approach are:

- Automate the process of generating high performance code optimizing simultaneously ILP and data locality. The main contribution here is that our approach allows to find the best tradeoff between these two optimization targets
- Achieve performance similar to hand coded routines
- Rely on flexibility (different versions) of the code generated to match varying data locality properties. For example, the real difficulty is not generating a matrix multiply code achieving high performance on square matrices, but a general matrix multiply code that will obtain high performance for arbitrary rectangular shaped matrices (where locality properties on each array can be very different)
- Reduce the cost of the optimization phase. Most of the search phase and experimentation phase are done on the kernels (which are mostly one dimensional loops) and not on the whole code

The paper is organized as follows: Section 2 describes the hierarchical kernel decomposition and the kernel optimization, Section 3 gives implementation details and experimental results obtained on different linear algebra codes comparing our approach with ATLAS and MKL, Section 4 describes related work and Section 5 provides some future directions.

1.1. Motivating Example

The histograms in both Figure 1 and Figure 2 compare the performance of two versions of a square matrix multiply primitive (DGEMM): one is generated by ATLAS (grey bars) and the other one is from the Intel MKL library (white bars). The target machine is an Itanium 2 running at 1.575 Ghz with a peak performance of 6.3 GFlops. MKL version clearly outperforms ATLAS version and gets performance numbers very close to peak.

Trying to understand the performance gap, we measured L2 and L3 misses for each code. The results in Figure 1 (resp. Figure 2) represent the average number of bytes fetched outside of L2 (resp. L3) per FMA (Fused Multiply Add). These metrics are simply computed according to the following formula: $(128 \times number\ of\ L2\ misses)/number\ of\ FMA$ (resp. $(128 \times number\ of\ L3\ misses)/number\ of\ FMA$). Surprisingly enough, MKL is performing on average 3 times more L2 misses than ATLAS and between 1.5 and 2 times more L3 misses (for matrices larger than 900×900). We checked that the number of prefetch instructions is similar in both cases and we also looked at the impact of TLB misses: again in both cases the impact is negligible meaning that both ATLAS and MKL have done an excellent job at minimizing TLB misses. It should be noted that although the stress imposed by MKL routines on L2 bandwidth (resp. on memory bandwidth) is much higher, this is still below the sustained bandwidth achievable by L2 cache: 24 GB/s (resp. by memory system: 6.4 GB/s). Similar experiments performed on Pentium 4 lead to similar observations.

What conclusions should be drawn from this example?

- Minimizing L2, L3 and TLB misses should not be the only goals;
- Furthermore, increasing L2, L3 and TLB miss rate (but still staying under the hardware bounds) can be the right path to reach peak performance;
- The real key to achieve peak performance is to achieve the right tradeoffs between ILP optimization and data locality optimization.

It should be noted that our results are fundamentally different from those obtained by Chen *et al.*[3] that are essentially focused on data locality optimization across all of the memory hierarchy levels.

In the sequel of this paper, we will show how our approach succeeds in achieving a well balanced optimization of all of these metrics, resulting in performance levels close and even superior to MKL.

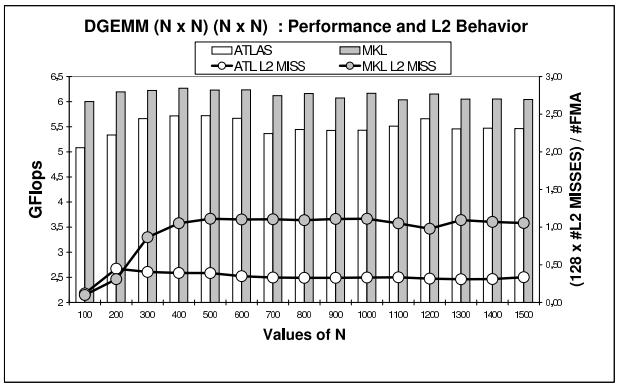

Figure 1. DGEMM performance and L2 behavior for ATLAS and MKL on Itanium 2

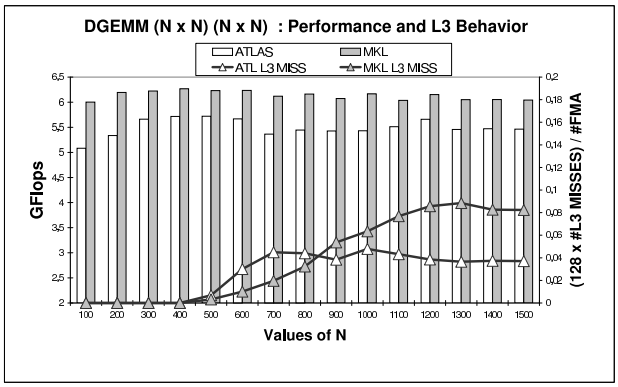

Figure 2. DGEMM performance and L3 behavior for ATLAS and MKL on Itanium 2

1.2. X-language Framework

In order to generate multiple versions of a code, we resort to a two-stage compilation framework. Source-to-source transformations are expressed in a meta-compilation language and are applied on the code after the first compilation stage. The second stage corresponds to a usual compilation phase. We used X-language[8], a language of pragmas, to describe these sequences of transformations. X-language pragmas enable to:

- Specify code fragments (scope) on which X-language transformations apply, using `#pragma xlang begin` and `#pragma xlang end` directives around selected code;
- Trigger source-to-source transformations on code fragment using pragma directives, such as

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

  The first directive tiles the loop `i` with a stride `STRIDE` into a new loop `II`. The second one partially unrolls `II` by a factor of `UNROLL`. Available transformations include unrolling, tiling, fission, fusion, interchange, scalar promote, ... The rule-based transformation engine of X-language enables more transformations.

The main advantage of X-language is that the user can apply very precisely desired source-to-source transformations. However, long optimization sequences are tedious to write with pragmas. Furthermore, there is no real search mechanism for optimization sequences and there is no feed-back from the compiler.

2. Hierarchical Kernel Decomposition

This section presents the detailed method for hierarchical kernel decomposition. Algorithm (Figure 3) sums up the main steps of the approach, further detailed in the following sections.

The code is first tiled for data locality. At this stage, the exact tile sizes are not selected yet, they are still parameters (Step 1 of Algorithm in Figure 3). But we assume that tile sizes are such that the whole array regions accessed in the tile fit into cache. The code associated with a tile is called the tile code.

On the resulting tile code, various code transformations are applied resulting in a large number of versions (Step 2). Then the loops of these versions are decomposed into simpler computational kernels: for example, for a multiply-nested do loop, tiling the innermost loop for 100 iterations generates a simple kernel with one loop of 100 iterations. More complex kernels can be obtained by tiling the two innermost and so on (Step 3a). We choose to bound the exploration of these new tile sizes to some given constants. Note that this tiling level is not directly linked to memory reuse concerns. Its purpose is to define some code fragments independent of the application that will be executed and their performance measured. In order to avoid any unnecessary tail code, the loop must have an iteration count multiple of the tile size (Step 3b), for example a multiple of 100 iterations.

Array accesses can then be optimized by scalar promotion (boiling down to loop invariant removal): array regions are copied into temporary scalars or arrays before the newly created tile code, and then copied out from these temporaries after the tile (Step 3c). The new tile code itself is called a kernel. An important point to note here is that when these copies are hoisted at the entry and exit of the tile code

Algorithm *Optimized tile code generation*

Input: *A linear algebra code* P
Output: *A set* $\mathcal{T}$ *of optimized tile codes and* $\mathcal{K}$ *of kernels.*

1. Tile P for data reuse. Tile sizes are kept parametric. Let T denote the tile code. If there is no reuse, T = P.
2. Apply various code transformations to generate multiple versions of T.
3. For each version V generated in Step 2:

   For each loop nest L of V:

   (a) Tile any number of inner loops of L with constant loop sizes. Tile code is denoted K.

   (b) Add the following constraint to V: the tile size of V must be a multiple of the size of K (no tail code).

   (c) Apply scalar promotion to array accesses in K, adding possible copy-in/out of array sections into/from array temporaries.

   (d) Hoist all copies to entry/exit of T.

   (e) Add the resulting tile code K to $\mathcal{K}$ and the resulting copies to $\mathcal{K}$.

   Add the tiled V into $\mathcal{V}$, for all possible tilings of Step 3a.
4. For each K $\in \mathcal{K}$:

   Measure performance of K for all array alignments
5. For each V $\in \mathcal{V}$:

   Estimate the performance of V by multiplying performance of its kernels/copies by their surrounding loop trip counts. Let $p(\mathrm{V})$ denote the performance estimation of V.

   If $p(\mathrm{V}) > \max_{\mathrm{U \text{ s.t. } V<U}} p(\mathrm{U})$, then add V to $\mathcal{T}$ and remove from $\mathcal{T}$ all versions such that V < U.

Figure 3. *Optimized tile code generation*

(Step 3d), as the working set fits into the cache, these copy operations fill the cache with the data used by kernels and these kernels will execute with data already in cache. Moreover, as there is a copy for each array accessed by a kernel, it is possible to choose precisely the array alignments that minimize cache bank conflicts inside kernel code. Thus, the performance of kernels is evaluated independently of the application context, with a working set already in cache and for different values of array alignment (Step 4). Note that the execution step is limited to kernel execution. As we can choose to consider only kernels with only one single loop, independently of the application, this makes the execution step rather cheap, compared to the execution of the whole tile code. Copies are considered as kernels and are benchmarked too (data comes from memory in this case).

Finally, the best tile code is composed of of kernels and copies. All different versions of tile codes are evaluated with a simple cost model relying on kernel measured performance (Step 5). There is no execution involved in this performance evaluation step. If a tile code U has size constraints that can be checked by another tile code V, for example when V must have a loop trip count multiple of 200 and U must have a loop trip count multiple of 100, then we note U < V. When U < V and U outperforms V, then we keep only U (Step 5). For example, if the performance of U is 3 GFlops and the tile size must be a multiple of 100 whereas the performance of V reaches 2 GFlops and the tile size must be a multiple of 200, then V is outperformed by U, whatever the matrix or vector sizes. On the contrary, if the performance U is 1.5 GFlops instead, then V is chosen for sizes multiple of 200 and U for the size 100. According to the size constraints coming from each kernel, the best tile code adapted to the input matrix/vector size is easily found.

Each step is presented in detail in the following sections. We will use the standard matrix multiply code given in Figure 4 to illustrate our method step by step.

```
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];
```

Figure 4. 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];
```

Figure 5. Tiled DGEMM.

2.1. Loop Tiling

The main goal of loop tiling[21, 5, 15, 14] is to reduce memory traffic by enhancing data reuse. Upon entry in the tile, data layout (the array organization) is restructured in particular to reduce TLB misses. Indeed, all copies are hoisted at the entry/exit of the tile by Step 3d, to generate

contiguous array sections which will minimize cache interferences and TLB misses.

At this stage, the only constraints imposed on tile sizes is that they should be chosen such that the working set used in the tile fits into the cache. For matrix multiply, computing the working set is fairly obvious: it amounts to sum up the size of the three subarrays accessed by the tile code. In more general cases, the exact evaluation of the working set can be done but can be fairly complex [4]. For our purpose, in most cases a simple method based on rectangular array subsections (derived from the tile sizes) will be sufficient.

In particular, we do not limit our search to square tiles. While using general rectangular tiles will increase the number of parameters to deal with, it provides much more degrees of freedom. First, it allows to cope more efficiently with the case where the original loop iteration space is rectangular (allowing for example to deal with the general rectangular matrix multiply problem). Second, it allows even to deal efficiently with more arbitrarily shaped iteration space such as triangular ones.

The tiling is classically obtained through strip mining of all loops, followed by a search using loop permutation, skewing, reversal and loop distribution.

The tiling applied on DGEMM is presented in Figure 5. The tile code (the 3 innermost loops) corresponds to a mini-MMM according to ATLAS terminology. The copy-out of `c` is not included.

2.2 Loop Transformations

The loop transformations we considered in Step 2 are loop interchange, partial unroll, and loop fusion. The goal of these transformations is to increase the parallelism inside some loops (by unrolling or fusion), to give opportunities for higher instruction parallelism and vectorization. The optimization space is described by the range of unrolling factors for each loop, through X-language pragmas. The optimization sequences considered are all possible interchange, followed by any combination of other transformations. Note that after these source-to-source transformations, the compiler achieves other loop transformations on its own: unrolling, loop vectorization, software pipelining, versioning (for alignment).

If the number of unrolling factors used for each loop is U, a perfect loop nest with n loops fully permutable generates $n!U^n$ different versions. This is not an issue since the maximum loop depth is usually lower than 4.

2.3 Data-Layout Optimization

Loop tiling followed by scalar promotion are used to generate kernels (Steps 3a, 3c). The kernels are the tile codes generated by this tiling step. Tiling is applied only to inner loops, from the innermost loop (for 1D kernel) to all loops. The complexity of the kernels depends highly on the number of loops considered for this tiling. Kernels with only one loop (1D kernels) have several advantages: compilers usually generate good quality codes for single loops, and the experimental step takes linear time with respect to the number of experiments performed (Step 4). In our study, we consider also kernels with up to 3 loops.

Data-layout transformations are applied to simplify array structures accessed by kernel. The optimization is focused on:

- locality optimization: higher locality and regular strides give more opportunities for the compiler to generate high performance memory accesses (better prefetch, fewer instructions to describe the address stream, vectorization,...)
- register usage optimization.

These are standard scalar promotion/blocking techniques. The first goal can be reached by transforming the arrays so that they contain only the working set accessed by the kernel. This can be achieved by using well known techniques[16, 2, 18]. Our implementation is based on scalar promotion techniques, and all arrays are resized to the size of kernel loops. Array dimensions that are only indexed by constants or loop counters not present in the kernel are removed (and arrays are renamed correspondingly). This may require the use of memory copy operations.

Figure 6 presents three mini-MMM optimizations achieved by our method. The first column gives the optimization sequence used and the resulting code is in the second column. From these codes, the kernels in the third column are generated by tiling of the innermost loop and scalar promotion. Scalar promotion creates new arrays. For the first example, the mapping between array elements of the first optimized version and array elements used in its 1D kernel is the following: `c1=c[ii]`, `a1=a[ii][kk]`, `b1=b[kk]`, `c2=c[ii+1]` and `a2=a[ii+1]`, where all arrays of the kernel have `ni` elements. In C, this implies that there is no need for copy at all. For the second optimized mini-MMM of Figure 6, the mapping is: `c1=c[ii][jj]`, `a1=a[ii]`, `b1[.]=b[.][jj]`, `c2=c[ii][jj+1]`, `b2[.]=b[.][jj+1]`, etc. Note that array `b` has to be transposed before entering this kernel. For the third optimized mini-MMM, the mapping is: `c1[.]=c[.][jj]`, `a1[.]=a[.][kk]`, `b1=b[kk][jj]`, `c2[.]=c[.][ii+1]` and `b2=b[kk][jj+1]`. This means that the arrays `a` and `c` have to be transposed. Compared with the first version, even if the kernel is the same, this decomposition is likely to be less efficient.

Concerning the mini-MMM code for DGEMM, enumerating all optimizations and generating all kernels leads to 5 different kernels, 4 of them are presented in Figures 7,

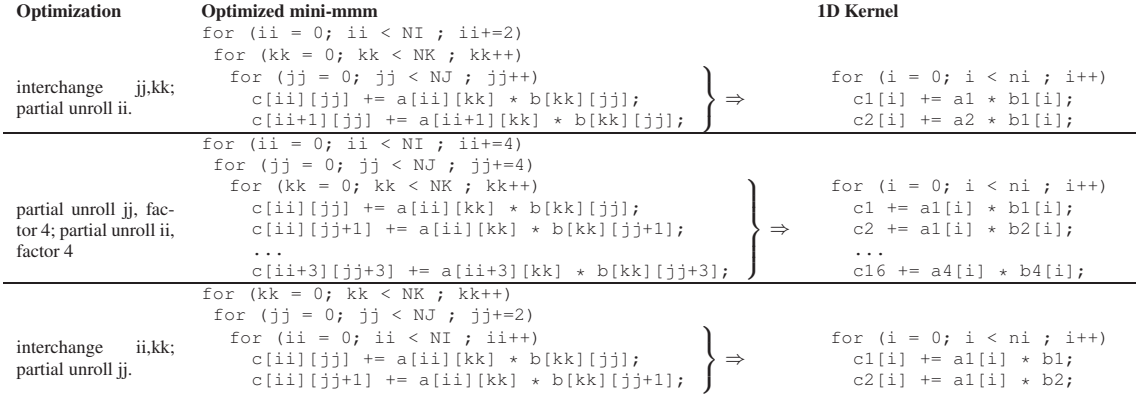

Figure 6. Several transformed mini-mmm and their corresponding 1D kernel. The first is a kernel of 2 daxpys, the second is 16 dot products and the third is the same as the first (modulo commutativity of the multiplication and renaming).

8, 9, 10. The remaining 3D kernel is the DGEMM itself. The values k, l correspond to the unrolling factor of the surrounding loops. For instance, `daxpy k,l` corresponds to daxpys accumulating in k different vectors, l daxpys sharing the same destination vector.

The method described here corresponds to a systematic enumeration of all possible optimization sequences (at the opposite of a selective search). A bound on the total number of kernels studied (which in fact correspond to the size of the search space) can be easily assessed. For perfect loop nests of depth n, for a given tile size, there are $n! \times U^n$ possible kernels consisting exactly of p loops, taking into consideration loop transformations. This is an upper bound since in practice, some dependences can forbid the use of some permutations. Moreover, the same kernels can be obtained by different transformations. Therefore an upper bound on the total number of kernels with at most p loops and exploring t different tile size values is $n! \times U^n \times p \times t^p$ where n is the depth of the initial loop nest and U the number of different unroll factors explored. Using existing search methods will be required when other transformations are considered.

When some variations are not handled (such as commutativity, as expressed in Figure 6), there are more kernels generated. The compilation step helps to detect these similarities, as explained in the following section.

2.4. Kernel Micro-optimization and Execution

Once kernels have been generated, they are optimized, compiled and evaluated separately, outside of the application context (Step 4 of Algorithm 3). While standard iterative compilation usually compiles and measures performance of the whole code, our approach focuses on smaller code fragments (the kernels) which are systematically benchmarked. This leads to substantial saving in computational time: for example testing a whole matrix multiply of size n will cost n^3 operations while testing a simple daxpy 10,1 of size n will cost $10 * n$ operations. Moreover, kernel optimizations and evaluations can be easily reused from one code to the other, when the same kernels appear.

For the evaluation of kernels, two key parameters are explored:

- Loop bounds: they correspond to tile sizes. First, loop bounds impact directly the working set, and therefore forcing to use other levels of cache. Second, mechanisms such as prefetching may be influenced by the actual value of the bound and loop overheads. Finally, pipelines with large MAKESPAN or large unrolling factors can take advantage of larger iteration counts. The span of the loop bound sampling can be user-defined through X-language pragmas.

- Array alignments: the code generated may be unstable with respect to the alignment of the array starting addresses. Important performance gains can be obtained by finding the best alignment[13]. 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.

The quality of the final code depends on the capacity of the compiler to take advantage of the parallelism expressed in the kernel. In particular, the generator has to perform the following operations appropriately:

```
for(i = 0 ; i < ni ; i++)
  c11 += a1[i] * b1[i];
  ...
  c1l += a1[i] * bl[i];
  c21 += a2[i] * b1[i];
  ...
  ckl += ak[i] * bl[i];
```

Figure 7. 1D Kernel: dot product k,l

```
for(i = 0 ; i < ni ; i++){
  c1[i] += a11 * b1[i];
  ...
  c1[i] += a1l * bl[i];
  c2[i] += a2l * b1[i];
  ...
  ck[i] += akl * bl[i];
```

Figure 8. 1D Kernel: daxpy k,l

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

Figure 9. 2D Kernel: dgemv k

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

Figure 10. 2D Kernel: outer product k

- Dependence analysis: failure to detect independence of statements degrades schedule latencies, ILP and impacts many other optimizations.
- Register allocation. Depending upon the architecture, this is more or less critical and failure to correctly allocate registers introduce dependencies, impacting latencies. Source-to-source transformations can help the compiler by using single assignment form codes.
- Vectorization: this can be memory access vectorization (Itanium, Pentium for instance) or computation vectorization (Pentium with SSE instruction set). Some code generators rely on pattern matching in order to decide whether or not to perform vectorization. Enumerating different versions of kernels helps in finding the appropriate code that can be matched by these rules.
- Constant propagation and use of static information: the compiler can take advantage for instance of the loop bounds values (which are explicitly provided) for the computation of the prefetch distance. Failure to do this means that the compiler optimizes in the same manner loops of different sizes.

Exploration space and executions can be limited by static evaluation and comparison of the assembly codes. Tools such as MAQAO[7] potentially detect inefficient codes from the assembly and compare different versions. Indeed, the compiler sometimes generates the same assembly code from two different source codes or may fail to perform some key optimization (such as vectorization for instance).

2.5. Putting Kernels to Work

The final step consists of choosing the best kernel decomposition from the available kernels. For each decomposition, the tile code is written as a combination of memory copies and kernel codes. All memory copies are hoisted and then performance of the tile is evaluated as the product of the individual performance of the kernels and the memory operations by their surrounding loop trip count. All kernel measurements are performed in the same context as they are used in the application (same alignment, same cache level).

All tiles in $\mathcal{T}$ correspond to the best tiles, given some constraints on tile sizes. In particular very thin tiles are taken into account by our approach and may require special kernels.

3. Experimental Results

First details on our implementation are presented followed by a description of the different architectures, compiler and libraries used. Then, kernel performance is described and analyzed. Finally kernel decomposition and combination on whole code is presented together with performance numbers.

3.1 Implementation of our approach

Kernel decomposition is implemented by a new pragma inside X-language. Compared to the version presented in [8], the version we have developed is based on a C99 front-end parser, tiny C compiler [1] and relies on a Prolog engine for source-to-source transformations. Most of the transformations, with their pragmas, are still the same. The main contributions of the new X-language version is the possibility to:

- 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 the decomposition of a code fragment into kernels:

`#pragma xlang decompose i`

This directive decomposes loop `i` into kernels, as explained in Section 2.2. This step corresponds to the tiling into computation kernels. X-language generates one file per kernel found.

Micro-optimization of these kernels still requires now another compilation step using X-language. Generation of the code testing array alignment stability is automatic. The automatic selection of the best tile according to the tile size is not yet automatic. Further automation of the method presented in this paper is planned for future work.

3.2 Experimental Environment

Three different codes from dense Linear Algebra have been targeted for validating our approach: Matrix Matrix Multiply (DGEMM), Matrix Vector Multiply (DGEMV) and 1D Convolution.

On the hardware side, two different architectures have been used:

Itanium 2: BULL Novascale server featuring 1.6GHz Itanium 2 processor, with 3 cache levels was used. Out of these 3 levels, only the 2nd level (256KB, unified) and the 3rd level (9MB, unified) can contain floating point values. The processor offers 128 floating point registers and can issue up to 6 instructions per cycle.

Pentium 4: PC equipped with an Intel Pentium 4 Prescott 2.80GHz processor with two cache levels: L1 D cache (16 KB) and L2 (1MB, unified) was used. The processor can issue up to 2 instructions per cycle and supports the SSE2 extensions. The SSE2 instructions allows to vectorize most of the standard floating operations (up to 2 double precision word can be packed in a single instruction). On this Pentium version, the number of registers available for SSE2 is limited to 8 making register allocation a sensitive issue.

Our kernels and codes were compiled using Intel ICC Compiler v9.0 on both platforms. The code generated with our approach was compared with the MKL library (version 8.02) and ATLAS library generator (version 3.6), the same release number being used for both machines.

3.3 Kernel Performance Analysis

Due to lack of space, we will focus on the some key kernels representative of kernel performance analysis. Since all of our kernels correspond in fact to rectangular matrix muliply operation, we also compared our kernels with MKL and ATLAS.

Dotproduct-k, k A dotproduct-k, k kernel of length n has a fairly small working set: $2 \times k \times n + k \times k$. If there is enough register space, it offers also a very good memory/arithmetic ratio: $2 \times k$ loads for $k \times k$ multiply add. Increasing k will decrease the stress on load instruction scheduling but at the same time will increase register pressure up to the point where the register allocator will have to insert expensive spill/fill instructions or to reload some of the operands. Results for the Itanium are presented in Figure 11. For all kernels (and vector length ranges), the working set is small enough so all of the data is contained the L2 cache. Increasing k improves performance because kernels become more and more floating point dominated. Dotproduct-$4, 4$ offers peak performance around 6.2 Gflops but this requires vector length of at least 200 (however using larger vector length does not generate performance loss). However, for $k = 8$, the register pressure is too high: the compiler start inserting spill/fill instructions and furthermore, the loop body becoming too complex, the compiler can no longer software pipeline the loop, explaining the much lower performance level obtained by this kernel. On the Pentium 4 (see Figure 12), the L1 size being much smaller, increasing vector length will push some of the operands out of L1. This explains the general trend on all curves, performance quickly decreases for larger vector lengths. Now, since there are only 8 SSE2 registers, starting with $k = 2$ the compiler can no longer keep data in registers but instead reloads them. From this angle, increasing k wil increase data reuse within L1 and therefore performance improves with larger values of k. However, since many reaload instructions have to be issued (even if some can be combined with arithmetic), the overall performance of all dotproduct kernels (under 1.8 Gflops) is fairly disappointing.

Outerproduct-k An outerproduct-k kernel of length n has a larger working set: $2 \times k \times n + n \times n$. The memory/arithmetic ratio is good: $2 \times n \times k$ loads for $n \times n$ stores and $n \times n$ multiply adds, but requires large temporary storage (much larger than register space). However, it should be noted that register pressure will increase quickly in function of k but also in function of n, therfore data will have to reloaded (but fortunately most of the time from cache). On Itanium 2 (see Figure 13), the increase in working set requirement explains the drop off in performance around $n = 900$. For n larger than 900, the operands can no longer be kept in the L3 cache, some of them come from main memory. A smaller drop off occurs between 100 and 200, because here operands no longer contained in the L2 cache. However, it should be noted that L3 organization is such that very quickly ($n = 400$), performance levels are high again although now a large fraction of the operation are coming from L3. This explains why minimizing the number of L2 misses is not such an important issue. Increasing k improves performance (not because of register reuse but due to reuse in L2) and allows to reach performance levels close to peak but slighly lower than the Dotproduct-k kernel. On Pentium 4 (see Figure 14), the performance drop off ap-

pears much quicker: for $n = 300$, the working set exceeds L2 cache size. Now increasing k, improves operand reuse in L1 but since L1 is fairly small, already $k = 8$ exceeds L1 capacity level. However, it should be noted that outerproduct-4 kernel on Pentium performs much better than all of the dotproduct-k, k kernels.

3.4 Kernel Decomposition/Combination

After decomposing the different target codes in several kernels, the combination step selects the best kernel (taking into account block size constraints) and then uses it to build the tile code. Furthermore, as said in Section 2.5, some memory move operations (direct copies or transpose) can be added to make sure that all array accesses during kernel execution are well aligned and use strides as much as possible. The cost of these memory copy/transpose operations is experimentally evaluated and is taken into account when performing the combination phase.

DGEMM: As described in previous sections, DGEMM can be decomposed in several ways leading to different kinds of kernels: dot product, daxpy, dgemv or outerproduct (see Figures 7, 8, 9 and 10).

For square matrices, the kernel performance analysis and the combination phase analysis selected dotproduct 4,4 kernel on the Itanium 2 and the daxpy-4,1 kernel on the Pentium 4. The resulting performance on the whole square matrix multiply is presented in Figure 16. It is interesting to note that on the Pentium 4, performance obtained still lags behind MKL. The main explanation is that we did not succeed in making the compiler make the best use of 8 SSE2 registers.

For rectangular matrices, our approach will use kernels presented in the previous sections. Now, for some performance numbers on rectangular matrices, refer back to Figures 11, 12, 13 and 14 since some of the kernels we used are in fact matrix multiply. For example, Figure 11 gives the performance results for matrix multplication of a flat matrix $k \times N$ by a tall matrix $N \times k$. Please note that such rectangular matrices cases can be used as building blocks/kernels in our approach. It is why performance results were integrated in the kernel section (Kernel decomposition).

DGEMV: Since DGEMV belongs to our kernel set, the decomposition can be straightforward using directly its equivalent in kernel, or can be more complex using 1D kernels.

Table 1 shows the performance results of combining a dotproduct-8,1 kernel for DGEMV code. This result is compared with the use of a straight DGEMV naive code and MKL one. As we can see kernel combination approach lead to speed-up on this target code.

1D Convolution: This code presented in Figure 15 is an example of how to reuse kernel micro-optimization for other codes. Indeed, this code can be decomposed again, after tiling, into daxpy and dotproduct kernels again.

N	100	200	2000	4000	6000
Naive DGEMV (GFlops)	4.92	3.59	1.23	1.23	1.18
DGEMV MKL (GFlops)	4.92	4	1.18	1.14	1.1
Composed DGEMV (GFlops)	5.71	4.38	4.38	1.83	1.92

Table 1. DGEMV and combination of dgemv (with dotproduct 8,1 kernel) on Itanium2

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

Figure 15. Code of 1D convolution.

On Itanium 2, daxpy 8 kernels were used while on Pentium 4 daxpy-10 were used. This leads to a 66% (resp. 60%) performance improvement on Itanium 2 (resp. Pentium 4) when compared with a use of the ICC compiler (see Figure 17).

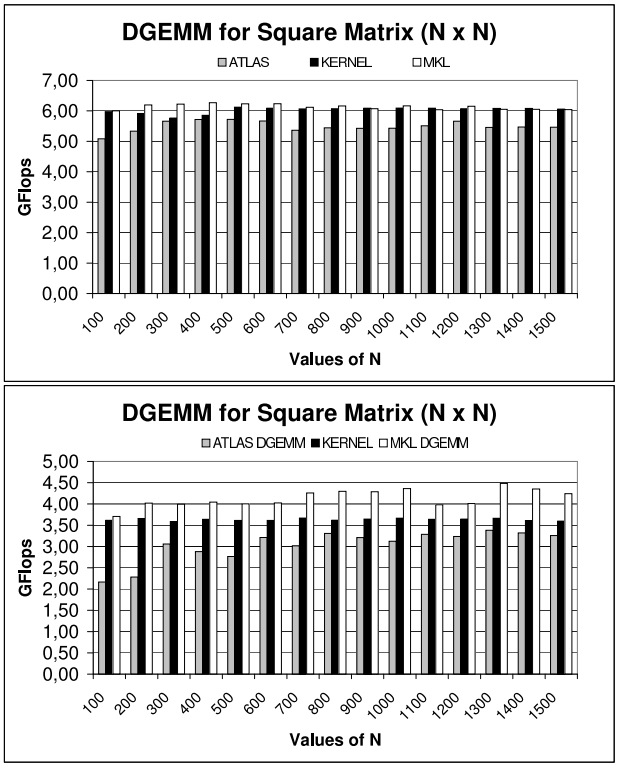

Figure 16. Matrix-matrix multiply on Itanium 2 (top) and Pentium 4 (bottom) after combination

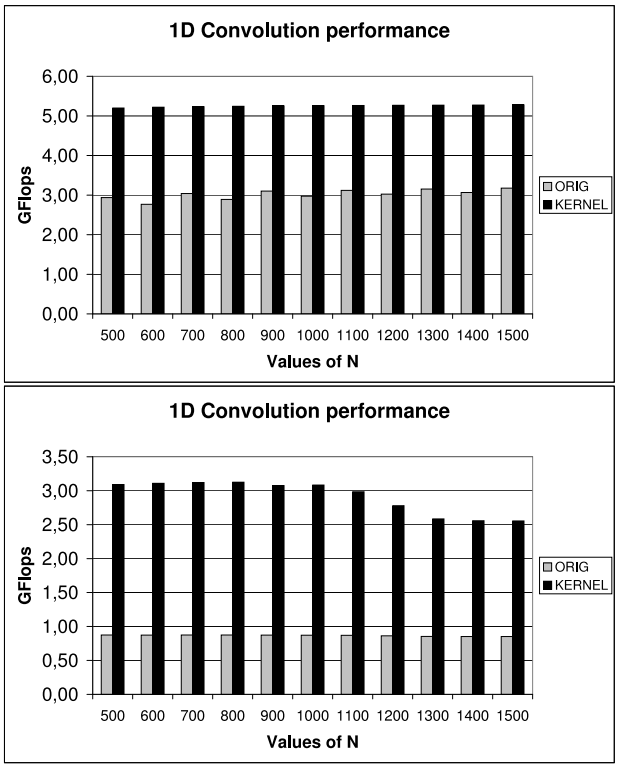

Figure 17. Results of 1D convolution on Itanium 2 (top) and Pentium 4 (bottom).

4. Related Works

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

ATLAS[20] explores tile sizes and performs some simple micro-optimization (software pipeline, scalar promotion,...), but it mainly relies on a single 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 enables 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. The advantage of our approach is clearly demonstrated by the performance results where our codes outperform ATLAS. On the other hand, our method relies on the programmer to find out the size of the cache or the maximum size of the tiles.

For model-based ATLAS[23], the model targets essentially cache behavior. Our approach focuses more on mini-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 (e.g. [11]) is still possible.

Extensive search among optimizations[6] 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 resorts to a very small number of transformations and relies on existing compiler to perform adapted optimizations.

The compiler optimization space exploration proposed by [19] changes the heuristic guiding optimizations by a search. This search is not exhaustive and is guided by a 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.

Chen *et al.*[3] describes an overall framework for optimizing multiple loop nests, focusing mainly on memory hierarchy behavior. Our approach is fairly different because we first consider the key problem to solve is to achieve a good balance between memory transfer optimization and ILP optimization. Second, we perform a much more systematic search on all the possible variants, taking into account not only cache behavior but also ILP. Third, instead of making the final adjustements for tile size (etc...) at the whole application, we perform systematic experimentation at the kernel level which is less costly, allowing exploration of more kernels and provides more flexibility: different kernels can be chosen to accomodate different input matrix sizes.

Finally, [12] 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 improving memory behavior in particular changing the data layout (making copies or transpositions of arrays) to improve TLB hit ratio. The whole 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.

5. Conclusion/Future Directions

This paper proposed a new automated approach for generating highly optimized code addressing 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 Pentium 4 architectures for BLAS3 routines, outperforming ATLAS and being very competitive with MKL highly tuned routines.

Now, it is clear that we rely in a critical manner on a ”good” compiler to achieve high performance on our building blocks/kernels. As a matter of test, we used our approach replacing ICC with GCC. On the Itanium 2, the performance results were much lower while on the Pentium the performance gap was smaller. This clearly shows the necessity of having a very good compiler for the kernels. However, since the kernels used in our approach are fairly simple, a specialized ”compiler” for such kernels can be developed. A good example of such a compiler is the XLG[22] tool developed by CAPS Enterprise. This code generator is fairly specialized in the sense that it only targets vector loops but for such loops, it is capable of applying very aggressive optimization: for example, it evaluates the performance and resource impact of several loop unrolling degrees and at the end generates several versions selecting the best one in function of the vector length.

Beyond the backend/compiler issue, this work needs to be extended into three major directions: first, pushing further the automation of the whole process (this includes pruning efficiently the optimization parameter space), second performing experiments on a larger number of codes to improve robustness of the method and third extending the approcah to cover the multicore architecture case. This last case will require delicate tradeoffs between locality, inter and intra processor parallelism.

References

[1] Tiny C compiler. http://www.tinycc.org.

[2] D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. In *Symposium on Principles of Programming Languages*, pages 98–106, 1998.

[3] C. Chen, J. Chame, and M. W. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *CGO ’05*, pages 111–122, 2005.

[4] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *ICS ’96*, pages 278–295, 1996.

[5] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *PLDI ’95*, pages 279–290, 1995.

[6] K. D. Cooper and T. Waterman. Investigating Adaptive Compilation using the MIPSPro Compiler. In *Proc. of the Symp. of the Los Alamos Computer Science Institute*, October 2003.

[7] 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 *Proceedings of the 6th LACSI Symposium*, Santa Fe, NM, Oct. 2005.

[8] 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 *LCPC ’05*, Hawthorne, New York, Oct. 2005.

[9] Engineering and scientific subroutine library. Guide and Reference. IBM.

[10] P. Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, Feb. 1991.

[11] B. B. Fraguela, R. Doallo, and E. L. Zapata. Automatic analytical modeling for the estimation of cache misses. In *PACT ’99*, page 221, 1999.

[12] 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.

[13] W. Jalby, C. Lemuet, and X. L. 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.

[14] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multilevel blocking. In *PLDI ’97*, pages 346–357, 1997.

[15] I. Kodukula and K. Pingali. Transformations for imperfectly nested loops. In *Supercomputing ’96*, page 12, 1996.

[16] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3–4):649–671, 1998.

[17] Intel math kernel library (intel mkl). Intel.

[18] W. Thies, F. Vivien, J. Sheldon, and S. P. Amarasinghe. A unified framework for schedule and storage optimization. In *PLDI ’01*, pages 232–242, 2001.

[19] S. Triantafyllis, M. Vachharajani, and D. I. August. Compiler Optimization-Space Exploration. *The Journal of Instruction-level Parallelism (JILP)*, 2005.

[20] R. Whaley and J. Dongarra. Automatically tuned linear algebra software, 1997.

[21] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, 1989.

[22] Caps entreprise. http://www.caps-entreprise.com.

[23] 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.

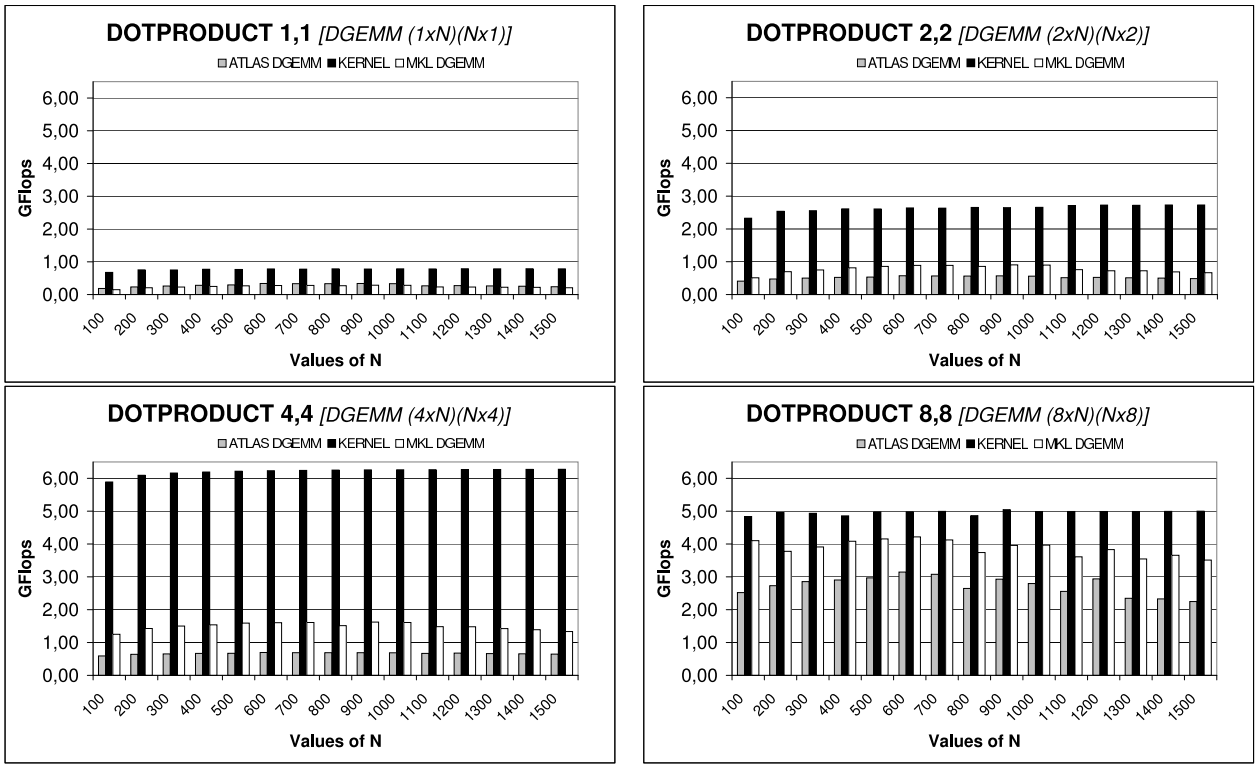

Figure 11. Performance of dotproduct-k, k kernel on Itanium 2 with $k = 1, 2, 4, 8$.

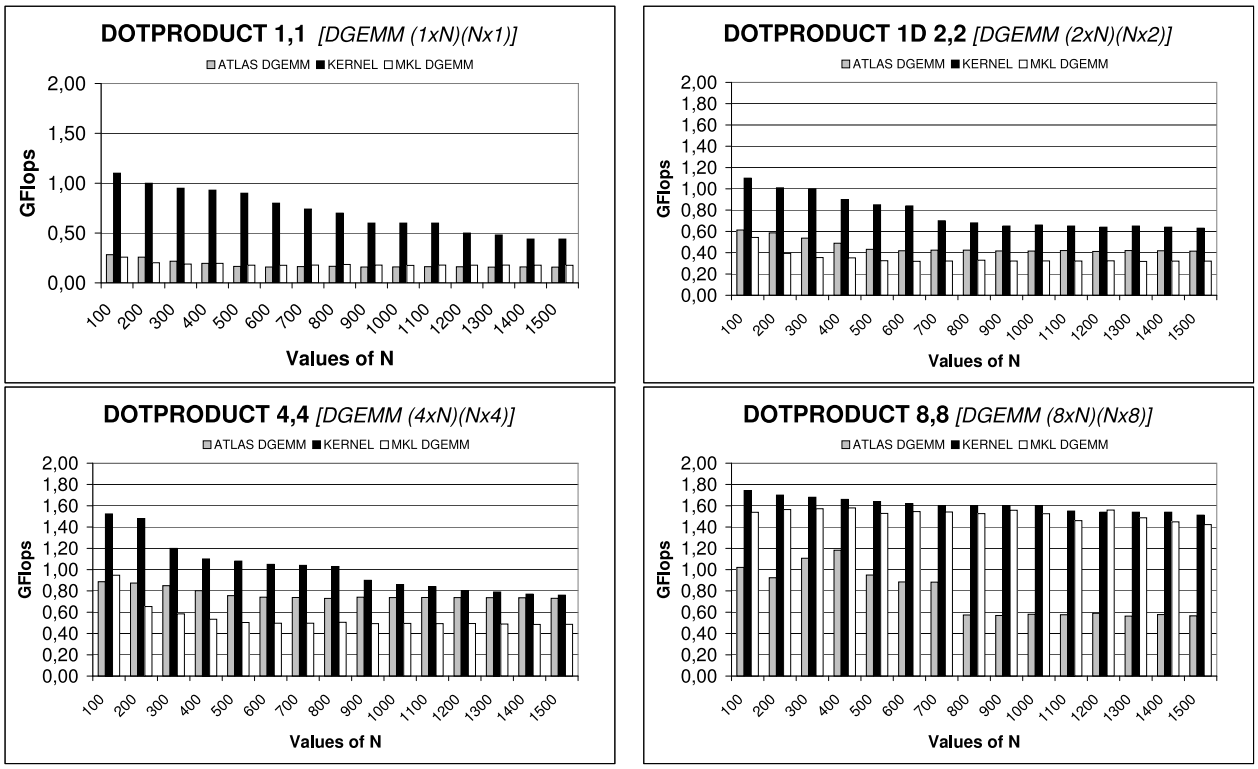

Figure 12. Performance of dotproduct-k, k kernel on Pentium 4 with $k = 1, 2, 4, 8$.

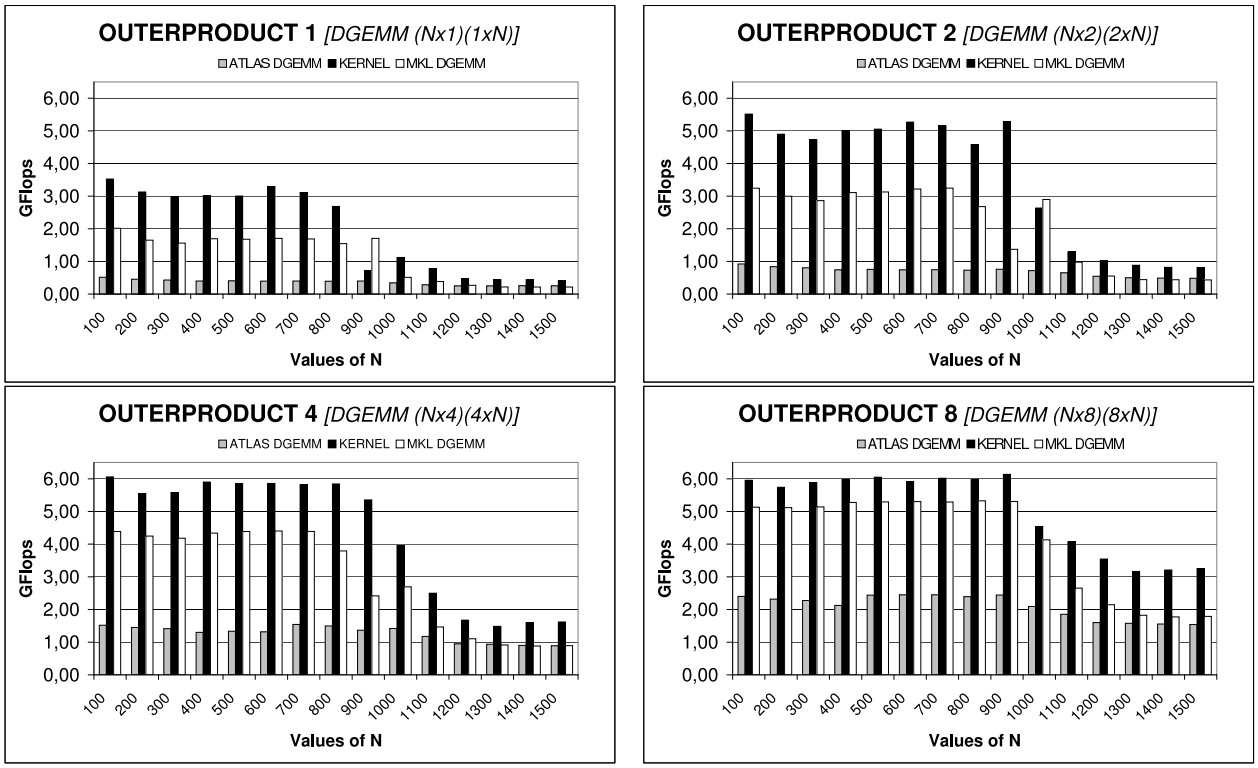

Figure 13. Performance of outerproduct-k kernel on Itanium2 with $k = 1, 2, 4, 8$.

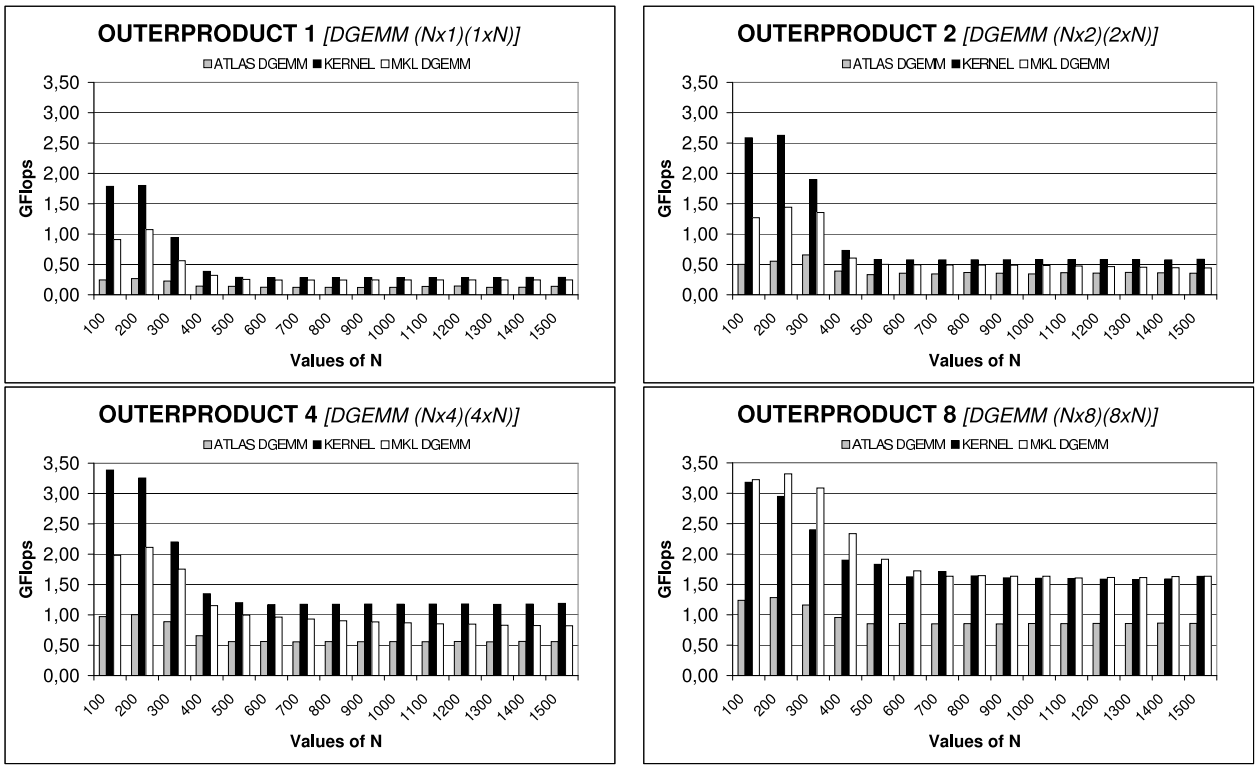

Figure 14. Performance of outerproduct-k kernel on Pentium4 with $k = 1, 2, 4, 8$.