Static versus Dynamic Memory Allocation: a Comparison for Linear Algebra Kernels

Toufik Baroudi
t.baroudi@univ-batna2.dz
Department of computer science
University of Batna 2
Batna, Algeria

Vincent Loechner
loechner@unistra.fr
INRIA and ICube Laboratory,
University of Strasbourg
Strasbourg, France

Rachid Seghir
r.saghir@univ-batna2.dz
Lastic Laboratory
University of Batna 2
Batna, Algeria

Abstract

The polyhedral model permits to automatically improve data locality and enable parallelism of regular linear algebra kernels. In previous work we have proposed a new data structure, 2d-packed layout, to store only the non-zeros elements of regular sparse (triangular and banded) matrices dynamically allocated for different basic linear algebra operations, and used Pluto to parallelize and optimize them. To our surprise, there were huge discrepancies in our measures of these kernels execution times that were due to the allocation mode: as statically declared arrays or as dynamically allocated arrays of pointers.

In this paper we compare the performance of various linear algebra kernels, including some linear algebra kernels from the PolyBench suite, using different array allocation modes. We present our detailed investigation of the possible reasons of the performance variation on two different architectures: a dual 12-cores AMD (Magny-Cours) and a dual 10-cores Intel Xeon (Haswell-EP).

We conclude that static or dynamic memory allocation has an impact on performance in many cases, and that the processor architecture and the gcc compiler's decisions can provoke significant and sometimes surprising variations, in favor of one or the other allocation mode.

Keywords Dynamic/static memory allocation, polyhedral model, linear algebra kernels, performance

1 Introduction

To fully use high performance computing, programmers must parallelize their applications. This task is far from trivial, hence the need to automate this process. This automation is partially handled by tools that analyze the code to exploit parallelism, using compiler or runtime optimization techniques. Several tools of automatic optimization and parallelization are proposed in the literature. Among these tools, Bondhugula et al. [5, 6] developed a source to source framework based on the polyhedral model called Pluto. Pluto allows to transform an input C source code into a semantically equivalent output C code that achieves parallelism and data locality.

IMPACT 2020, January 22, 2020, Bologna, Italy

Linear algebra (in particular, multiplication of two matrices) lies at the heart of many calculations in scientific computing [1]. Optimizing this kind of calculations is one of the research topics that have attracted the scientists interest over the past years. In order to show the efficiency of parallelization in high performance computing, many researchers are focusing on sparse matrices by suggesting different data structures to store the non-zero elements of this type of matrices [4, 9, 10, 12]. In our previous work [3], we have suggested a new approach to optimize triangular and banded matrix operations by using a dense and regular 2-dimensional data structure dynamically allocated for sparse matrix storage. To our surprise, we measured huge discrepancies in the execution time of different versions of those codes that were caused only by the memory allocation type: as static declared arrays or as dynamically allocated arrays of pointers.

In this paper we compare several matrix computation kernels using static and dynamic memory allocation, both in original (sequential) form and in optimized and parallelized form generated by the Pluto compiler. We show that allocating the matrices dynamically or allocating them statically has a noticeable influence in the performance of the resulting codes. We explore the possible causes of these variations using the hardware performance counters, on two different architectures.

The remainder of the paper is organized as follows: in Section 2, a related work is given. Section 3 gives a quick overview of our previous work on 2d-packed layouts for sparse triangular matrices. We present a performance comparison between dynamic and static allocation on these kernels in Section 4. Section 5 focuses on the precise study of the matrix multiplication kernel. The effect of the number of vectorized instructions is studied in Section 6 for all the 2d-packed benchmarks. Another experimental study is provided in Section 7 for the linear algebra kernels from the PolyBench suite. Finally, this work is concluded in Section 8.

2 Related work

Linear Algebra is a branch of mathematics that deals with the study of vectors, matrices and linear equations. In addition to mathematics, engineering and science, linear algebra has extensive applications in natural as well as social sciences.

It has been shown that the optimization of sparse matrix operations significantly enhances the performance of many scientific and engineering applications [7, 11, 15, 17]. Many manual optimizing techniques targeting matrix computation kernels have been proposed [8, 13, 19].

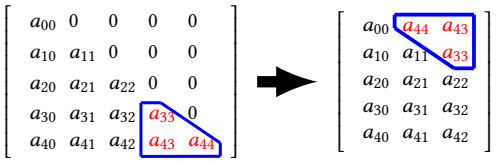
The automatic optimization and parallelization techniques consist of creating and implementing tools and compilers that are able to translate a given code into an optimized and parallelized code [14]. Pluto [2, 5, 6] is an example of automatic optimization and parallelization source-to-source compiler that can be used to this purpose.

Many new data formats for sparse matrices have been proposed to improve the performance of Linear Algebra benchmarks. Among these works, Gustavson et al. [12] propose a compact way to store triangular, symmetric and Hermitian matrices called Rectangular Full Packed Format (RFPF). In order to obtain better computation and memory performances, the authors propose to store only the non-zero elements of triangular matrices in a compact data structure, where the size of the allocated space is N×(N+1)/2. To illustrate the benefits of this RFPF format, they have compared the performance of different Cholesky Algorithms by using RFPF versus the LAPACK library routines. In our previous work [3] on 2d-packed layouts we have improved their storage format, in order to perform a unique data transformation for both odd and even ordered matrices: in the Gustavson et al.'s proposal, odd and even ordered matrices are stored differently, which requires the resulting code to distinguish between these two cases and the generated code to nest many if-then-else structures. We have also proposed a packed storage format for banded and triangular-banded matrices and evaluated their performance on Cholesky and on three extra kernels: MatMul, SolveMat and sspfa.

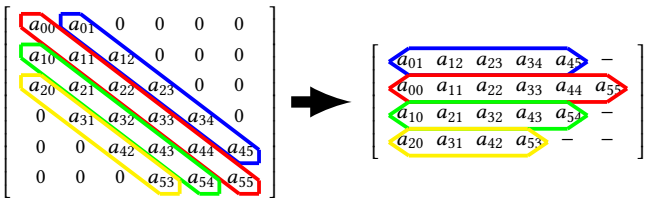
There has been little attention drawn so far at performance issues resulting from static versus dynamic memory allocation. In IMPACT'19, Shirako and Sarkar [18] proposed to integrate affine data layout transformations to polyhedral compilers. In their experiments they compared the original program, the Pluto optimized program and their version of the code with dynamic memory allocation. They provided their version of the code and its memory allocation, however, they did not state whether the original and the Pluto optimized programs performed static or dynamic memory allocation, and if they isolated this effect. If the allocation type is different for the various code versions they compared (see Table 1 of their paper) then their results could have been altered by its effect on performance that we expose here.

3 Overview of the 2d-packed format transformation

Our proposal of 2d-packed layout [3] is quickly described hereunder in three parts: the input, the transformation function for the new storage format and the output.



(a) 2d-packed transformation for triangular matrices



(b) 2d-packed transformation for banded matrices

Figure 1. New data structures for different sparse matrices

```
for (i = 0; i < n; i++) {
  for (j = 0; j <= i; j++) {
    if (2*j <= n)
      C[i] += A[i][j] * V[j];
    if (2*j > n)
      C[i] += A[n-i-1][n-j] * V[j];
  }
}
```

Figure 2. Matrix-vector multiplication code with matrix A in 2d-packed format

3.1 Input

Our proposal takes as input (i) the original code to be transformed and (ii) the identification of the sparse matrices to be transformed. We have focused on three types of regular sparse matrices: triangular, banded and banded-triangular matrices.

3.2 2d-packed transformation function

The 2d-packed transformation consists in two steps: (i) we store the non-zero elements of the sparse matrix in a new 2-dimensional dense data array, using the 2d-packed layout (depending on the sparsity type). Figure 1 illustrates the 2d-packed data structure for different sparse matrix types. Then, (ii) the original code is transformed into a new code accessing the 2d-packed data structure, eventually adding tests into the code to distinguish between the different matrices parts.

3.3 Output

The automatically generated code is a SCoP which can be fed to Pluto to obtain an optimized and parallelized C program, equivalent to the original program manipulating dense matrices. The 2d-packed code of the matrix vector multiplication kernel for a triangular matrix A is shown in Figure 2.

```
// dynamic allocation for triangular matrices
// in 2d-packed format
double **allocTriPackMat(int n)
{
  int i;
  double **mat = malloc(n*sizeof(double *));
  for (i=0; i<n/2; i++)
      mat[i]=calloc((n+1)/2,sizeof(double));
  for (; i<n; i++)
      mat[i]=calloc(((n/2)+1),sizeof(double));
  return mat;
}
```

Figure 3. Matrix dynamic allocation code

4 Dynamic allocation versus static allocation in 2d-packed layout

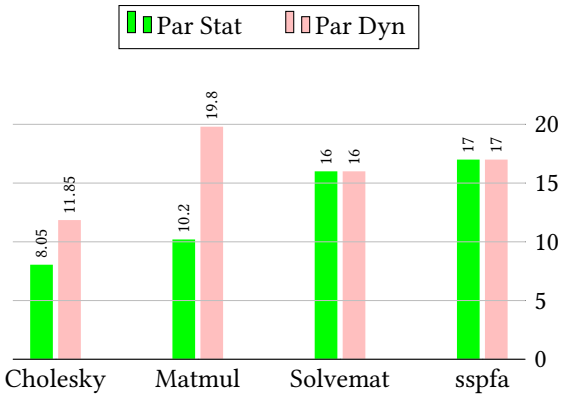
We have tried various allocation modes for 2d-packed matrices of `double`'s: as statically declared global variables, stack allocated arrays, dynamically allocated 2d arrays and dynamically allocated arrays of pointers to dynamically allocated rows. Our results are summarized hereafter as a comparison between "*static allocation*" as statically declared global array and "*dynamic allocation*" as an array of pointers. The stack allocated arrays results are very similar in performance to the static allocation, and the dynamically allocated 2d arrays results very similar to the dynamic allocation, so it did not make much sense to present all four of them.

In the dynamic allocation, we store triangular matrices in 2d-packed format using an array of pointers of size N. The data itself is stored in the dynamically allocated rows, with the addition of an extra column of size N/2 when N is even. The allocation code for triangular matrices in 2d-packed format is shown in Fig. 3. We also tried to allocate the rows in a single `calloc` call, and it did not have much influence on the performance of the resulting benchmarks. The advantage of this row allocation is that each row pointer is aligned such that it can be used for any data type, including vector types.

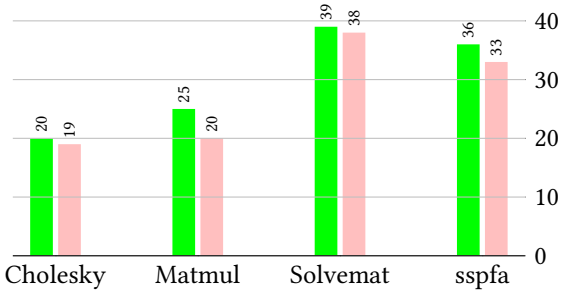
In the static allocation, we have stored the triangular matrices in 2d-packed format in statically declared global arrays of size $N \times ((N/2) + 1)$.

4.1 Execution times and performance

We have applied our approach to optimize and parallelize four double-precision linear algebra kernels using triangular matrices: Cholesky factorization, multiplication of two matrices (MatMul), matrix-matrix solver (SolvMat, see annex A) and sspfa factorization (annex B). We have compared the execution times of the automatically optimized and parallelized kernels in 2d-packed format using static allocation to the ones using dynamic allocation. All reported results are obtained from an average of five executions of each kernel using matrices of size $N = 8000$. The allocation time was not



(a) Execution time for config1 machine (seconds)



(b) Execution time for config2 machine (seconds)

Figure 4. Comparison of the execution time between static and dynamic allocations

included in the time measures. The experiments were run on two different computers:

- **config1** is a dual socket Intel Xeon E5-2650 v3 (Haswell-EP) of 2x10 hyperthreaded cores, with AVX2 (256 bits) support;
- **config2** is a dual socket AMD Opteron 6172 (Magny-Cours) of 2x12 cores, with SSE (128 bits) support.

Both configurations run the exact same system: Ubuntu (bionic) with a Linux 4.0.15 standard kernel. No huge page support was enabled. We have compiled our codes on both configurations using the exact same gcc version (7.3.0), with options `-O3 -march=native -fopenmp`. We have used Pluto 0.11.4 with options `--tile --parallel` to optimize and parallelize the programs, using the default tile sizes and no other fine tuning option.

Figure 4 shows the execution times obtained on our config1 and config2 machines. In both subfigures, each couple of bars represent the execution times with static and dynamic allocation respectively. In the first configuration (config1, Fig. 4a) one can notice that the execution times in the two cases are almost the same for sspfa and Solvemat, whereas for Cholesky and MatMul the static allocation performs better than the dynamic allocation. In the second configuration (config2, Fig. 4b) the dynamic allocation leads to better

performance compared to the static allocation for all those benchmarks.

There are remarkable discrepancies between those execution times depending on whether the arrays are statically or dynamically allocated. The compiler is the same, the kernel is the same, so what might be the reasons for such differences? To answer this question we have performed a set of further experiments on the MatMul benchmark, since it is the one which presents important differences in execution times and a complete opposite effect of the memory allocation type on our two test platforms. The following section presents a detailed performance analysis of the MatMul benchmark using the two memory allocation modes.

5 Experimental study of the matrix multiplication in 2d-packed format

The original algorithm of multiplication of two lower triangular square matrices of the same order N is given in Algorithm 1.

Algorithm 1: Lower triangular matrices multiplication

Input: Two lower triangular matrices A and B of order N

```
1 for i ← 0 to N − 1 do
2 |   for k ← 0 to i do
3 |   |   for j ← 0 to k do
4 |   |   |   C[i][j] ← C[i][j] + A[i][k] ∗ B[k][j];
5 |   |   end
6 |   end
7 end
```

In our study, we have used two variants of the matrix multiplication in 2d-packed format, where the input matrices A and B as well as the output matrix C are stored in 2d-packed data structures. The first variant of the matrix multiplication code is obtained applying the raw 2d-packed transformation algorithm. The resulting code, named C1, is shown in Fig. 5. This code is only composed of one perfect loop nest in which the innermost loop contains three tests. The second variant, named C2, is an equivalent code in which the tests were removed thanks to loop splitting. The resulting code, shown in Fig. 6, is a non-perfect loop nest containing no test. In the following subsections, we will compare the performance of the static and dynamic allocations for both codes C1 and C2, by measuring their execution time, number of executed instructions, number of cache loads, number of cache misses, number of TLB misses and number of vectorized instructions. These comparisons are illustrated on Figures 7, 8 and 9. In all these experiments, the four bars represent respectively the original code with static allocation, the original code with dynamic allocation, the code optimized and parallelized

```
const int nn=n/2;
for (i=0 ; i<n ; i++) {
  for (k=0 ; k<=i ; k++) {
    for (j=0 ; j<=k ; j++) {
      if (i>nn && j>nn && k>nn)
        C[n-i-1][n-j] += A[n-i-1][n-k] * B[n-k-1][
            n-j];
      if (i>nn && j<=nn && k>nn)
        C[i][j] += A[n-i-1][n-k] * B[k][j];
      if (j<=nn && k<=nn)
        C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Figure 5. C1 code: triangular matrix multiplication in 2d-packed format

```
const int nn=n/2;
for(i=0; i<nn+1; i++) {
  for(k=0; k<=i; k++) {
    for(j=0; j<=k; j++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
for(i=nn+1; i<n; i++) {
  for(k=0; k<=i && k<nn+1; k++) {
    for(j=0; j<=k; j++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
  for(k=nn+1; k<=i; k++) {
    for(j=0; j<=k && j<nn+1; j++) {
      C[i][j] += A[n-(i)-1][n-(k)] * B[k][j];
    }
    for(j=nn+1; j<=k; j++) {
      C[n-(i)-1][n-(j)] += A[n-(i)-1][n-(k)] * B[n
          -(k)-1][n-(j)];
    }
  }
}
```

Figure 6. C2 code: triangular matrix multiplication after loop splitting in 2d-packed format

by Pluto with static allocation and the code optimized and parallelized by Pluto with dynamic allocation.

In Figure 7, looking individually at each group of bars and comparing the first to the third, and the second to the last, we see that optimizing and parallelizing our codes using Pluto is always beneficial. For code C1 (with tests), Figure 8 confirms that applying Pluto reduces the number of executed instructions, L1 cache loads, cache misses and TLB misses.

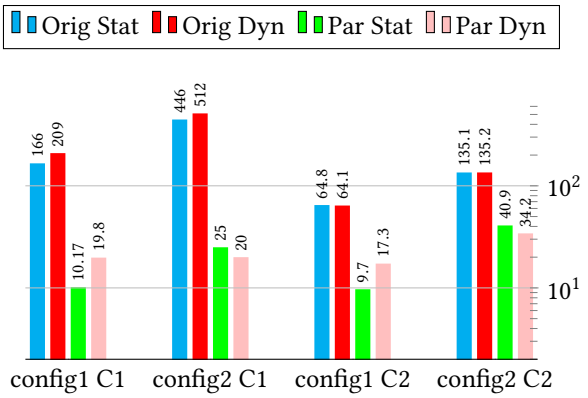


Figure 7. Execution time (in seconds, logscale) for static and dynamic allocations, on config1 and config2, for C1 and C2 codes

Applying Pluto to code C2 (without tests) also reduces the number of cache misses, but it increases the total number of executed instructions and the number of cache loads, probably as an effect of tiling and parallelization. The next subsections present a detailed analysis of those graphs to compare static and dynamic allocations.

5.1 Execution time

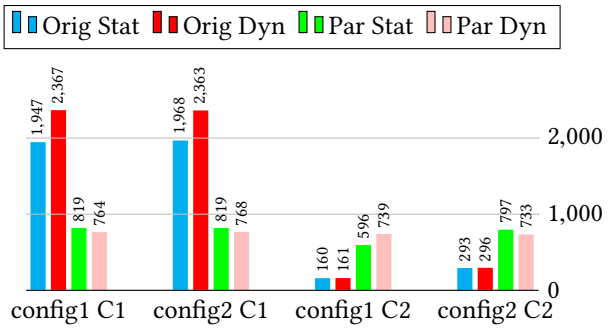
Comparing each couple of bars in Figure 7 we observe that the difference in execution time between the static and the dynamic version varies from 0.80× up to 1.95×. In the original (sequential) C2 code, static or dynamic allocation does not influence the execution times (first two bars in the two right side groups of bars). In the other versions (original and parallel) and configurations (config1 or config2), static allocation performs better than dynamic allocation, except for the Pluto-optimized code on config2: the dynamic version performs better than the static version (e.g. 20s *vs.* 25s for code C1).

5.2 Total number of instructions

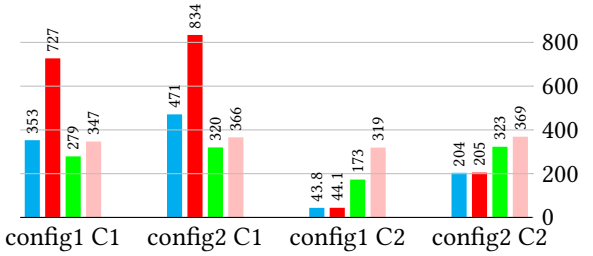
In Figure 8a, one can notice that the number of executed instructions of code C1 is similar on both configurations. For example: Instr (orig-static, config1, C1) ≃ Instr (orig-static, config2, C1) ≃ 1,950 Billions instructions. For code C2, it is higher on config2 than on config1, which means that the compiler did certainly generate different codes from C2 on those two different architectures.

For code C1, dynamic allocation increases the number of executed instructions in the original code, while it is reduced on the Pluto code. But those results do not correlate to the execution time.

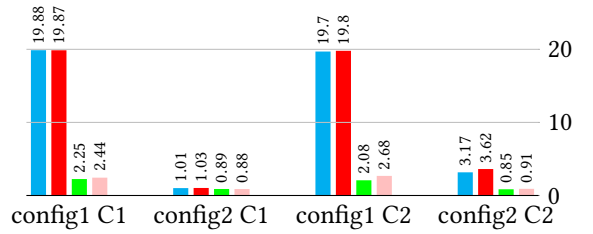
For code C2, the number of executed instructions increases when it is optimized and parallelized by Pluto. The original code is composed of three nested imperfect loops, and Pluto generates a more complex tiled and parallelized code



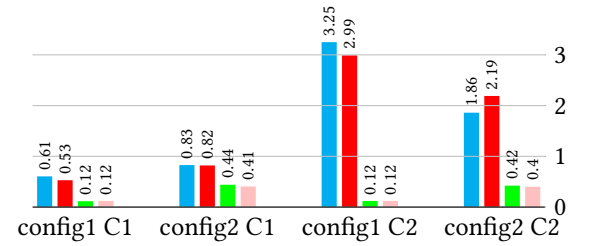
(a) Number of executed instructions (Billions)



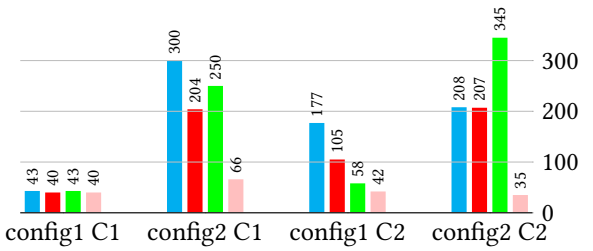
(b) Number of L1-dcache-loads (Billions)



(c) Number of L1-dcache-misses (Billions)



(d) Number of L3-cache-misses (Billions)



(e) Number of dTLB-misses (Millions)

Figure 8. Performance counters for static and dynamic allocations, on config1 and config2, for C1 and C2 codes

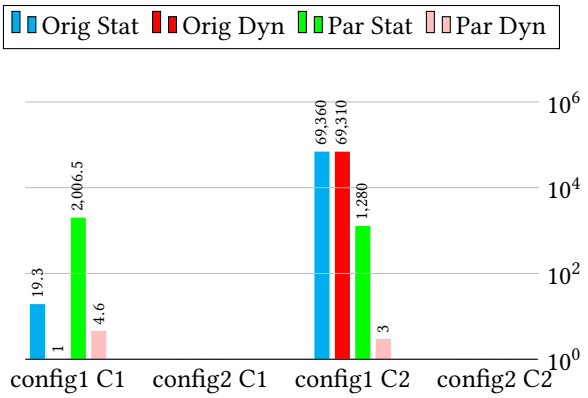


Figure 9. Number of vectorized instructions (Millions, logscale) for static and dynamic allocations on config1 for C1 and C2 codes

of imperfect loops of depth six. The number of executed instructions in dynamic allocation is greater than the one of static allocation on config1, probably due to the extra control and the extra pressure on register allocation of this complex code. This effect is the opposite on the other versions.

Overall, these measures show that the compiler seems to take different decisions on our two different test architectures. There is some correlation between the difference in number of executed instructions and the difference in performance between static and dynamic allocation shown in Fig. 7 except for the optimized version of config1/C1.

5.3 Number of L1-dcache-loads

Figure 8b shows the number of L1-dcache-loads: this feature can be used to determine how many times the load ports are used. In all the cases in this experiment, the number of L1-dcache-loads in the dynamic allocation is greater than the one of the static allocation. The reason for this is the use of pointers which obviously leads to more memory accesses in order to access data.

For the comparison between static and dynamic memory allocation, the number of L1-dcache-loads can be somehow correlated to the execution time on config1 but not for the config2 parallel versions, where the execution time in dynamic allocation is better than the one of static allocation.

5.4 Number of L1 and L3 cache-misses

The number of L1 and L3 cache-misses are shown in Figure 8c and 8d.

There is sometimes a very important ratio of cache-misses compared to the total number of loads (up to $19.7/43.8 = 45\%$ L1-dcache-misses for config1/C2 with static allocation). As expected, in the Pluto optimized codes the cache-misses are substantially diminished in most cases (e.g. $2.08/173 = 1.2\%$ in the same example).

The second observation is that the sum of the number of L1 and L3 cache-misses slightly correlates with the execution times in the Pluto optimized versions, except for the config2/C2 version. However, in the original versions of the codes it is not true.

5.5 Number of TLB misses

Counter-intuitively, Figure 8e shows that in all cases there are less TLB misses in the dynamic allocation versions than in the static ones: an extra memory access to an array of pointers reduces the overall number of TLB misses! However, the TLB replacement policy and the order of the memory accesses play an important and complex role, so their interaction is hardly predictable.

The number of TLB misses does not seem to correlate with the execution time results (Fig.7). However, this is the only measure where a significant difference between the static and the dynamic versions for the Pluto optimized code on config2 could explain their difference in performance.

5.6 Number of vectorized instructions

In this last experiment, we measured the number of vectorized instructions on config1 only, since this performance counter is not available on our oldish config2 processor. One can clearly notice from Figure 9 that the C2 original code is well vectorized, while the other versions are only partially vectorized at best. Since the C1 code contains tests in the innermost loops, it was more difficult for the compiler to discover vectorization opportunities in this code. In both cases, there are more vectorized instructions in the static allocation versions than in the dynamic one.

We believe that this is one of the reasons of the differences reported earlier, and it explains why the static allocation on config1 is significantly better than its dynamically allocated version. However, on config2, activating option `-fopt-info-vec` in gcc did not show any difference between static and dynamic memory allocation, so those codes are vectorized in the same way probably without much impact on performance.

5.7 Synthesis

The conclusion of these measures is that it is very difficult to get a simple explanation on the reason why some codes with static allocation perform better than the ones with dynamic allocation, or the opposite. Our best hypothesis is that the overall performance (Fig. 7) can be deduced from a combination of (1) the total number of memory accesses (2) the number of cache and TLB misses and (3) the number of vectorized instructions.

On config1 however, the number of vectorized loops is probably the main factor affecting performance.

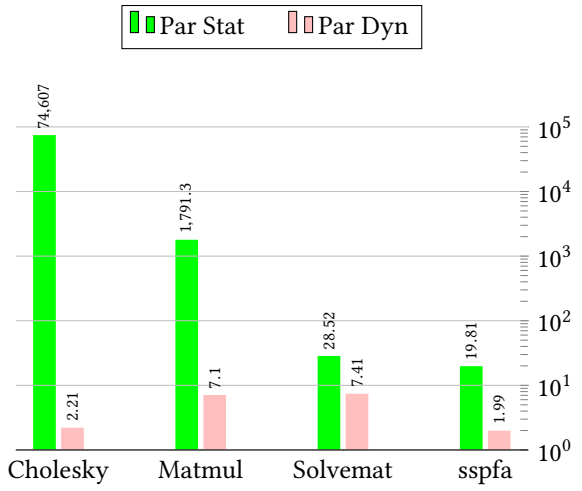


Figure 10. Number of vectorized instructions (Millions, logscale) for static and dynamic allocations on config1 for the 2d-packed benchmarks

6 Number of vectorized instructions in all 2d-packed benchmarks

To confirm this hypothesis, we compared the number of vectorized instructions on config1 of static and dynamic allocations in the four benchmarks presented earlier (Fig. 4a). The results are shown in Figure 10. We can clearly see that the number of vectorized instructions in static allocation is always greater than the one in dynamic allocation (ex: Nbr_vec_instr (Cholesky, static)=74,607 Million instructions, Nbr_vec_instr (Cholesky, dynamic)=2.21 Million instructions). But it is very significant in the first two benchmarks (Cholesky and Matmul) and these are the ones presenting the most performance variation in Fig. 4a. From all the previous experiments, we conclude that on config1 the number of vectorized instructions is probably one of the main factor affecting performance of the benchmarks studied till now.

In the next section, we will focus on the Polyhedral Benchmark suite (PolyBench) [16] and check whether the number of vectorized instructions is also affecting the performance of those benchmarks in static and in dynamic memory allocations.

7 Dynamic allocation versus static allocation in PolyBench

We have studied the difference in performance of the two memory allocation modes by considering eight linear algebra kernels from the PolyBench suite: `2mm`, `3mm`, `atax`, `bicg`, `mvt`, `trisolv`, `lu` and `cholesky`. All the experiments are performed on our config1 machine, based on the Intel Xeon (Haswell-EP). The matrix sizes used in these experiments are 2,000 for the $O(N^3)$ algorithms (`2mm`, `3mm`, `cholesky` and `lu`), and 20,000 for the $O(N^2)$ ones (`atax`, `bicg`, `mvt` and `trisolv`).

PolyBench provides a C macro for declaring its data as stack allocated arrays (declared in the main function) or as heap-allocated multidimensional arrays (with a `posix_memalign` call, the default). We compiled all programs with gcc version 7.3.0 using options `-O3 -march=native -fopenmp` for the dynamic versions, and adding option `-DPOLYBENCH_STACK_ARRAYS` for the static versions.

We have also tried to add the options (`-DUSE_RESTRICT` and `-DC99_PROTOS`) to use the `restrict` keyword and inform the compiler that the arrays do not alias. This did not have any significant effect on the performance: the only gain was to spare a test to protect each potentially aliased region (as reported by `gcc -fopt-info-vec`), but since these tests were never taken they were correctly predicted and harmless in our experiments.

7.1 Measurements

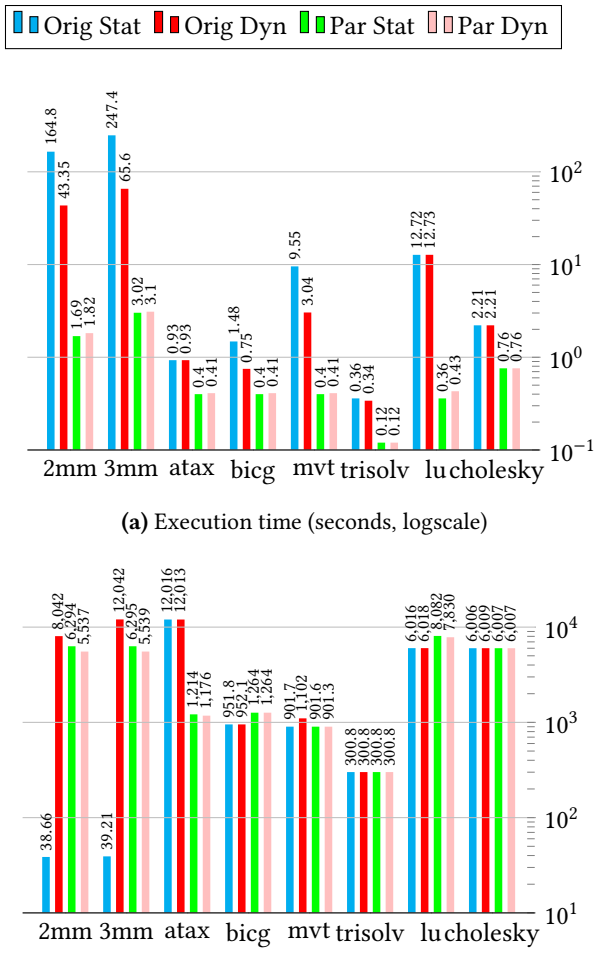
The first plot in Fig. 11a shows the execution times of the different benchmarks. In regard of this figure, we plotted the number of vectorized instructions in Fig. 11b.

The execution times of all the optimized and parallelized benchmarks (right-hand side couples of bars) are very similar whether the arrays are statically or dynamically allocated. A noticeable exception is `lu`: 0.36s for the static version versus 0.43s for the dynamic one, a 20% slowdown. On Fig. 11b, the difference between the number of vectorized instructions for these two executions is only of 3%, so the main reason of this gap is probably not vectorization. The other exception is `2mm`, where the 8% difference in execution time can be imputed to a 12% difference in the number of vectorized instructions.

For the sequential original codes (left-side couples of bars), the execution times are very close for `atax`, `trisolv`, `lu` and `cholesky`. But the other four benchmarks (`2mm`, `3mm`, `bicg` and `mvt`) show significant differences: the dynamic allocation performs better than the static allocation. The number of vectorized instructions is much better for the dynamically allocated versions of `2mm` and `3mm`. This can clearly explain the difference for those two benchmarks. The difference in number of vectorized instructions in `mvt` is 22%, not enough to explain a 3× difference in execution time. For `bicg`, there is no difference. So, for `bicg` and `mvt`, we need to find another explanation than vectorization for those speedups.

At this point, there is no explanation for the difference in performance between static and dynamic allocation of three benchmarks: the Pluto version of `lu` and the original versions of `bicg` and `mvt`.

The other performance counters are shown in Fig. 12. The difference in performance for the original versions of `bicg` and `mvt` can be correlated to the number of L1-dcache-loads from Fig. 12b and to the number of cache misses from Fig 12c and 12d. The number of total loads is higher of 64% for `bicg` and 38% for `mvt` in the static allocation versions, and the



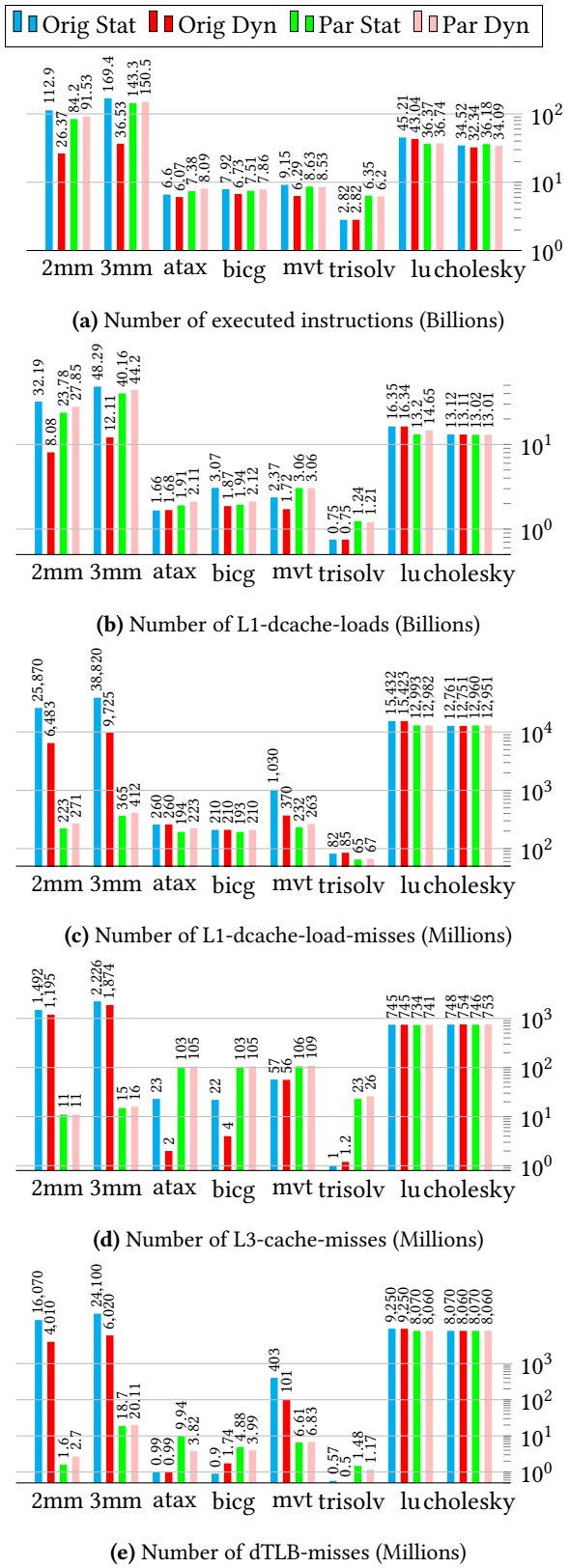
(a) Execution time (seconds, logscale)

(b) Number of vectorized instructions (Millions, logscale)

Figure 11. Execution time and number of vectorized instructions, for static and dynamic allocations in PolyBench

same kind of difference is observable on the two original benchmarks `2mm` and `3mm`.

For the parallel `lu` this difference is of 12.5%, which could also partially explain the difference in execution time for this benchmark. But the other parallelized benchmarks also show a significant difference in their number of loads that did not have any impact on their execution times! The dynamic allocation versions have around 10% more memory loads than the static versions, up to 17% for `2mm`. However, this is probably a consequence of the static allocation versions having a bit more vectorized instructions than the dynamic ones, as seen on Fig. 11b. So for the Pluto dynamic allocation versions, there are more executed vectorized instructions but the pressure on memory is higher, which combined together results in the same overall performance. There is no satisfactory explanation in those measurements for the difference in performance between static and dynamic allocation of the Pluto optimized `lu` benchmark.



(a) Number of executed instructions (Billions)

(b) Number of L1-dcache-loads (Billions)

(c) Number of L1-dcache-load-misses (Millions)

(d) Number of L3-cache-misses (Millions)

(e) Number of dTLB-misses (Millions)

Figure 12. Comparison of different performance counters between static and dynamic allocation for PolyBench

7.2 Analysis

Out of eight benchmarks in original and in Pluto optimized forms, six show significant performance variations either in original or in Pluto-optimized form between static allocation and dynamic allocation. The differences are mainly due to the number of vectorized instructions and to the number of memory loads and misses.

Something that is difficult to explain is the reason why accessing static allocated matrices provoke more total memory accesses (and misses) than dynamic ones in the original versions of the codes, for 4 out of 8 benchmarks. In the dynamic allocated version there is a potential extra memory access for each performed array access; it could be cached by temporal reuse through a register if the arrays are accessed line by line, but it could hardly be of opposite effect on the total number of memory accesses. Vectorization could explain it, since the memory accesses are packed together when being accessed as a vector, but this can happen only in 2 out of 4 of these benchmarks. Another possibility is that this could be a side effect of a higher pressure on the register allocation, resulting in a non optimal decision of the gcc compiler and finally in more memory accesses. That effect could inverse in the Pluto versions of the codes since the Pluto optimized versions show more obvious temporal locality that could be exploited by the compiler. Further investigations should be made to evaluate the pressure on register allocation and possibly validate this hypothesis.

As a conclusion, in the original versions of the code, static memory allocation versions perform better than the dynamic ones in 50% of our benchmarks on our config1 experimental platform. On config2, we observed the opposite. In addition to our first series of benchmarks on the 2d-packed codes, those results on PolyBench show that the best performing code after optimization by Pluto seems unpredictable: sometimes the static allocation performs better thanks to vectorization (as in `Cholesky` and `Matmul` on config1 in the first series of experiments) and sometimes the dynamic allocation one performs better (as on config2 or in `lu` or `2mm` on config1 in the second series of experiments).

8 Conclusion

We have shown in this paper that the memory allocation type, as static declared array or dynamically heap allocated array, has a significant impact on the performance of some linear algebra kernels. This happens both on sequential codes and on automatically parallelized and optimized codes. In our experimental study, the identified potential causes of these performance variations are (1) the ability of the compiler to vectorize the kernels, and (2) the variation of the number of performed memory accesses and cache misses in the generated code. The variation of performance can be alternatively in favour of the static allocation mode or the dynamic one, and can even flip on different target processor architectures in an unpredictable manner!

Further experiments could be conducted in future work on different processor architectures, different operating systems, using different compilation options and different compilers to confirm the possible causes of the performance variation due to memory allocation.

Although this work is not explicitly about polyhedral compilation, we believe that our research community should be thoughtful of this effect, and take a particular care about memory allocation when running benchmarks to compare the performances of various versions of codes, especially when modifying the way arrays are declared and allocated. In particular, all research on data layout transformations (e.g. [18]) should be very careful for their benchmarks not to be perturbed by the sometimes surprising side effect of static versus dynamic memory allocation on performance.

References

[1] Äke Björck. 2015. *Numerical Methods in Matrix Computations.* Springer International Publishing.

[2] Athanasios Athanasios Konstantinidis and Paul H. J. Kelly. 2011. More Definite Results from the PluTo Scheduling Algorithm. In *1st International Workshop on Polyhedral Compilation Techniques (IMPACT)*, C. Alias and C. Bastoul (Eds.). Chamonix, France. http://perso.ens-lyon.fr/christophe.alias/impact2011/impact-02.pdf

[3] Toufik Baroudi, Rachid Seghir, and Vincent Loechner. 2017. Optimization of Triangular and Banded Matrix Operations Using 2d-Packed Layouts. *ACM Trans. Archit. Code Optim.* 14, 4, Article 55 (Dec. 2017), 19 pages. https://doi.org/10.1145/3162016

[4] Nathan Bell and Michael Garland. 2008. *Efficient Sparse Matrix-Vector Multiplication on CUDA.* NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation.

[5] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. 2008. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *International Conference on Compiler Construction (ETAPS CC).*

[6] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[7] Aydin Buluc and John R. Gilbert. 2008. Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication. In *Proceedings of the 2008 37th International Conference on Parallel Processing (ICPP '08).* IEEE Computer Society, Washington, DC, USA, 503–510. https://doi.org/10.1109/ICPP.2008.45

[8] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. 2009. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. *Parallel Comput.* 35, 1 (Jan. 2009), 38–53. https://doi.org/10.1016/j.parco.2008.10.002

[9] Huimin Cui, Jingling Xue, Lei Wang, Yang Yang, Xiaobing Feng, and Dongrui Fan. 2012. Extendable Pattern-oriented Optimization Directives. *ACM Trans. Archit. Code Optim.* 9, 3, Article 14 (Oct. 2012), 37 pages. https://doi.org/10.1145/2355585.2355587

[10] Huimin Cui, Qing Yi, Jingling Xue, and Xiaobing Feng. 2013. Layout-oblivious Compiler Optimization for Matrix Computations. *ACM Trans. Archit. Code Optim.* 9, 4, Article 35 (Jan. 2013), 20 pages. https://doi.org/10.1145/2400682.2400694

[11] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. 2007. High-performance Graph Algorithms from Parallel Sparse Matrices. In *Proceedings of the 8th International Conference on Applied Parallel Computing: State of the Art in Scientific Computing (PARA'07)*. Springer-Verlag, Berlin, Heidelberg, 260–269. http://dl.acm.org/citation.cfm?id=1775059.1775097

[12] Fred G. Gustavson, Jerzy Waśniewski, Jack J. Dongarra, and Julien Langou. 2010. Rectangular Full Packed Format for Cholesky's Algorithm: Factorization, Solution, and Inversion. *ACM Trans. Math. Softw.* 37, 2, Article 18 (April 2010), 21 pages. https://doi.org/10.1145/1731022.1731028

[13] Ziang Hu, Juan del Cuvillo, Weirong Zhu, and Guang R. Gao. 2006. *Optimization of Dense Matrix Multiplication on IBM Cyclops-64: Challenges and Experiences.* Springer Berlin Heidelberg, Berlin, Heidelberg, 134–144. https://doi.org/10.1007/11823285_14

[14] G. M. Megson and X. Chen. 1997. *Automatic Parallelization for a Class of Regular Computations.* World Scientific Publishing Co., Inc., River Edge, NJ, USA.

[15] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2016. Adaptive Multi-level Blocking Optimization for Sparse Matrix Vector Multiplication on GPU. *Procedia Computer Science* 80 (2016), 131 – 142. https://doi.org/10.1016/j.procs.2016.05.304

[16] L.N. Pouchet. 2015. *Polybench/c 4.1: The polyhedral benchmark suite.* Technical Report. http://polybench.sourceforge.net

[17] Y. Saad. 2003. *Iterative Methods for Sparse Linear Systems* (2nd ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[18] Jun Shirako and Vivek Sarkar. 2019. Integrating Data Layout Transformations with the Polyhedral Model. In *Proceedings of International Workshop on Polyhedral Compilation Techniques (IMPACT'19)*, D. Wonnacott and O. Zinenko (Eds.). Valencia, Spain. http://impact.gforge.inria.fr/impact2019/papers/IMPACT_2019_paper_8.pdf

[19] Vasily Volkov and James W. Demmel. 2008. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC '08)*. IEEE Press, Piscataway, NJ, USA, Article 31, 11 pages. http://dl.acm.org/citation.cfm?id=1413370.1413402

A SolveMat source code

```
void SolveMat(int n, double **A, double **B, double **Y)
{
  int i, j, k;
  for (k = 0; k < n; k++) {
    for (i = k; i < n; i++) {
      for (j = 0; j < i; j++) {
        if (j >= k)
          B[i][k] = B[i][k] - A[i][j] * Y[j][k];
      }
      Y[i][k] = B[i][k] / A[i][i];
    }
  }
}

void SolveMat_2dp(int n, double **A, double **B, double **Y)
{
  int i, j, k;
  #pragma scop
  for (k = 0; k < n; k++) {
    for (i = k; i < n; i++) {
      for (j = 0; j < i; j++) {
        if (j >= k) {
          if ((2 * j > n) && (2 * k > n))
            B[n-(i)-1][n-(k)] = B[n-(i)-1][n-(k)] - A[n-(i)-1][n-(
                j)] * Y[n-(j)-1][n-(k)];
          if ((2 * j <= n) && (2 * k > n))
            B[n-(i)-1][n-(k)] = B[n-(i)-1][n-(k)] - A[i][j] * Y[j
                ][k];
          if ((2 * j > n) && (2 * k <= n))
            B[i][k] = B[i][k] - A[n-(i)-1][n-(j)] * Y[j][k];
          if ((2 * i > n) && (2 * j <= n) && (2 * k <= n))
            B[i][k] = B[i][k] - A[i][j] * Y[j][k];
          if (2 * i <= n)
            B[i][k] = B[i][k] - A[i][j] * Y[j][k];
        }
      }
      if (2 * k > n)
        Y[n-(i)-1][n-(k)] = B[n-(i)-1][n-(k)] / A[n-(i)-1][n-(i)];
      if ((2 * i > n) && (2 * k <= n))
        Y[i][k] = B[i][k] / A[n-(i)-1][n-(i)];
      if (2 * i <= n)
        Y[i][k] = B[i][k] / A[i][i];
    }
  }
  #pragma endscop
}
```

B Sspfa source code

```
void sspfa(double **m, int n)
{
  int i, j, k;
  for (j = 0; j < n; j++) {
    for (k = 0; k < j; k++) {
      for (i = 0; i < k; i++) {
        m[j][k] = m[j][k] - m[j][i] * m[k][i];
      }
    }
  }
}


void sspfa_2dp(double **m, int n)
{
  int i, j, k;
  #pragma scop
  for (j = 0; j < n; j++) {
    for (k = 0; k < j; k++) {
      for (i = 0; i < k; i++) {
        if ( 2 * i > n && 2 * k > n)
          m[n-(j)-1][n-(k)] = m[n-(j)-1][n-(k)] - m[n-(j)-1][n-(i)
              ] * m[n-(k)-1][n-(i)];
        if ( 2 * i <= n && 2 * k > n)
          m[n-(j)-1][n-(k)] = m[n-(j)-1][n-(k)] - m[j][i] * m[k][i
              ];
        if ( 2 * i > n && 2 * k <= n)
          m[j][k] = m[j][k] - m[n-(j)-1][n-(i)] * m[n-(k)-1][n-(i)
              ];
        if ( 2 * i <= n && 2 * k <= n)
          m[j][k] = m[j][k] - m[j][i] * m[k][i];
      }
    }
  }
  #pragma endscop
}
```