Memory Hierarchy Behavior Study during the Execution of Recursive Linear Algebra Library

I. Šimeček

For good performance of every computer program, good cache and TLB utilization is crucial. In numerical linear algebra libraries (such as BLAS or LAPACK), good cache utilization is achieved by explicit loop restructuring (mainly loop blocking), but this requires difficult memory pattern behavior analysis. In this paper, we represent the recursive implementation ("divide and conquer" approach) of some routines from numerical algebra libraries. This implementation leads to good cache and TLB utilization with no need to analyze the memory pattern behavior due to "natural" partition of data.

Keywords: numerical linear algebra, code restructuring, loop unrolling, recursive implementation, memory hierarchy utilization.

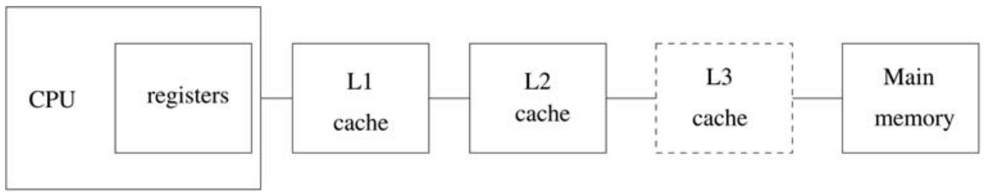

Fig. 1: The memory hierarchy scheme

1 Introduction

In the following text, we assume that A, B and C are real matrices of order n. Every modern CPU architecture use a complex memory hierarchy scheme (for details see [6]), we assume the scheme depicted in Fig. 1. In contrast to other authors, we also assume TLB performance impact.

1.1 The cache model

The cache model we consider corresponds to the structure of L1 and L2 (and optionally L3) caches in the Intel x86 architecture. An *s-way set-associative* cache consists of h *sets* and one set consists of s independent *blocks* (called *lines* in the Intel terminology). Let C_S denote the size of the data part of a cache in bytes and B_S denote the cache block size in bytes. Then $C_S = s \cdot B_S \cdot h$. Let S_D denote the size of type `double` and S_I the size of type `integer`.

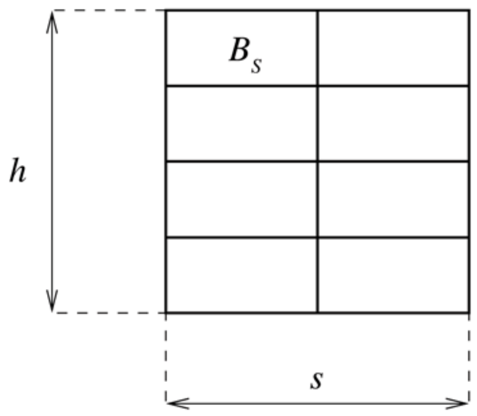

Fig. 2: Parameters of the cache model

We consider only two types of cache misses:

- *Compulsory* misses (sometimes called *intrinsic* or *cold*) that occur when new data is loaded into empty cache blocks.
- *Thrashing* misses (also called *cross-interference*, *conflict*, or *capacity* misses) that occur if useful data has been replaced prematurely from a cache block for capacity reason.

1.2 The code restructuring

There are many compiler techniques for improving the performance (for details see [8, 9]). These compiler techniques are used in programs to maximize the cache hit ratio and improve ALU and FPU utilization.

2 Linear codes

2.1 Existing LA libraries and their possible drawbacks

There exist many libraries for the linear algebra (LA) such as BLAS [5], LAPACK [2], Intel MKL [1] etc. They are tuned for the given architecture, but they also have some drawbacks:

- We must include the whole library in the final code, it can significantly increase the code size.
- Licence problems can occur.
- Some format for storing sparse matrices are not supported.
- Operations are "black boxes", so the user cannot manage the inner solution process.
- A combination of two or more operations with the same operands is solved separately (i.e. inefficiently).

2.2 *Linear code optimization*

There are many routines for linear algebra. For demonstration purposes, we choose following:

1. matrix-matrix multiplication (`GEMM` in BLAS notation).
2. symmetric matrix-matrix multiplication (`SYRK` in BLAS notation).
3. backward substitution (`TRSM` in BLAS notation).
4. Cholesky factorization (`POTRF` in LAPACK notation).

The standard code of these routines has good performance due to the high cache hit ratio only for small sizes of order of matrix. When good performance is required for larger values, modifications must be made. In numerical algebra packages, this is achieved by explicit loop restructuring. This includes loop unrolling-and-jam which increase the FPU pipeline utilization in the innermost loop, loop blocking (which is why we refer to *blocked* codes) and loop interchange to maximize the cache hit ratio. After applying of these transformations, these codes are divided into two parts. The outer loops are "out-cache", and the inner loops are "in-cache". The codes have almost the same performance irrespective on the amount of data.

But good cache utilization can also be achieved by the "divide-and-conquer" approach. These codes, which use recursive-style formulations, referred to as *recursive-based*.

2.3 *Comparison of two approaches for the design of numerical LA routines*

2.3.1 *Blocked code programming*

Effective implementation in blocked code programming style requires the following steps:

1. Straightforward implementation.
2. Design a coarse cache model of this code.
3. Apply source code transformations (according to the architecture parameters).
4. Refine the cache model.
5. Derive optimal values of the program thresholds from the cache parameters.

2.3.2 *Recursive code programming*

The motivation idea of recursive codes, is to divide the matrices into disjoint submatrices of the same size (see [3, 4]). The resulting code has much better spatial and temporal locality. Effective implementation in recursive code programming style requires the following steps:

1. Straightforward implementation.
2. Apply source code transformations (according to architecture parameters).
3. Express routine(s) in the recursive style.
4. Measure threshold value(s).

2.4 *Discussion*

The main differences between blocked and recursive-based codes are

- Cache analysis of some LA routines is very difficult, much more difficult than recursive formulation of these routines.
- For optimal performance of blocked codes, the program may have different in-L1-cache loops and in-L2-cache loops (and in-L3-cache respectively). In recursive-based codes, this is automatically done by the partitioning of the data.
- Blocked codes lead to even better cache utilization because the data is optimally divided to fit into the cache. In recursive codes some part of cache memory remains unused.
- Recursive style of expression is very easy to understand and results in error-free codes.
- Recursive-based codes have additional stack management overhead in comparison to blocked codes.

3 Detailed overview of matrix-matrix multiplication procedure

3.1 *Matrix-matrix multiplication*

We consider input real matrices $A(i \times k)$, $B(k \times j)$ and $C(i \times j)$. A standard sequential pseudocode for matrix-matrix multiplication $C = A \cdot B$ (abbreviated as GEMM, assuming the ordering of loops: i, j, k) is as follows:

```
Pseudocode GEMM_STD
(* Standard matrix-matrix multiplication *)
(1) for $i = 1$ to $n$ do
(2)     for $j = 1$ to $n$ do
(3)         $sum = 0$
(4)         for $k = 1$ to $n$ do
(5)             $sum += A[i][k] * B[k][j]$;
(6)         $C[i][j] = sum$;
```

3.2 *Recursion in matrix-matrix multiplication*

Let us denote $C^{(i)}$ if the final result (matrix C) is computed from the recursion by i variable. Consider the partitioning of matrices A, B and C as follows. For better illustration, the partitioning of matrices is depicted in the Fig. 3.

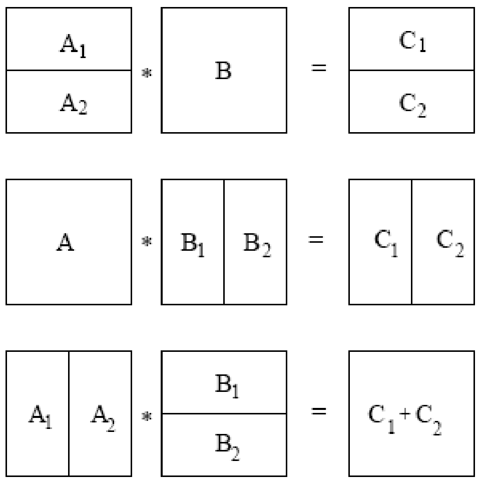

Fig. 3: Recursion in GEMM

$$\mathbf{C}^{(i)} = (C_1 \| C_2) = \mathtt{gemm}(A_1 \| A_2, B) = (A_1 \| A_2) \bullet B = (A_1 \bullet B \| A_2 \bullet B)$$
$$= \left(\mathtt{gemm}(A_1, B) \| \mathtt{gemm}(A_2, B)\right)$$

$$\mathbf{C}^{(j)} = \begin{pmatrix} C_1 \\ C_2 \end{pmatrix} = \mathtt{gemm}\left(A, \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}\right) = A \bullet \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = \begin{pmatrix} A \bullet B_1 \\ A \bullet B_2 \end{pmatrix}$$
$$= \begin{pmatrix} \mathtt{gemm}(A, B_1) \\ \mathtt{gemm}(A, B_2) \end{pmatrix}$$

$$\mathbf{C}^{(k)} = C_1 + C_2 = \mathtt{gemm}\left(\begin{pmatrix} A_1 \\ A_2 \end{pmatrix}, (B_1 \| B_2)\right) = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \bullet (B_1 \| B_2)$$
$$= A_1 \bullet B_1 + A_2 \bullet B_2 = \mathtt{gemm}(A_1, B_1) + \mathtt{gemm}(A_2, B_2)$$

```
Pseudocode GEMM_REC ($A, B, C, i, j, k$)
(* Standard implementation of recursive matrix-matrix multiplication *)
(* limit is an architecture dependent constant *)
(1)   $n = \min(i, j, k)$;
(2)   if ($n < limit$) then
(3)         $C$ = GEMM_STD($A$, $B$);
(4)   else
(5)         switch which of $i, j, k$ is maximum
(6)             case $i$:   is maximum
(7)                 $C$ = MMM_REC($A$, $B$); break;
(7')                (* recursion by $i$ variable *)
(8)             case $j$:   is maximum
(9)                 $C$ = MMM_REC($A$, $B$); break;
(9')                (* recursion by $j$ variable *)
(10)            case $k$:   is maximum
(11)                $C$ = MMM_REC($A$, $B$); break;
(11')               (* recursion by $k$ variable *)
```

3.3 Compulsory misses

During the execution of GEMM(A, B, C, i, j, k) routine all elements from arrays A, B and C must be loaded or written to the cache. So, the number of compulsory misses is

$$N = \frac{S_D(ij + ik + jk)}{B_S}.$$

A similar idea is also valid for TLB misses.

4 Evaluation of the results

4.1 Testing configuration

4.1.1 HW configuration

All results were measured on Pentium Celeron M420 at 1.6 GHz, 2GB@ 333 MHz, with the following cache parameters:

L1 cache is a data cache with $B_S = 64$, $C_S = 32$ kB, $s = 8$, $h = 64$, and LRU replacement strategy. L2 cache is unified cache with $B_S = 64$, $C_S = 1$ MB, $s = 8$, $h = 2048$, and LRU strategy.

4.1.2 SW configuration

We have used the following SW:

- OS Windows XP Professional SP3,
- Microsoft Visual Studio 2003,
- Intel compiler version 9.0 with switches: `/O3 /Og /Oa /Oy /Ot /Qpc64 /QxB /Qipo /Qsfalign16 /Zp16`
- Intel MKL library 8.1.

4.1.3 Test data

We have measured instances using square matrices with the order of n in range from 10 to 1000 (for cache analysis to 800) with the step 10.

4.1.4 Methodology of the measuring

- All cache events were monitored by the Cache Analyzer (CA) [7].
- For TLB miss ratio measuring, we also used the CA with the following parameters: $h = 1$, $s = 128$, $B_S = 4096$. This means, that we assume TLB as a fully-associative cache with its block size equal to the system page size.
- For performance measurements, the average of five measured values was taken as the result.
- The caches were flushed out before each measurement. In real measurement, they were flushed by a reading of a given amount of useless data. In measurement using the CA, they were flushed by a command of the CA.
- The codesize of the innermost loops is negligible in comparison to the size of the L2 cache, so we assume that the unified L2 cache is used only for data.

4.2 Experimental results

4.2.1 L1 cache miss ratio

A comparison of the L1 cache miss rate for three different variants of the GEMM procedure is illustrated in Fig. 4. The peaks in this graph are caused by a resonance between the matrix element mapping function and the cache memory mapping function. Both unrolled variants achieve better L1 utilization.

4.2.2 L2 cache miss ratio

A comparison of the L2 cache miss rate for three different variants of the GEMM procedure is illustrated in Fig 5. We can conclude that if the size of operands exceeds the data cache size (for $n > 350$), the L2 cache utilization drops significantly. Both unrolled variants achieve better L2 utilization.

4.2.3 Choosing the optimal value of the threshold

A comparison of cache utilization when three different threshold values were chosen, is depicted in Figs. 6 and 7. If the threshold is too large, the L1 cache miss ratio increases rapidly. The L2 cache miss ratio remains the same across the whole measured set.

Comparisons of cache miss rates for three different variants of GEMM procedure are illustrated in Figs. 8 and 9. The recursive variant of the GEMM procedure achieves much better cache utilization across the whole measured set. The

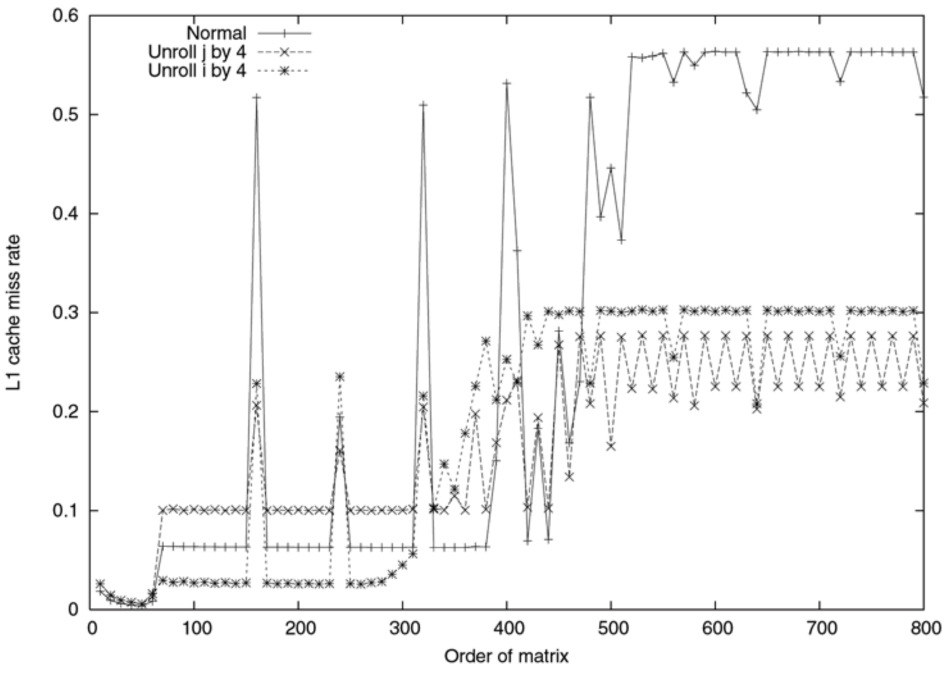

Fig. 4: L1 cache miss rate for different variants of the GEMM procedure

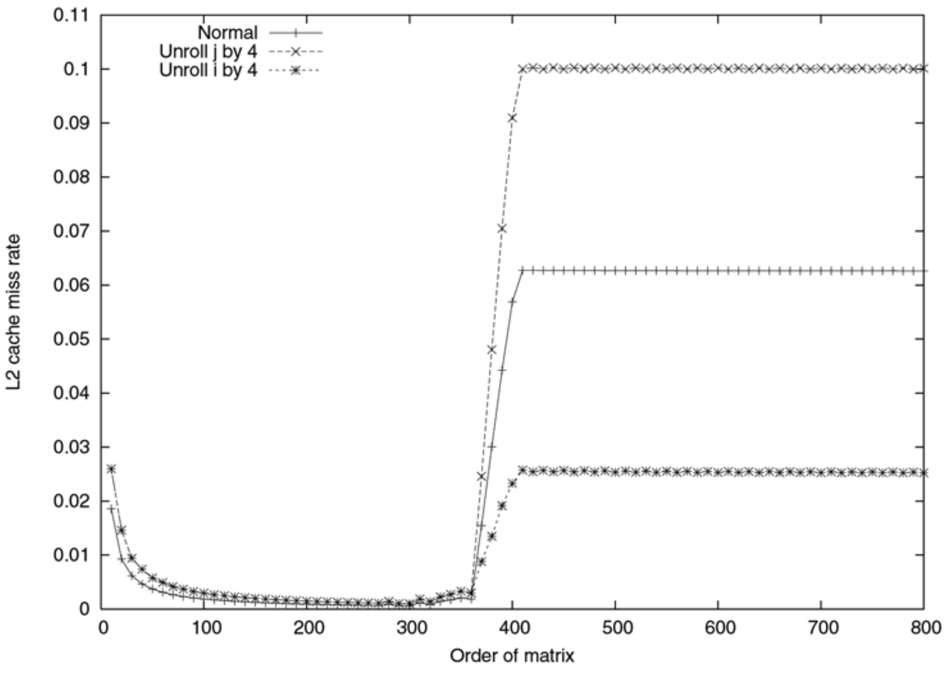

Fig. 5: L2 cache miss rate for different variants of the GEMM procedure

peaks in this graph are caused by resonance of the matrix element mapping function with cache memory mapping function.

4.2.4 TLB cache miss ratio

A comparison of TLB utilization when three different threshold values were chosen, is depicted in Fig. 10. With the exception of one case (caused by resonance of the matrix element mapping function for $n = 510$ with the TLB memory mapping function), the number of TLB misses is very low.

A comparison of the TLB miss rates for three different variants of the `GEMM` procedure is illustrated in Fig. 11. The recursive variant of the `GEMM` procedure achieve much better TLB utilization for larger matrices ($n > 250$).

4.2.5 Performance of GEMM

$$\text{Performance [MFLOPS]} = \frac{number\ of\ FPU\ operations}{execution\ time\ [in\ \mu s]}$$

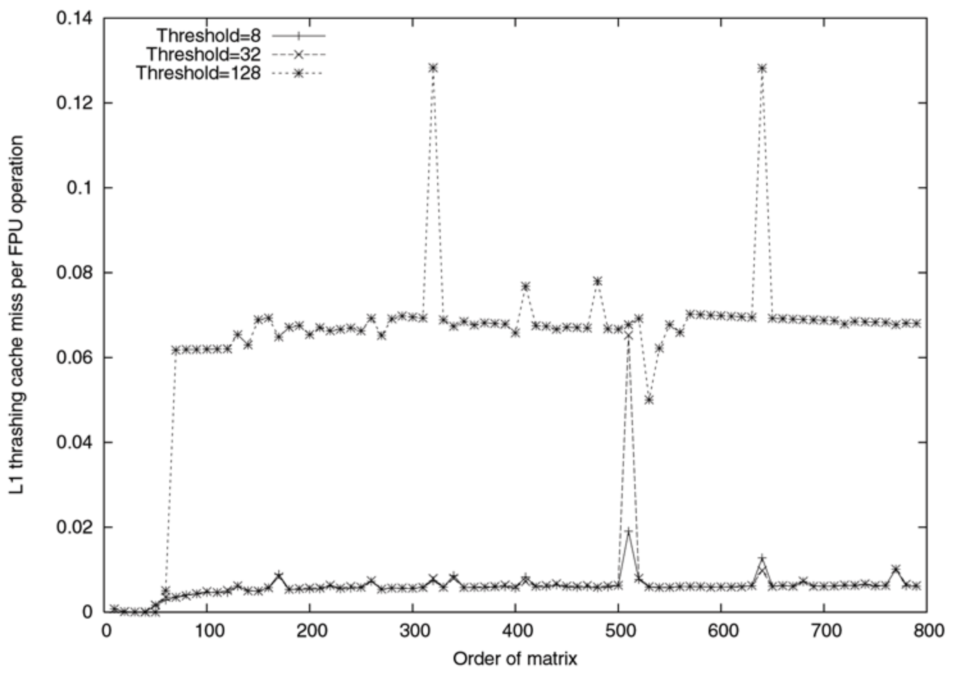

Fig: 6: Dependence of L1 cache utilization on the threshold value

$$\text{Performance [MFLOPS]}_{\mathtt{GEMM}} = \frac{2n^3}{\textit{execution time } [\textit{in } \mu s]}$$

We compare the performance of five different variants:

- The standard variant (GEMM_STD),
- two unrolled variants,
- the recursive variant,
- the vendor implementation (Intel MKL library).

The resulting performances of different versions of GEMM procedure are illustrated in the Fig. 12. We can conclude that

- the standard variant has very low performance.
- two unrolled variants have reasonable performance until the sizes of the operands exceed the cache size. Then

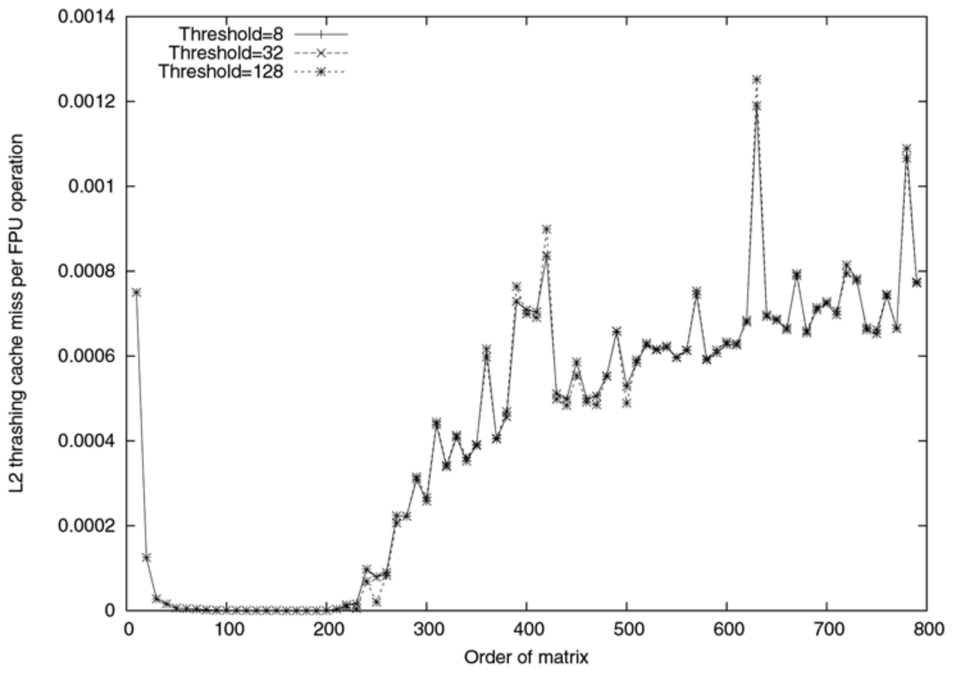

Fig. 7: Dependence of L2 cache utilization on the threshold value

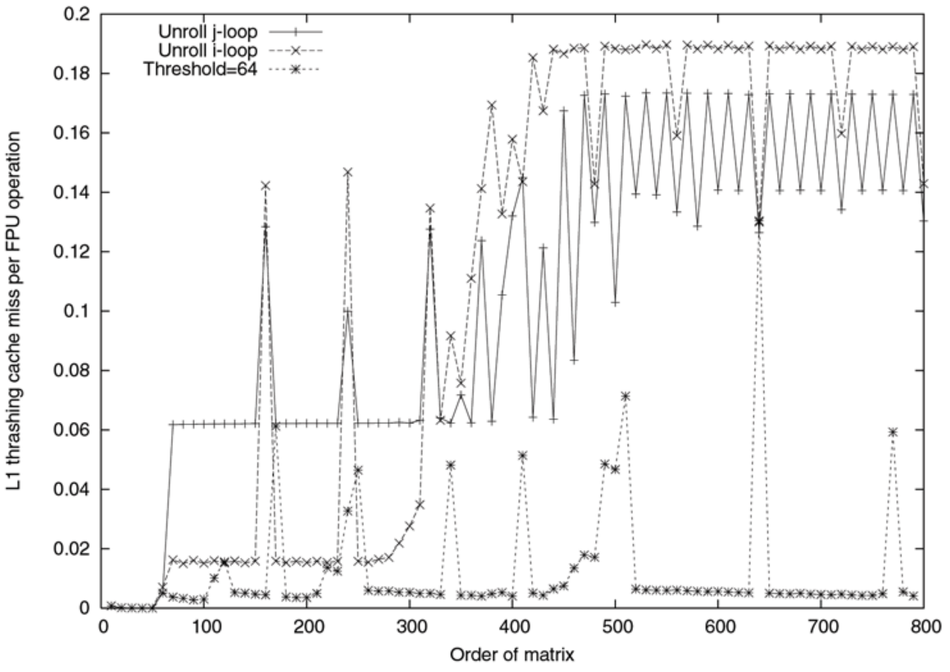

Fig. 8: Comparison between unrolled and recursive variants

the performance suffers from low memory hierarchy utilization.

- the recursive variant achieves very good performance irrespective of matrix size. It was just slightly slower than the vendor implementation.

5 Conclusions

In this paper, we have described the implementation of some important routines from LA using a recursive approach, concentrating on the matrix-matrix multiplication routine. All tested routines achieve performance about 80–90 % in comparison to the vendor MKL library. We can conclude that

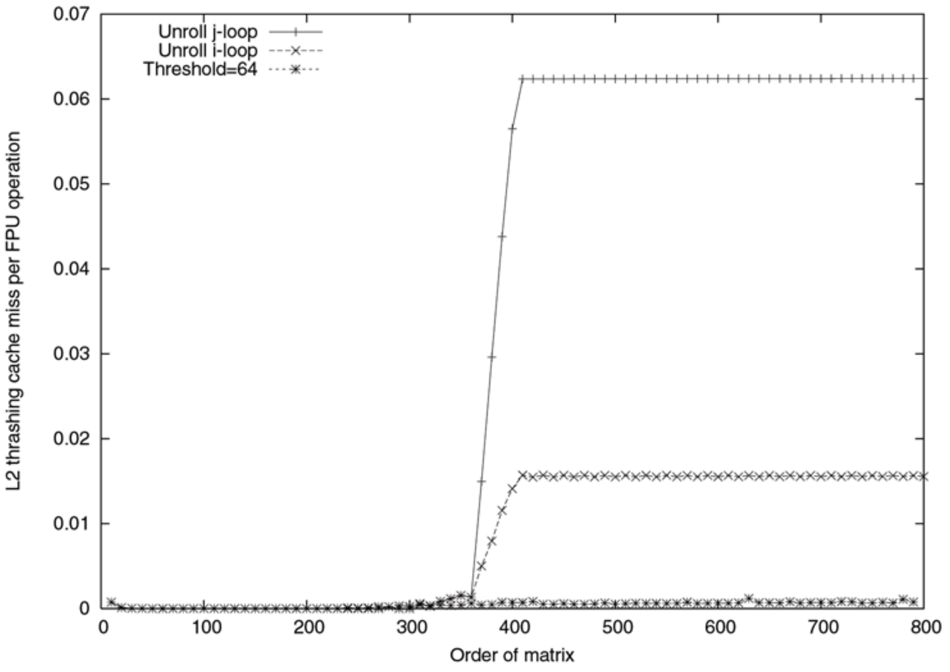

Fig. 9: Comparison between unrolled and recursive variants

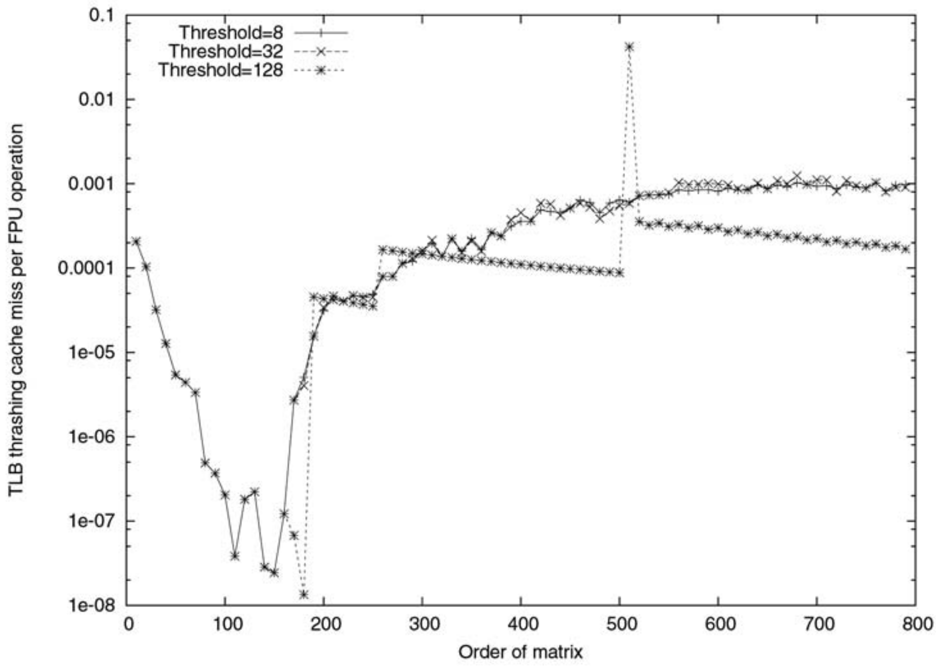

Fig. 10: Dependence of TLB utilization on the threshold value

recursive codes achieve very good performance due to effective memory hierarchy utilization. Unlike other (more complicated) methods, no difficult memory pattern behavior analysis is needed.

Acknowledgement

This research has been supported by MŠMT under research program MSM6840770014.

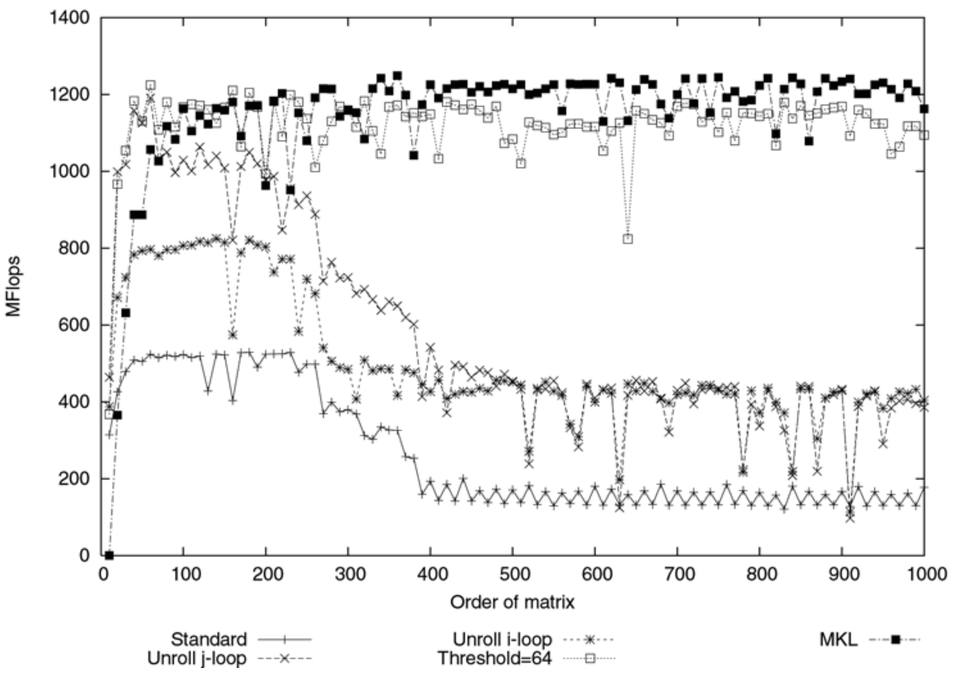

Fig. 11: Comparison between unrolled and recursive variants

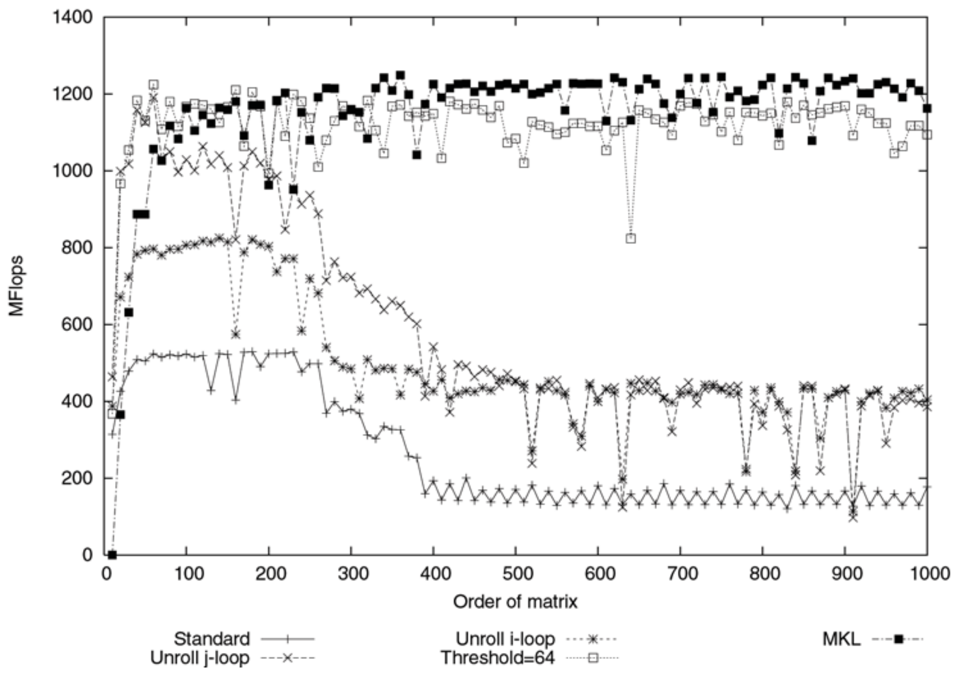

Fig. 12: Performance of different versions of GEMM procedure in double precision

References

[1] Intel® math kernel library 10.1 – overview.

[2] Lapack – linear algebra package.

[3] Carr, S., Kennedy, K.: Compiler Blockability of Numerical Algorithms. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 1992, p. 114–124.

[4] Carr, S., Lehoucq, R. B.: A Compiler-Blockable Algorithm for QR Decomposition. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, February 1995.

[5] Dongarra, J. J., Croz, J. D., Hammarling, S., Duff I.: A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, Vol. **16** (1990), No. 1, p. 1–17, Mar. 1990.

[6] Hennessy, D. A. P. John L.: *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann; 4 edition (September 27, 2006), 2006.

[7] Tvrdík, P., Šimeček, I.: Software Cache Analyzer. In *Proceedings of CTU Workshop*, Vol. **9**, p. 180–181, Prague, Czech Republic, Mar. 2005.

[8] Wadleigh, K. R., Crawford, I. L.: *Software Optimization for High Performance Computing*. Hewlett-Packard professional books, 2000.

[9] Wolfe, M.: *High-Performance Compilers for Parallel Computing*. Addison-Wesley, Reading, Massachusetts, USA, 1995.

Ing. Ivan Šimeček
phone: +420 224 357 268
e-mail:xsimecek@fel.cvut.cz

Department of Computer Science and Engineering

Czech Technical University in Prague
Karlovo nám. 13
121 25 Prague 2, Czech Republic