Performance Aspects of Sparse Matrix-Vector Multiplication

I. Šimeček

Sparse matrix-vector multiplication (shortly SpM×V) is an important building block in algorithms solving sparse systems of linear equations, e.g., FEM. Due to matrix sparsity, the memory access patterns are irregular and utilization of the cache can suffer from low spatial or temporal locality. Approaches to improve the performance of SpM×V are based on matrix reordering and register blocking [1, 2], sometimes combined with software-pipelining [3]. Due to its overhead, register blocking achieves good speedups only for a large number of executions of SpM×V with the same matrix A.

We have investigated the impact of two simple SW transformation techniques (software-pipelining and loop unrolling) on the performance of SpM×V, and have compared it with several implementation modifications aimed at reducing computational and memory complexity and improving the spatial locality. We investigate performance gains of these modifications on four CPU platforms.

Keywords: sparse matrix-vector multiplication, code restructuring, loop unrolling, software pipelining, cache hierarchy.

1 Terminology and notation

Consider a sparse $n \times n$ matrix A with elements A_{ij}; $i, j \in \langle 1, n \rangle$. The largest distance between nonzero elements in any row is the **bandwidth** of matrix A and is denoted by ω_B, i.e.,

$$\mathrm{l}_i = \min_j (j; \mathrm{A}_{ij} \neq 0)$$
$$\mathrm{r}_i = \max_j (j; \mathrm{A}_{ij} \neq 0)$$
$$\omega_B = \max_i (\mathrm{r}_i - \mathrm{l}_i + 1)$$

1.1 Storage schemes for sparse matrices

1.1.1 Compressed sparse row (CSR) format

Matrix A is represented by 3 linear arrays A, *adr*, and *ci* (see Fig. 1). Array A stores the nonzero elements of input matrix A, array *adr*$[1, \ldots, n]$ contains indexes of the initial nonzero elements of rows of A, and array *ci* contains column indexes of nonzero elements of A. Hence, the first nonzero element of row j is stored at index *adr*$[j]$ in array A.

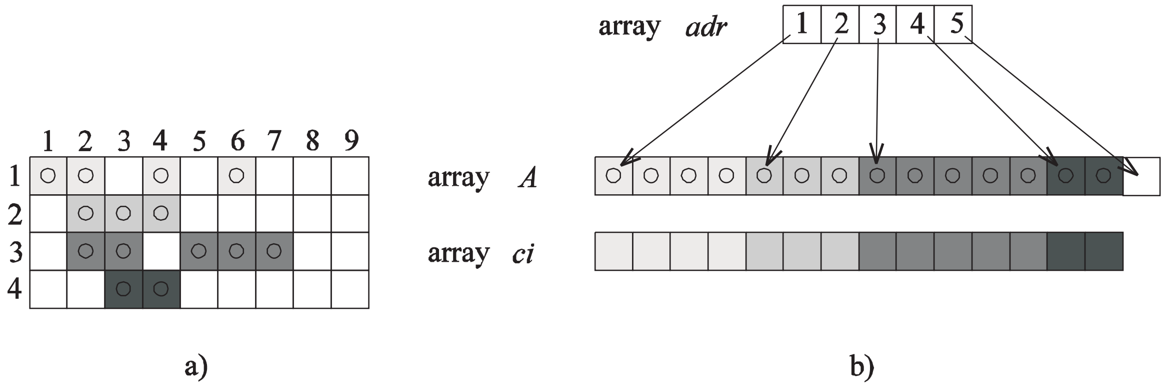

Fig. 1: The idea of the CSR format: a) A sparse matrix A in dense format, b) The CSR representation of A

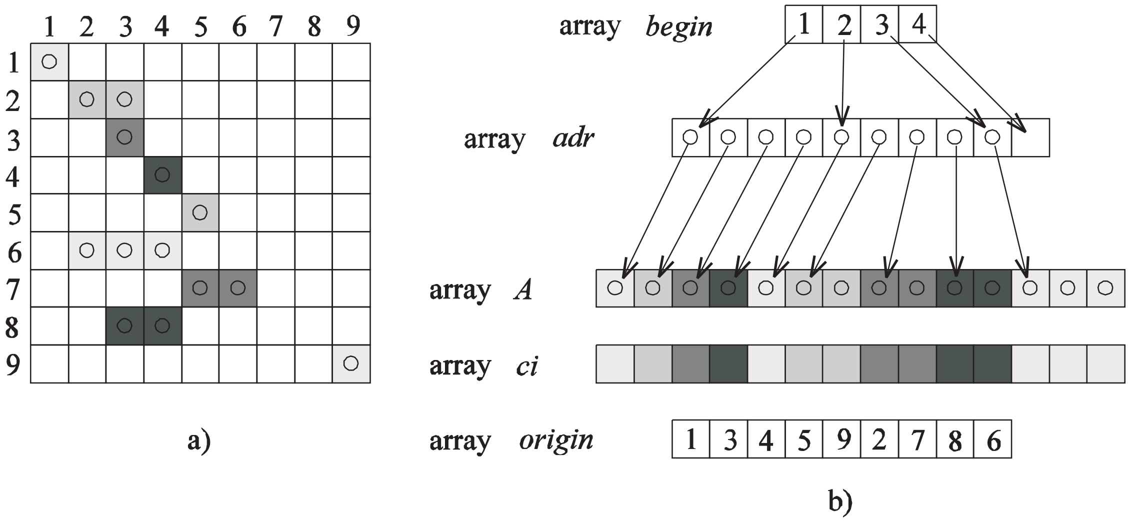

Fig. 2: The idea of the static L-CSR format: a) A sparse matrix A in dense format, b) The static L-CSR representation of A

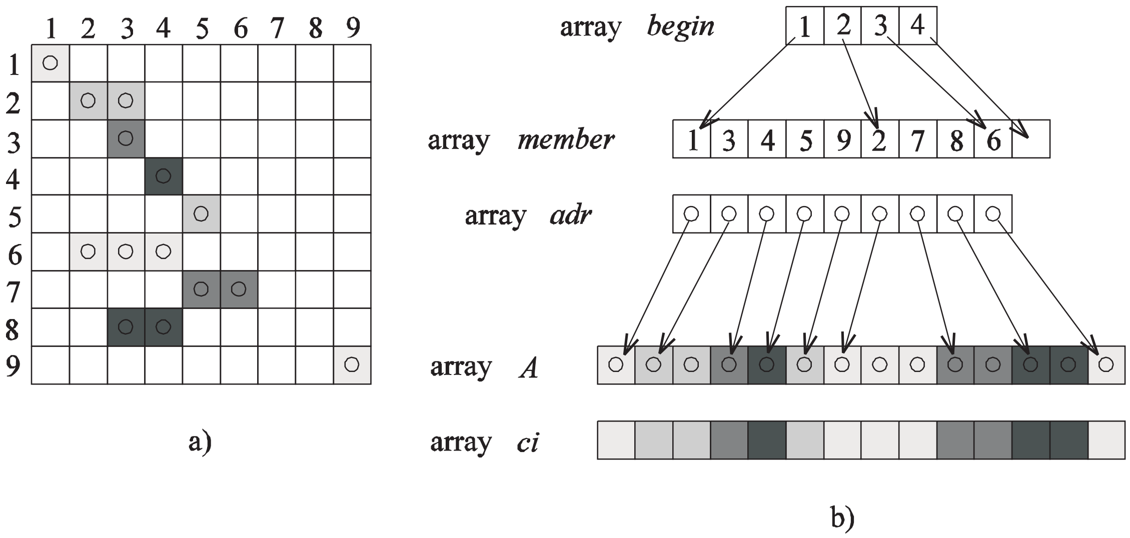

Fig. 3: The idea of the dynamic L-CSR format: a) A sparse matrix *A* in dense format, b) The dynamic L-CSR representation of *A*

1.1.2 Length-sorted CSR (L-CSR) storage format

The main idea is explained in [4]. The data is represented as in the CSR format, but the rows are sorted by length in increasing order. This means that the length of row i is less or equal to the length of row $i+1$. There are two variants:

1. Static: The rows are physically stored in the sorted order in the CSR format (see Fig. 2).
2. Dynamic: The original CSR format is extended with two additional arrays (see Fig. 3). Array *member*$[1 \dots n]$ contains the indexes of the rows after sorting. Array *begin*$[1 \dots \omega_B]$ contains the indexes into the array *member: begin*$[i]$ is the index of the first row of length i in the array *member*.

1.2 Code restructuring

For demonstration purposes, we will use the following pseudocode:

An example code

```
$s = 0.0$;
**for** $i = 1$ **to** $n$ **do**
    $s = s + A[i]$;
```

1.2.1 Loop unrolling

Modern CPUs with deep instruction and arithmetic logic unit (shortly ALU) pipelines achieve peak performance if they execute straight serial codes without conditional branches. A non-optimizing compiler translates a **for** cycle with n iterations into a code with a single loop in which the condition must be tested n times, even if the loop body is very small. Loop unrolling by *unrolling factor* U_f consists in constructing another loop whose body consists of U_f instances of the loop body in a sequence followed by a *clean-up* sequence if n is not a multiple of U_f. This makes the serial code longer, so that the instructions can be better scheduled, the internal pipeline can be better utilized, and the number of condition tests drops from n to $\lceil n / U_f \rceil$.

Loop unrolling applied to the example code ($U_f = 2$)

```
$s = 0.0$;
**for** $i = 1$ **to** $n$ **step** $2$ **do**
    $s = s + A[i]$;
    $s = s + A[i+1]$;
```

1.2.2 Loop unrolling-and-jam (loop jamming)

Since operations in the floating point unit (shortly FPU) on modern CPUs are multi-stage pipelined operations, dependences between iterations can prevent the floating-point pipeline from filling, even if unrolling is used. To improve pipeline utilization in dense matrix codes that have a recurrence in the inner loop, the unroll-and-jam transformation is often used. The transformation consists in unrolling the outer loop and fusing the resulting inner loop bodies. This can increase floating point pipeline utilization by interleaving the computation of multiple independent recurrences.

Loop unrolling-and-jam applied to the example code ($U_f = 2$)

```
$s1 = 0.0$;
$s2 = 0.0$;
**for** $i = 1$ **to** $n$ **step** $2$ **do**
    $s1 = s1 + A[i]$;
    $s2 = s2 + A[i+1]$;
$s = s1 + s2$;
```

1.2.3 Software pipelining

The initial instruction(s) of the first iteration is/are moved into the *prologue* phase, and the final instruction(s) of the last iteration is/are moved into the *epilogue* phase. This technique is usually combined with *loop unrolling* and makes the loop code larger and more efficient for *instruction scheduling*.

Software pipelining applied to the example code

```
$s = 0.0$;
$a1 = A[1]$;
for $i = 2$ to $n$ do
    $a2 = A[i]$;
    $s = s + a1$;
    $a1 = a2$;
$s = s + a2$;
```

1.2.4 Sparse matrix-vector multiplication

Consider a sparse $n \times n$ matrix A stored in the format CSR as defined in the previous section and input dense array $x[1, \ldots, n]$ representing vector x. The goal is to compute output dense array $y[1, \ldots, n]$ representing vector $y = Ax$. The following algorithm `MVM_CSR` is a straightforward implementation of *SpM×V*.

Algorithm `MVM_CSR`

```
$low = adr[1]$;
for $i = 1$ to $n$ do
    $s = 0.0$;
    $up = adr[i + 1]$;
    for $j$ = low to up do
        $k = c[j]$;
        $s = A[j]*x[k]$;
    $y[i] = s$;
    $low = up$;
```

2 Improving the performance of sparse matrix-vector multiplication

The `MVM_CSR` code stated above has poor performance on modern processors. Due to matrix sparsity, the memory access patterns are irregular and the utilization of caches suffers from low spatial and temporal locality. The multiplication at codeline (4) requires indirect addressing, which causes performance degradation due to the small cache-hit ratio. This problem is difficult to solve in general [5] and it is even more difficult if the matrix has only a few nonzero elements per row. In our case, we consider sparse matrices produced by discretization of 2D differential equations of the second order with typical stencils. These matrices contain only a few (typically between 4 and 20) nonzero elements per row.

The performance of *SpM×V* is influenced by SW transformations (shortly SWT) and implementation decisions (shortly ID).

a) **Using explicit software preload (SWT,** *SpM×V_(a)* **)**
The elements of arrays A and ci are loaded in advance by using software-pipelining in the loop at codeline (4). This should hide memory system latencies.

b) **Interleaving of adjacent rows (SWT,** *SpM×V_(b)* **)**
Two (or more) adjacent rows are computed simultaneously by using loop unrolling in the loop at codeline (4). This should improve FPU pipeline utilization.

c) **Condensed implementation of the CSR format (ID,** *SpM×V_(c)* **)**
The matrix in the CSR format is not represented by two independent arrays ci and A, but by a single array that holds two records. This should improve spatial locality, because elements of these arrays are always used together.

d) **Use of pointers in array** ***ci*** **(ID,** *SpM×V_(d)* **)**
The array ci contains pointers to array x instead of indexes into it. This should decrease the amount of work for the operand address decoder in the instruction pipeline.

e) **Use of the static L-CSR format for storing** ***A*** **(ID,** *SpM×V_(e)* **)**
Arrays A and ci are reordered to the static L-CSR format. This modification can save some FPU and ALU operations. The number of conditional branches is strongly reduced, and operations for loading zero into floating point registers are completely removed. For rows with a very small number of nonzero elements, this can have a significant impact.

f) **Using single precision (ID,** *SpM×V_(f)* **)**
All elements of array A are stored in the single precision format (*float*). This floating point format requires half of the space of the *double* format, so the memory requirements for storing A drop by 33 %.

3 HW and SW configuration

The impact of using these transformations of *SpM×V* on the performance was evaluated empirically by measurements on four different processors:

1. IBM Power 3, 200 Mhz, 1 GB RAM, 32 KB instruction and 64 KB data L1 cache (128 B cache line size), and 1 MB L2 cache (32 B cache line size), OS AIX 4.3.2, IBM C compiler, used switches: `-O5`.
2. AMD Opteron, 1.6 GHz, 1 GB RAM, 64 KB instruction and 64 KB data L1 cache, and 1 MB L2 cache, OS Linux Debian, kernel version 2.4.18, GNU C compiler, used switches: `-O3`.
3. Intel Pentium III Coppermine 1 GHz, 512 MB RAM, 16 KB instruction and 16 KB data L1 cache (32 byte cache line size), and 256 KB L2 cache (32 byte cache line size), OS Linux Debian, kernel version 2.4.18, Intel compiler version 6.0 build 020312Z, used switches:
`icc -O3 -fno_alias -pc64 -tpp6 -xK -ipo -align -Zp16.`
4. SUN UltraSparc IIIi (Sparc v9), 1 GHz, 1 GB RAM, 32 KB instruction and 64 KB data L1 cache, and 1 MB L2 cache, OS SunOS 5.9, GNU C compiler, used switches: `-O3`.

The order n starts at $n_0 = 5100$ and grows by a geometric series with factor $q = 1.32$ up to $n_{10} = n_0 \cdot q^{10} = 80400$. All measurements on all four processors were performed with the same set of matrices generated by an FEM generator.

The graphs illustrate the performance of *SpM×V* on these four processors measured in MFlops as a function of the order of matrix A:

- **mv** represents performance of standard *SpM×V*.
- **mv_a** represents performance of *SpM×V_(a)*.
- **mv_b2** represents performance of *SpM×V_(b)* interleaving of 2 rows.
- **mv_b4** represents performance of *SpM×V_(b)* with interleaving of 4 rows.
- **mv_c** represents performance of *SpM×V_(c)* with preload with 1 iteration distance.
- **mv_d** represents performance of *SpM×V_(d)*.
- **mv_e** represents performance of *SpM×V_(e)*.
- **mv_f** represents performance of *SpM×V_(f)*.

4 Results

Evaluation of the results:

- **Using structures (** *SpM×V_(a)* **)**

  Our assumptions were not fulfilled; this modification caused slowdown on all architectures. One possible reason is that the loop code becomes less clear and these compilers are unable to optimize it.
- **Using explicit preload (** *SpM×V_(b)* **)**

  This modification caused slowdown on all architectures. The reason is that explicit SW preload collides with default compiler preload transformation.

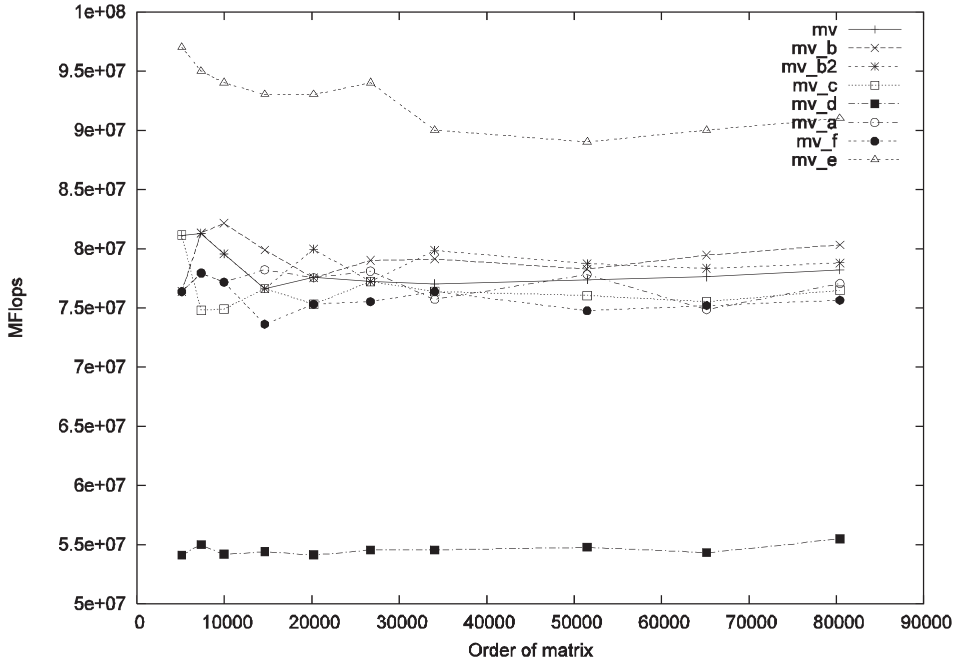

Fig. 4: The performance of algorithms on IBM Power 3

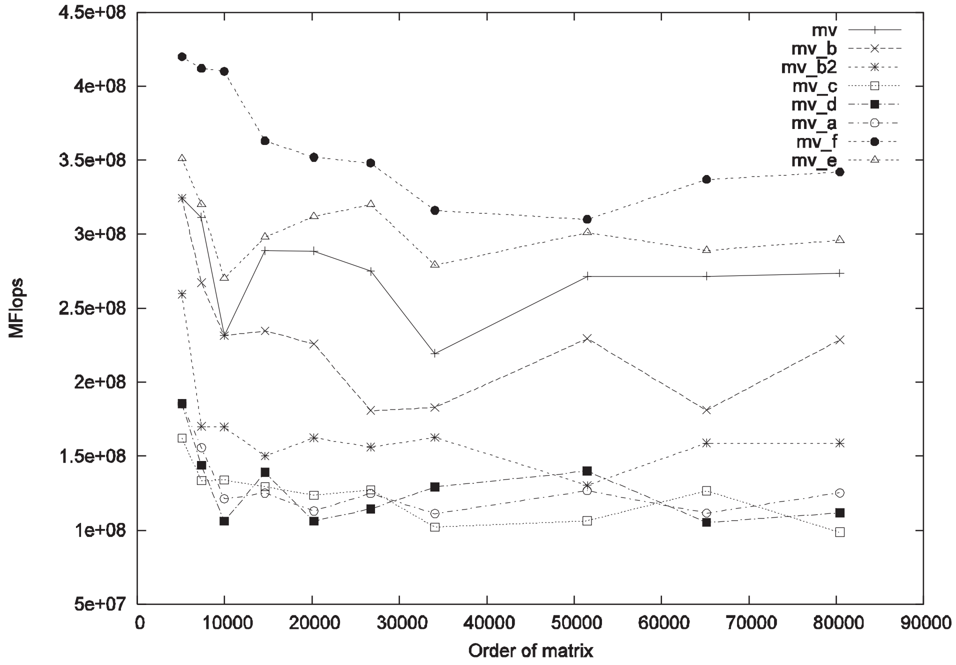

Fig. 5: The performance of algorithms on AMD Opteron

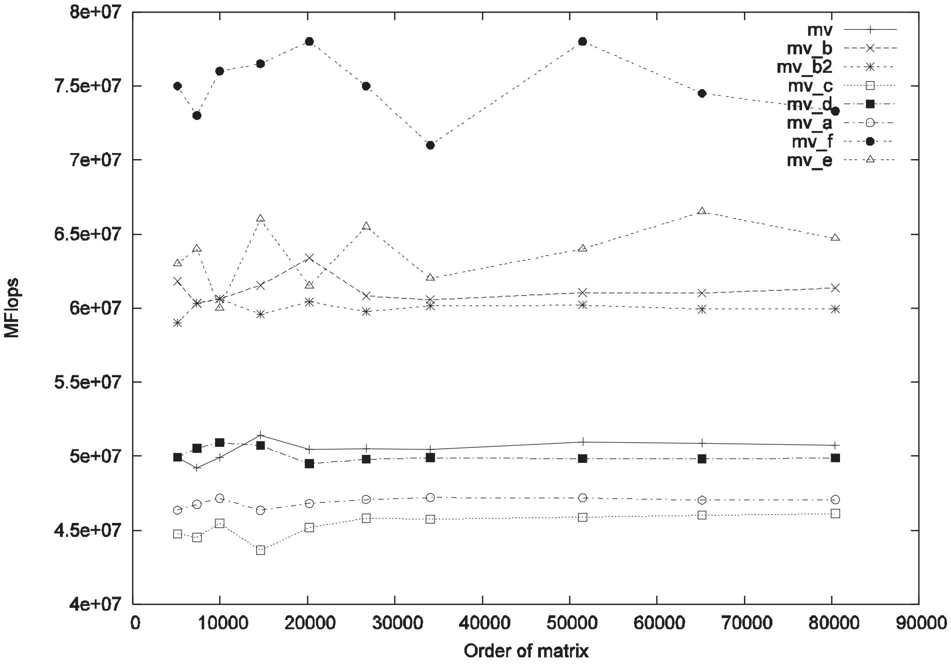

Fig. 6: The performance of algorithms on Intel Pentium 3

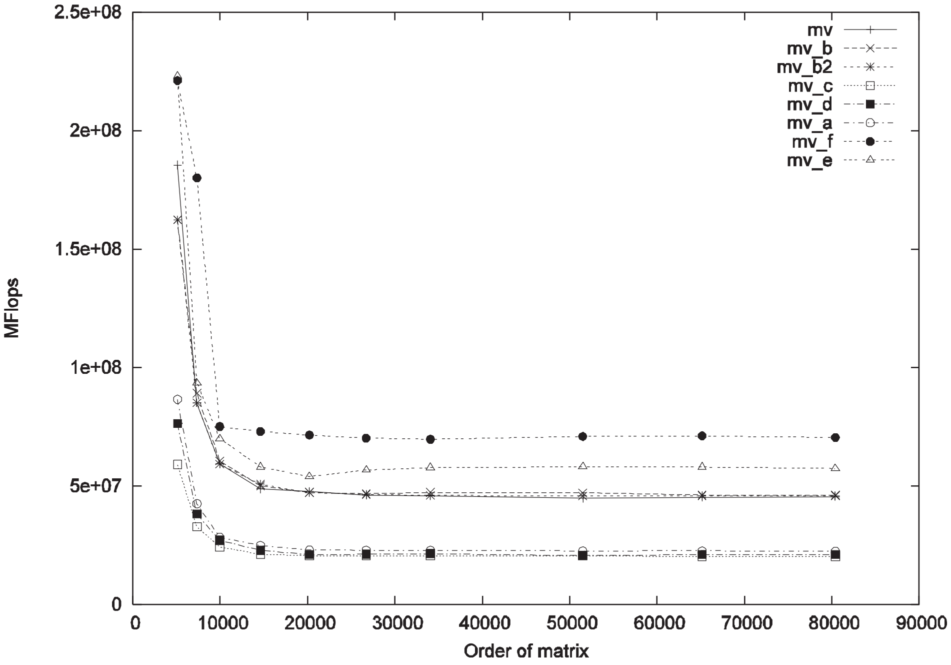

Fig. 7: The performance of algorithms on Sun UltraSparc IIIi

- **Interleaving of 2 adjacent rows (** *SpM*×*V_(c)* **)**
  The effect of this modification is very different (Pentium III: speedup about 20 %, SUN: constant performance, Power 3: constant performance, Opteron: slowdown about 40 %.). One possible reason is that explicit *loop unroll--and-jam* collides with default *loop unroll-and-jam* heuristics in compilers.
- **Using pointers (** *SpM*×*V_(d)* **)**
  Our assumptions were not fulfilled; this modification caused slowdown on all architectures. One possible reason is that the loop code becomes less clear and these compilers are unable to optimize it.
- **Matrix** A **is stored in L-CSR format (** *SpM*×*V_(e)* **)**
  This modification achieves speedup on all architectures due to slight reduction of the number of FPU operations and conditional branches. The main drawback of this method is that "typical" row lengths must be known at compile-time.
- **Using single precision (** *SpM*×*V_(f)* **)**
  This modification achieves speedup on all architectures caused by 33 % smaller amount of data for matrix and by 50 % smaller amount of data for vectors. The main drawback of this method is lower precision of the resulting vector.

5 Conclusion

We have tried to increase the performance of *SpM×V*, one of most common routines in LA.

To fulfill this goal, we have used either SW code transformation techniques or some implementation decisions. We have measured the performance of several modifications of an *SpM×V* algorithm on four different HW platforms. The results differ due to the use of different CPU architectures and compilers, but we can conclude that three of the techniques improve the performance of the code and can be used to accelerate *SpM×V*.

6 Acknowledgment

This work was supported by MŠMT under research program MSM6840770014.

References

[1] Heras, D. B., Cabaleiro, J. C., Rivera, F. F.: Modeling Data Locality for the Sparse Matrix-Vector Product Using Distance Measures. *Parallel Computing*, Vol. **27** (2001), No. 7, p. 897–912, June 2001.

[2] Vuduc, R., Demmel, J. W., Yelick, K. A., Kamil, S., Nishtala, R., Lee, B.: Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In: *Proceedings of Supercomputing 2002*. Baltimore (MD, USA), November 2002.

[3] Rollin, S., Geus, R.: Towards a fast parallel sparse matrix-vector multiplication. In: *Parallel Computing: Fundamentals and Applications*. (D'Hollander, E. H., Joubert, J. R., Peters, F. J., Sips, H. eds.), Proc. of PARCO'99, Imperial College Press, 2000, p. 308–315.

[4] White, J., Sadayappan, P.: On improving the performance of sparse matrix-vector multiplication. In: *Proceedings of the 4th International Conference on High Performance Computing* (HiPC '97), IEEE Computer Society, 1997, p. 578–587.

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

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

Department of Computer Science

Czech Technical University in Prague
Faculty of Electrical Engineering
Technická 2
166 27 Praha 6, Czech Republic