Performance of Portable Sparse Matrix-Vector Product Implemented Using OpenACC

Kinga Stec, Przemysław Stpiczyński

Maria Curie-Skłodowska University, Institute of Computer Science
Akademicka 9, 20-033 Lublin, Poland
Email: kingastec439@gmail.com, przemyslaw.stpiczynski@umcs.pl

Abstract—The aim of this paper is to study the performance of OpenACC implementations of sparse matrix-vector product for several storage formats: CSR, ELL, JAD, pJAD, and BSR, achieved on Intel CPU and NVIDIA GPU platforms to compare them with the performance of SpMV implementations using the BSR storage format provided by Intel MKL and NVIDIA cuSPARSE. Numerical experiments show that vendor-provided BSR is the best format for CPUs but in the case of GPUs, the pJAD storage format allows to achieve better performance.

I. INTRODUCTION

Sparse matrix-vector product (SpMV) is a central part of many numerical algorithms and its performance can have a very big impact on the performance of scientific and engineering applications [1], [2]. There are a lot of various sparse matrix storage formats and sophisticated techniques for developing efficient implementations of SpMV that utilize the underlying hardware of modern multicore CPUs and GPUs [3], [4], [5], [6], [7], [8], [9]. Unfortunately, these methods are rather complicated and usually depend on particular computer architecture, thus developing efficient and portable sparse matrix source code is still a challenge. However, the results presented in [10] and [11] show that simple SPARSKIT SpMV routines using various storage formats (CSR, ELL, JAD) [1] can be easily and efficiently adapted to modern CPU-based or GPU-accelerated architectures. Loops in source codes can be easily parallelized using OpenMP [12] or OpenACC [13], [14] directives, while the rest of the work can be done by a compiler. Such parallelized SpMV routines achieve performance comparable with the performance of the SpMV routines available in libraries optimized by hardware vendors (i.e. Intel MKL, NVIDIA cuSPARSE). OpenACC, a standard for accelerated computing, provides compiler directives for offloading C/C++ programs from host to attached accelerator devices. Such simple directives allow marking regions of source code for automatic acceleration in a portable vendor-independent manner. Moreover, OpenACC programs can be compiled using the multicore option, and then such programs can also be run on CPU-based architectures [15], [16], [17] without any changes in source codes.

Recently, the Block Compressed Row (BSR) format [18], [19], which is a generalization of the Compressed Sparse Row (CSR) format, has become very popular. Intel MKL and NVIDIA cuSPARSE provide optimized SpMV implementations for this format. Moreover, the other formats have been deprecated. Especially, BSR has replaced the HYB format in cuSPARSE. In this paper we compare the performance of portable OpenACC implementations of sparse matrix-vector product for CSR, ELL, JAD, pJAD, and BSR with the performance of SpMV implementations using the BSR storage format provided in Intel MKL and NVIDIA cuSPARSE libraries.

II. SPARSE MATRIX REPRESENTATIONS

Let us assume that $A$ is a sparse matrix with a significant number of zero entries, and $x$, $y$ are dense vectors. The SpMV operation is defined as follows:

$$y \leftarrow Ax.$$  \hspace{1cm} (1)

It is clear that if we do not multiply entries of $x$ by zero entries of $A$, then (1) requires $2 \cdot n_{nz}$ floating point operations (one multiplication and one addition per nonzero entry of $A$). The structure of a sparse matrix can be characterized by $n$, $n_{nz}$, $n_{nz}/n$, and $\max_{nz}$, where $n$ is the number of rows, $n_{nz}$ is the total number of nonzero elements, $n_{nz}/n$ the average number of nonzero elements per row, $\max_{nz}$ is the biggest number of nonzero elements per row.

Table I shows values of these parameters for a set of test matrices, selected from Matrix Market [20] and University of Florida Sparse Matrix Collection [21]. It is clear that the performance of SpMV depends on the matrix storage format that utilizes the underlying hardware.

For description purposes of several possible sparse matrix storage formats, let us consider the following matrix as an example:

$$A = \begin{bmatrix}
7 & 0 & 1 & 0 \\
0 & 4 & 2 & 3 \\
1 & 8 & 0 & 0 \\
0 & 9 & 0 & 0
\end{bmatrix}. \hspace{1cm} (2)$$

where $n = 4$, $n_{nz} = 8$, $n_{nz}/n = 2$, and $\max_{nz} = 3$. Now let us consider a few basic (ELL, JAD, CSR [1], [22]), as well as, more sophisticated (pJAD [11], BSR [18], [19]) storage formats for sparse matrices.

IEEE Catalog Number: CFP2385N-ART ©2023, PTI

Thematic track: Computer Aspects of Numerical Algorithms
A. ELL

The ELL storage format was introduced in Ellpack-Itpack package. It assumes that a sparse matrix is represented by two arrays (Figure 1). Nonzero elements are stored in the first one called a. The second one called ja contains the corresponding column indices [23]. Both arrays are $n \times n_{col}$, where $n_{col} = \max_{n_{nz}}$. While ELL is simple and provides easy access to matrix entries, when $n_{nz} / n < \max_{n_{nz}}$, the number of stored zero entries of the matrix increases significantly.

$$PA = \begin{bmatrix} 0 & 4 & 2 & 3 \\ 7 & 0 & 1 & 0 \\ 1 & 8 & 0 & 0 \\ 0 & 9 & 0 & 0 \end{bmatrix}.$$  

The arrays a and ja of dimension nz contain nonzero elements (i.e. jagged diagonals) and the corresponding column indices. The array ia contains the beginning position of each jagged diagonal. Additionally, we can add array rlen which contains the number of nonzero elements in each row. Entries of this array can be calculated (in parallel) using the following formula. Let $jdiag$ be the number of jagged diagonals. Then for each row, $i = 0, \ldots, n - 1$, we have

$$rlen[i] = |\{j : 0 \leq j \leq jdiag - 1 \land ja[j + 1] - ja[j] > i\}|.$$  

Note that this format is devoid of the inconvenience associated with the need to store zero elements in rows completed to the width of $\max_{n_{nz}}$.

$$a: \begin{bmatrix} 4 & 2 & 3 \\ 7 & 1 & 8 \\ 9 \end{bmatrix}, \quad ja: \begin{bmatrix} 0 & 2 & 3 \\ 1 & 2 & 3 \\ 1 \end{bmatrix}, \quad rlen: \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}.$$  

B. JAD

The JAD (i.e. Jagged Diagonal) format storage is represented by three arrays (Figure 2). It is similar to ELL, but removes the assumption on the fixed-length rows [22]. Firstly, a sparse matrix needs to be sorted in non-increasing order of the number of nonzeros per row.

$$PA = \begin{bmatrix} 0 & 4 & 2 & 3 \\ 7 & 0 & 1 & 0 \\ 1 & 8 & 0 & 0 \\ 0 & 9 & 0 & 0 \end{bmatrix}.$$  

The pJAD storage format is an optimized version of JAD (Figure 3). This format assumes aligning (padding) columns of the arrays a and ja [11]. We add zero elements, thus the number of elements of each column should be a multiple of a given bsize and rows of each block should have the same length. Entries of the array brlen contain widths of blocks of bsize rows. Note that pJAD assumes to store at most $jdiag \cdot (bsize - 1)$ additional zero entries, where $jdiag$ is the number of jagged diagonals stored in a. Padding of jagged diagonals is important especially for GPUs. It allows coalesced memory access and reduces thread divergence within a block of threads.

$$a: \begin{bmatrix} 4 & 2 & 3 \\ 7 & 1 & 8 \\ 9 \end{bmatrix}, \quad ia: \begin{bmatrix} 0 & 2 & 3 \\ 1 & 2 & 3 \\ 1 \end{bmatrix}, \quad brlen: \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}.$$  

### TABLE I: Set of test matrices [11]

<table>
<thead>
<tr>
<th>Matrix</th>
<th>$n$</th>
<th>$n_{nz}$</th>
<th>$n_{nz}/n$</th>
<th>$\max_{n_{nz}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>cry10000</td>
<td>10000</td>
<td>49699</td>
<td>5.0</td>
<td>5</td>
</tr>
<tr>
<td>possession</td>
<td>13514</td>
<td>352762</td>
<td>26.1</td>
<td>110</td>
</tr>
<tr>
<td>af23560</td>
<td>23560</td>
<td>484256</td>
<td>20.6</td>
<td>21</td>
</tr>
<tr>
<td>g7jac10</td>
<td>41490</td>
<td>565956</td>
<td>13.6</td>
<td>153</td>
</tr>
<tr>
<td>fidapm37</td>
<td>9152</td>
<td>765944</td>
<td>83.7</td>
<td>255</td>
</tr>
<tr>
<td>bcsstk36</td>
<td>23052</td>
<td>1143140</td>
<td>49.6</td>
<td>178</td>
</tr>
<tr>
<td>majorbasis</td>
<td>160000</td>
<td>1750416</td>
<td>10.9</td>
<td>11</td>
</tr>
<tr>
<td>bimat</td>
<td>38744</td>
<td>1771722</td>
<td>45.7</td>
<td>126</td>
</tr>
<tr>
<td>cfd1</td>
<td>70656</td>
<td>1828364</td>
<td>25.9</td>
<td>33</td>
</tr>
<tr>
<td>ASIC_680ks</td>
<td>682712</td>
<td>2329176</td>
<td>3.4</td>
<td>210</td>
</tr>
<tr>
<td>FEM_3D_thermal2</td>
<td>147900</td>
<td>3489300</td>
<td>23.6</td>
<td>27</td>
</tr>
<tr>
<td>parabolic_fem</td>
<td>525825</td>
<td>3674625</td>
<td>7.0</td>
<td>7</td>
</tr>
<tr>
<td>ecology2</td>
<td>999999</td>
<td>4995991</td>
<td>5.0</td>
<td>5</td>
</tr>
<tr>
<td>prec2</td>
<td>659033</td>
<td>5959282</td>
<td>9.0</td>
<td>628</td>
</tr>
<tr>
<td>boneS01</td>
<td>127224</td>
<td>6715152</td>
<td>52.8</td>
<td>81</td>
</tr>
<tr>
<td>torso1</td>
<td>116158</td>
<td>8516500</td>
<td>73.3</td>
<td>3263</td>
</tr>
<tr>
<td>thermal2</td>
<td>1228045</td>
<td>8580313</td>
<td>7.0</td>
<td>11</td>
</tr>
<tr>
<td>atmosmodl</td>
<td>1489752</td>
<td>10319760</td>
<td>6.9</td>
<td>7</td>
</tr>
<tr>
<td>bwin3_2</td>
<td>227362</td>
<td>11288630</td>
<td>49.7</td>
<td>336</td>
</tr>
<tr>
<td>af_shell8</td>
<td>504855</td>
<td>17588875</td>
<td>34.8</td>
<td>40</td>
</tr>
<tr>
<td>cage14</td>
<td>1505785</td>
<td>27130349</td>
<td>18.0</td>
<td>41</td>
</tr>
<tr>
<td>md24k</td>
<td>72000</td>
<td>28715634</td>
<td>398.8</td>
<td>520</td>
</tr>
<tr>
<td>intine_1</td>
<td>503712</td>
<td>36816342</td>
<td>73.1</td>
<td>843</td>
</tr>
<tr>
<td>doo1</td>
<td>952203</td>
<td>46522475</td>
<td>48.9</td>
<td>77</td>
</tr>
<tr>
<td>cage15</td>
<td>5154859</td>
<td>99199551</td>
<td>19.2</td>
<td>47</td>
</tr>
</tbody>
</table>

$P = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$
D. CSR

A sparse matrix in CSR (i.e. Compressed Sparse Rows) is stored in three arrays (Figure 4). The first array called `data` contains nonzero elements, and the second one called `cols` contains corresponding column indices of nonzero values. Indices of the beginning of rows in `data` array are stored at the `ptr` array.

```
data:  7  1  4  2  3  1  8  9  
cols: 0  2  1  2  3  0  1  2  
ptr:  0  2  5  7  8
```

Fig. 4: CSR format for (2)

E. BSR

The BSR storage format can be treated as a generalization of CSR. A sparse matrix is represented by four arrays (Figure 5). Array `vals` contains column ordered values from blocks with nonzero values. Array `cols` stores columns indices of the first element per block. The `ptrB` and `ptrE` arrays contain the indices of the beginning and ending positions of the elements in the block row respectively.

```
vals:  7  0  0  4  1  2  0  3  1  0  8  9
cols:  0  1  0  
ptrB:  0  2  
ptrE:  2  3
```

Fig. 5: BSR format for (2)

III. ALGORITHMS

The SpMV operation for all storage formats presented in Section II can be implemented using OpenACC to be executed on both GPU-accelerated and CPU-based systems. OpenACC offers compiler directives for offloading selected computations from host to attached accelerator devices. It allows to indicate regions of source code for automatic parallelization in a portable manner. Algorithms 1, 2, 3, 4, and 5 show how to implement SpMV in C/C++ using OpenACC for all considered formats: ELL, JAD, pJAD, CSR, and BSR formats, respectively. OpenACC-specific parts of the implementation start with #pragma acc directives. The parallel loop directive defines a loop to be accelerated on GPU. Additional clauses, namely `gang` and `vector_length` tell that gangs (i.e. blocks of threads) should perform an iteration of loops. Threads within gangs work in vector or SIMD mode [13]. The loop seq construct placed before a loop within parallel loop says that such a loop should be executed sequentially by a single thread. The present clause says that indicated variables are previously allocated on GPU. It allows to avoid unnecessary data movements between host and device memory systems. OpenACC provides the data construct that can be used to specify such scope of data in accelerated regions. Data transfers can also be initialized using the enter data and exit data constructs [13]. Figure 6 shows output messages generated by the compiler using the -acc=gpu option.

When OpenACC programs are compiled using the -acc=multicore option, the compiler generates appropriate parallel regions to be executed in parallel on CPU cores (Figure 7). It should be noticed that if we omit OpenACC directives, we will get sequential implementations of SpMV.

IV. PERFORMANCE OF SpMV

All OpenACC implementations of SpMV have been tested on the computer equipped with two Xeon Gold 6342 @ 2.80GHz (48 cores) and NVIDIA A40 GPU (10752 cores, FP64 Peak perf. 584.6 GFLOPS), running under Linux Oper-
Algorithm 4 SpMV using CSR in OpenACC

```c
// auxiliary routine
double count_per_row(int nz_in_row, int idx_start, int *cols, double *data, double *x) {
    double t = 0;
    #pragma acc loop seq
    for(int j = 0; j < nz_in_row; j++) {
        t += x[cols[idx_start+j]] * data[idx_start+j];
    }
    return t;
}

// driver routine
void CSR_SpMV(int n, double *data, int *cols, int *ptr, double *x, double *y) {
    #pragma acc parallel loop gang vector_length(128)
    present(x, y, cols, ptr, data, y)
    for(int i = 0; i < n; i++) {
        y[i] = count_per_row(ptr[i]+1-ptr[i], ptr[i], cols, data, x);
    }
}
```

Algorithm 5 SpMV using BSR in OpenACC

```c
// auxiliary routine
void count_per_block(int block_size, int rows_begin, int rows_end, int *cols, double *vals, double *x, double *y, int i) {
    #pragma acc loop seq
    for(int j = rows_begin; j < rows_end; j++) {
        int base = j * block_size + block_size;
        for(int jdx = cols[j] * block_size; jdx < (cols[j]+1) * block_size; jdx++) {
            for(int idx = j * block_size; idx < (i + 1) * block_size; idx++) {
                y[idx] += vals[base] * x[jdx];
                base += 1;
            }
        }
    }
}

// driver routine
void BSR_SpMV(int rows, int block_size, int *ptrB, int *ptrE, int *cols, double *vals, double *x, double *y) {
    #pragma acc parallel loop gang vector_length(128)
    present(x, y, vals, ptrB, ptrE, cols)
    for(int i = 0; i < rows; i++) {
        count_per_block(block_size, ptrB[i], ptrE[i], cols, vals, x, y, i);
    }
```

Fig. 6: Compiler output messages for Algorithm 1 compiled using -acc=multicore

Fig. 7: Compiler output messages of Algorithm 1 compiled using -acc=multicore -acc=gpu

ELL_SpMV:
- 13, Generating Multicore code
- 15, #pragma acc loop gang

count_per_row:
4, Generating implicit acc routine seq
Generating acc routine seq
Generating NVIDIA GPU code

ELL_SpMV:
13, Generating present(y[:],x[:],ja[:],a[:])
Generating NVIDIA GPU code
15, #pragma acc loop gang, vector(128)
/* blockIdx.x threadIdx.x */

V. RESULTS OF EXPERIMENTS

On CPU, the best performance for the majority of matrices is obtained for Intel MKL BSR implementation. For the smaller matrices, the best results are achieved by OpenACC implementation of SpMV using the CSR format. Other OpenACC implementations achieve worse performance than Intel MKL BSR. Especially, OpenACC BSR is much slower than its well-optimized counterpart. In most cases pJAD achieves better performance than JAD, however its performance is
still worse than the optimized BSR provided by Intel. The ELL format gives the worse performance for matrices with $n_{nz}/n \leq \max_{n_{nz}}$, when the number of stored zero entries increases significantly.

On GPU, we can observe that pJAD implementation achieves the best results for the largest number of matrices (eleven matrices). It outperforms JAD format for all matrices and Nvidia cuSPARSE BSR for almost all matrices. Moreover, for several matrices pJAD outperforms Intel MKL BSR significantly. The second best format is CSR. It gains the best results for seven matrices. For the others, the BSR format is always better and the JAD format is almost always better. The ELL format gives the worse performance for matrices with $n_{nz}/n \leq \max_{n_{nz}}$. Nvidia cuSPARSE BSR gains the highest performance for matrices with $n_{nz}/n > \max_{n_{nz}}$.

On GPU, we can observe that pJAD implementation achieves the best results for the largest number of matrices (eleven matrices). It outperforms JAD format for all matrices and Nvidia cuSPARSE BSR for almost all matrices. Moreover, for several matrices pJAD outperforms Intel MKL BSR significantly. The second best format is CSR. It gains the best results for seven matrices. For the others, the BSR format is always better and the JAD format is almost always better. The ELL format gives the worse performance for matrices with $n_{nz}/n \leq \max_{n_{nz}}$. Nvidia cuSPARSE BSR gains the highest performance for only one matrix (i.e., ldooor), however for larger matrices it is much faster than OpenACC BSR. As with CPU, the difference between OpenACC and Nvidia cuSPARSE implementations of BSR is significant.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>OpenACC CPU</th>
<th>MKL</th>
<th>OpenACC GPU</th>
<th>cuSPARSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CSR</td>
<td>ELL</td>
<td>JAD</td>
<td>pJAD</td>
</tr>
<tr>
<td>cry10000</td>
<td>2.56</td>
<td>1.07</td>
<td>1.16</td>
<td>1.02</td>
</tr>
<tr>
<td>poisson3Da</td>
<td>7.57</td>
<td>1.43</td>
<td>6.85</td>
<td>6.37</td>
</tr>
<tr>
<td>af23560</td>
<td>7.78</td>
<td>7.59</td>
<td>5.13</td>
<td>4.76</td>
</tr>
<tr>
<td>g7jac140</td>
<td>8.07</td>
<td>9.77</td>
<td>2.65</td>
<td>3.11</td>
</tr>
<tr>
<td>lidopm37</td>
<td>14.36</td>
<td>3.31</td>
<td>7.27</td>
<td>6.91</td>
</tr>
<tr>
<td>bcsstk36</td>
<td>12.18</td>
<td>2.40</td>
<td>6.46</td>
<td>7.47</td>
</tr>
<tr>
<td>majorbasis</td>
<td>12.51</td>
<td>11.05</td>
<td>8.38</td>
<td>9.35</td>
</tr>
<tr>
<td>bbbmat</td>
<td>13.41</td>
<td>4.14</td>
<td>5.59</td>
<td>6.09</td>
</tr>
<tr>
<td>cage1</td>
<td>14.22</td>
<td>8.38</td>
<td>8.50</td>
<td>8.89</td>
</tr>
<tr>
<td>ASIC_680ks</td>
<td>4.25</td>
<td>0.21</td>
<td>2.07</td>
<td>2.20</td>
</tr>
<tr>
<td>FEM_3D_thermal2</td>
<td>14.27</td>
<td>9.76</td>
<td>6.81</td>
<td>9.17</td>
</tr>
<tr>
<td>parabolic_fem</td>
<td>6.15</td>
<td>5.73</td>
<td>5.45</td>
<td>5.68</td>
</tr>
<tr>
<td>ecology2</td>
<td>5.86</td>
<td>6.48</td>
<td>5.43</td>
<td>5.67</td>
</tr>
<tr>
<td>pre2</td>
<td>6.99</td>
<td>0.13</td>
<td>2.34</td>
<td>3.14</td>
</tr>
<tr>
<td>boneS01</td>
<td>14.03</td>
<td>7.98</td>
<td>5.84</td>
<td>8.91</td>
</tr>
<tr>
<td>torso1</td>
<td>12.07</td>
<td>0.27</td>
<td>0.71</td>
<td>0.52</td>
</tr>
<tr>
<td>thermal2</td>
<td>6.19</td>
<td>5.36</td>
<td>4.54</td>
<td>5.43</td>
</tr>
<tr>
<td>atmosmodl</td>
<td>7.73</td>
<td>8.11</td>
<td>6.88</td>
<td>6.93</td>
</tr>
<tr>
<td>bmxw3_2</td>
<td>10.83</td>
<td>1.53</td>
<td>5.14</td>
<td>9.73</td>
</tr>
<tr>
<td>af_shell8</td>
<td>4.24</td>
<td>4.98</td>
<td>4.86</td>
<td>6.21</td>
</tr>
<tr>
<td>cage14</td>
<td>6.29</td>
<td>2.83</td>
<td>4.12</td>
<td>3.68</td>
</tr>
<tr>
<td>nd24k</td>
<td>10.64</td>
<td>7.00</td>
<td>6.31</td>
<td>5.38</td>
</tr>
<tr>
<td>inline_1</td>
<td>9.95</td>
<td>0.72</td>
<td>6.08</td>
<td>5.72</td>
</tr>
<tr>
<td>cage15</td>
<td>10.50</td>
<td>3.68</td>
<td>5.60</td>
<td>6.40</td>
</tr>
</tbody>
</table>


