Regu2D: Accelerating Vectorization of SpMV on Intel Processors through 2D-partitioning and Regular Arrangement

Xiang Fei & Youhui Zhang*
Tsinghua University, Beijing, China
Outline

• Introduction
• Previous Work Analyses
• Solutions and Implementation
  • 1 Regular Arrangement
  • 2 Adaptive 2D-partitioning
  • 3 Indices Compression
  • 4 Load Balancing
• Experiment
  • Preprocessing
• Conclusions
Introduction

• Sparse matrix-vector multiplication (SpMV) is an elementary and necessary kernel in many HPC (high-performance computing) domain applications.

• Accelerating SpMV faces three main issues:
  • Irregular data access patterns (because of indexed-load like a[b[i]])
  • Memory bandwidth (one multiplication of SpMV requires three load instructions, two of which (array data&col in Alg. 1) access the matrix data sequentially without data reuse)
  • Short vector problem on vector processors (or SIMD processors in other words). Due to the sparsity and irregularity of non-zero elements in some rows, the lanes of a vector register may not be fully filled
Introduction

• Sparse matrix-vector multiplication (SpMV) is an elementary and necessary kernel in many HPC (high-performance computing) domain applications.

• Accelerating SpMV faces three main issues:
  • Irregular data access patterns (because of indexed-load like a[b[i]])
  • Memory bandwidth (one multiplication of SpMV requires three load instructions, two of which (array data & col in Alg. 1) access the matrix data sequentially without data reuse)
  • Short vector problem on vector processors (or SIMD processors in other words). Due to the sparsity and irregularity of non-zero elements in some rows, the lanes of a vector register may not be fully filled.
Introduction

• Sparse matrix-vector multiplication (SpMV) is an elementary and necessary kernel in many HPC (high-performance computing) domain applications.

• Accelerating SpMV faces three main issues:
  • Irregular data access patterns (because of indexed-load like a[b[i]])
  • Memory bandwidth (one multiplication of SpMV requires three load instructions, two of which (array data&col in Alg. 1) access the matrix data sequentially without data reuse)
  • Short vector problem on vector processors (or SIMD processors in other words). Due to the sparsity and irregularity of non-zero elements in some rows, the lanes of a vector register may not be fully filled
Introduction

- Sparse matrix-vector multiplication (SpMV) is an elementary and necessary kernel in many HPC (high-performance computing) domain applications.

- Accelerating SpMV faces three main issues:
  - Irregular data access patterns (because of indexed-load like a[b[i]])
  - Memory bandwidth (one multiplication of SpMV requires three load instructions, two of which (array data&col in Alg. 1) access the matrix data sequentially without data reuse)
  - Short vector problem on vector processors (or SIMD processors in other words). Due to the sparsity and irregularity of non-zero elements in some rows, the lanes of a vector register may not be fully filled.
Outline

• Introduction

• Previous Work Analyses

• Solutions and Implementation
  • 1 Regular Arrangement
  • 2 Adaptive 2D-partitioning
  • 3 Indices Compression
  • 4 Load Balancing

• Experiment
  • Preprocessing

• Conclusions
Previous Work Analyses

- There are many existing sparse matrix formats on vector processors
  - ELL: may introduce too many padding zeros
  - JDS: (a variant of ELL) too many storage/updating operations
  - SELL-C-\(\sigma\): (a variant of ELL) still contain many padding zeros, the best value of \(\sigma\) is unknown

- Some new formats are proposed
Table 1: Optimization methods of the previous works

<table>
<thead>
<tr>
<th></th>
<th>ALBUS</th>
<th>VHCC</th>
<th>CSR5</th>
<th>CVR</th>
<th>Regu2D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D-partition</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Splice rows</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Use FMA</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Avoid branches</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Use scatter instr.</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- **ALBUS**: It is a simple work, and more likely to meet the short vector problem.
- **VHCC**: Uses splicing to avoid the short vector problem, but introduces many reduction additions. It needs conditional branches to store the results to the output vector.
- **CSR5**: Arranges elements vertically. It puts elements vertically to partly avoid reduction additions, and cuts one row into several parts and needs additional operation to reduce. It also needs conditional branches to store the results.
- **CVR**: Avoids reduction additions completely, but also needs conditional branches to store the results.
- **Regu2D**: Avoids conditional branches, and uses gather/scatter to update the results.

**Figure 3**: The evolution of the previous works and Regu2D.

**Figure 2**: A sparse matrix example and the corresponding data layouts for the previous works and Regu2D.
Figure 2: A sparse matrix example and the corresponding data layouts for the previous works and Regu2D.

Figure 3: The evolution of the previous works and Regu2D.

Algorithm 5: ALBUS kernel

for each row do
  sum = 0 (vector variable);
  for each vector do
    Vector load values and do vector multiplication;
    sum += result;
  end
end
Reduce sum and store with a scalar instruction;
end

Table 1: Optimization methods of the previous works

<table>
<thead>
<tr>
<th>Method</th>
<th>ALBUS</th>
<th>VHCC</th>
<th>CSR5</th>
<th>CVR</th>
<th>Regu2D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D-partition</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>v</td>
</tr>
<tr>
<td>Splice rows</td>
<td>x</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Use FMA</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Avoid branches</td>
<td>v</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Use scatter instr.</td>
<td>x</td>
<td>x</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
</tbody>
</table>

FGCS 2021
CGO 2015
ICS 2015
CGO 2018
ICPP 2021

More likely to meet the short vector problem
Conditional branches to test whether some vector lanes meet the last element of one row after each multiplication.

Algorithm 4: VHCC kernel

For each vector do
    Vector load values and do vector multiplication;
    if meet the last element of certain row(s) then
        Calculate segment sum with masks;
        while meet the last element of certain row(s) do
            Access value of corresponding vector lane;
            Update result with scalar instructions;
        end
    else
        Reduce the whole vector and store at lane 0;
    end
end
Table 1: Optimization methods of the previous works

<table>
<thead>
<tr>
<th></th>
<th>ALBUS</th>
<th>VHCC</th>
<th>CSR5</th>
<th>CVR</th>
<th>Regu2D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D-partition</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Splice rows</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Use FMA</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Avoid branches</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Use scatter instr.</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

FGCS 2021

CGO 2015

ICS 2015

CGO 2018

ICPP 2021

Too complicated to show the pseudo code of CSR5
### Table 1: Optimization methods of the previous works

<table>
<thead>
<tr>
<th>Method</th>
<th>ALBUS</th>
<th>VHCC</th>
<th>CSR5</th>
<th>CVR</th>
<th>Regu2D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D-partition</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Splice rows</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Use FMA</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Avoid branches</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Use scatter instr.</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

Use **FMA** (fused multiply-add) to avoid reduction additions.

**FGCS 2021**

**CGO 2015**

**ICS 2015**

**CGO 2018**

**ICPP 2021**

---

**Algorithm 3: CVR kernel**

```plaintext
for each vector do
    Vector load values and do vector multiplication;
    while meet the last element of certain row(s) do
        Access value of corresponding vector lane;
        Update result with scalar instructions;
        Clear corresponding vector lane to zero;
    end
end
```

Use FMA (fused multiply-add) to avoid reduction additions.

---

**Figure 2: A sparse matrix example and the corresponding data layouts for the previous works and Regu2D**

- **Original matrix**
- **SIMD lanes**
  - **SIMD lanes** 0 1 2 3
  - **SIMD lanes** 0 1 2 3
- **ALBUS**
  - **ALBUS** 0 1 2 3
  - **ALBUS** 0 1 2 3
- **VHCC**
  - **VHCC** 0 1 2 3
  - **VHCC** 0 1 2 3
- **CSR5**
  - **CSR5** 0 1 2 3
  - **CSR5** 0 1 2 3
- **CVR**
  - **CVR** 0 1 2 3
  - **CVR** 0 1 2 3
- **Regu2D**
  - **Regu2D** 0 1 2 3
  - **Regu2D** 0 1 2 3

---

**International Conference on Parallel Processing (ICPP)**
August 9-12, 2021 in Virtual Chicago, IL
Table 1: Optimization methods of the previous works

<table>
<thead>
<tr>
<th>Method</th>
<th>ALBUS</th>
<th>VHCC</th>
<th>CSR5</th>
<th>CVR</th>
<th>Regu2D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D-partition</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Splice rows</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Use FMA</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Avoid branches</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Use scatter instr.</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Use gather/scatter to update the results

FGCS 2021
CGO 2015
ICS 2015
CGO 2018
ICPP 2021

Figure 2: A sparse matrix example and the corresponding data layouts for the previous works and Regu2D

Algorithm 7: Computations of Regu2D

```plaintext
for each group do
    len = group length;
    sum = 0;
    for i = 0 to len do
        Load matrix value a and column indices;
        Indexed load input vector b:
        sum += a * b;
    end
    Load row indices (if no compression);
    Scatter sum to the output vector;
end
for other rows do
    Do it as ALBUS does;
end
```
Previous Work Analyses

• The computation of SpMV can be divided into three stages:
  • The multiplication stage
  • The addition or reduction stage of each row
  • The storing/updating stage

• Different element layouts decide the specific reduction and update methods. An ideal approach should concern
  • (1) To decrease the number of addition operations during reduction, and avoid using conditional branch instructions whenever possible;
  • (2) To avoid accessing lane(s) of a vector separately;
  • (3) To use vector gather/scatter instr. in the updating stage whenever possible.
Previous Work Analyses

- The computation of SpMV can be divided into three stages:
  - The multiplication stage
  - The addition or reduction stage of each row
  - The storing/updating stage

- Different element layouts decide the specific reduction and update methods. An ideal approach should concern
  - (1) To decrease the number of addition operations during reduction, and avoid using conditional branch instructions whenever possible;
  - (2) To avoid accessing lane(s) of a vector separately;
  - (3) To use vector gather/scatter instr. in the updating stage whenever possible.
Outline

• Introduction
• Previous Work Analyses
• Solutions and Implementation
  • 1 Regular Arrangement
  • 2 Adaptive 2D-partitioning
  • 3 Indices Compression
  • 4 Load Balancing
• Experiment
  • Preprocessing
• Conclusions
Solutions and Implementation

- Regu2D = Regular Arrangement + Adaptive 2D-partitioning
- Use 2D-partitioning to alleviate Irregular access
  - but it cuts each row into several parts, and aggravates the short vector problem and need to store the row indices several times (large storage space)
- Use splicing to alleviate the short vector problem
  - Use regular arrangement to solve the Inefficiency of reduction and updating
- Use indices compression to decrease the storage space and the amount of memory IO

Figure 4: Four challenges and three solutions we adopt. The solid line and the dashed line respectively indicate that the solution has alleviated or aggravated the corresponding problem.
1 Regular Arrangement

• Select \( N \) rows that have the same number of elements to be a group and then make the \( i \) th element of row \( j \) \((0 \leq j < N)\) occupy lane \( j \) of vector \( i \) (arrange non-zero elements \textit{vertically}). Thus each row needs the same number of FMA (fused multiply-add) in the reduction stage.
1 Regular Arrangement

- The regular arrangement is required to find as many groups as possible to cover the maximum number of rows and elements. We use a dynamic programming algorithm to find them.
  - To increase the flexibility of regular arrangement, we set a threshold $T1$ and limit the maximum difference of the number of elements in $N$ rows to $T1$.
  - Not all the rows are involved by regular arrangement
  - Key difference with SELL-C-σ

![Diagram showing SIMD lanes and row elements]
1 Regular Arrangement

• Let \( f[i] \) be the maximum number of **groups** between the first \( i \) rows, \( g[i] \) be the minimum number of **elements** (including padding zeros) between the first \( i \) rows.

• There are two choices for each row \( i \) (i.e. \( f[i] \)): combine row \( i \) with the last \( N-1 \) rows to be a group or not.

\[
f[i] = \begin{cases} 
    f[i - N] + 1, & \text{if combining increases } f[i] \text{ or decreases } g[i] \\
    f[i - 1], & \text{otherwise, not to combine}
\end{cases}
\]

• For those rows that are not to be combined, we compute them row by row as ALBUS does.
2 Adaptive 2D-partitioning

• Fixed size is not the best choices for every matrices, Because of various distributions of elements within different regions
  • The small sizes (i.e. $P \times Q$) make some chunks contain too few elements to improve the data access locality
  • The large sizes represent the large range of irregular data accesses for other chunks and results in poor data locality as well

• irregular matrices or scale-free matrices: The number and the position distribution of elements in each row vary greatly, and a small number of rows contain a great number of elements (power-law distribution). They are more challenging in terms of the computation regularity and load balancing in a multi-threaded environment
2 Adaptive 2D-partitioning

• Solutions: after 2D-partitioning with fixed size $P \times Q$, we merge adjacent chunks horizontally to form a larger chunk. Two rules need to obey:
  • Only the chunks that hold too few elements (less than threshold $T_2$) need to be merged.
  • We don’t merge two chunks that distribute too far
    • Maintain the maximum range of the input vector indices requested by compression rule (next slice)
• Example: $T_2 = 20$ and the maximum span of each block does not exceed four chunks.
  • Because of the first rule, block 3 cannot contain the next chunk
  • Block 1 cannot contain the fifth chunk limited by the second rule

![Diagram of adaptive 2D-partitioning](image)
3 Indices Compression

• Decrease the storage of row indices
• Renumber the row and column indices starting at 0 within a chunk and compress these two indices into an int type (32 bits).
• The sizes of chunks P * Q need to ensure $[\log P] * [\log Q] \leq 32$.
  • This is the compression rule that constrains the merging strategy during 2D-partitioning
4 Load Balancing

• The previous four works (ALBUS, CSR5, VHCC, and CVR) allocate all the elements evenly to each thread

• Because of the random distribution of sparse matrices, an equal number of elements does not guarantee the equal number of corresponding rows, that is, the number of updating operations varies widely
  • Particularly evident in the scale-free matrices

• Example: different threads have a different number of scatter instructions, resulting in different execution times, and the trends of them are basically the same

• Solutions: the number of elements and the number of corresponding rows both need to be considered during load balancing. We count the number of rows of the output vector for each thread and use it as a weight to readjust computation loads of each thread based on experience
Outline

• Introduction
• Previous Work Analyses
• Solutions and Implementation
  • 1 Regular Arrangement
  • 2 Adaptive 2D-partitioning
  • 3 Indices Compression
  • 4 Load Balancing
• Experiment
  • Preprocessing
• Conclusions
Experiment

- **Platform:** Intel Xeon processors (Skylake architecture) with AVX-512 SIMD instructions
- **Datasets:** 30 real-world scale-free and 16 HPC (regular) sparse matrices from the University of Florida Sparse Matrix Collection
- **Comparison:** Regu2D VS ALBUS, CVR, CSR5, and SELL-C-σ
Experiment

<table>
<thead>
<tr>
<th>Speedup of Regu2D compared with</th>
<th>Max(scale-free)</th>
<th>Avg(scale-free)</th>
<th>Max(HPC)</th>
<th>Avg(HPC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALBUS</td>
<td>5.23</td>
<td>1.69</td>
<td>2.40</td>
<td>1.34</td>
</tr>
<tr>
<td>CVR</td>
<td>2.93</td>
<td>1.93</td>
<td>2.85</td>
<td>1.89</td>
</tr>
<tr>
<td>CSR5</td>
<td>2.19</td>
<td>1.40</td>
<td>2.17</td>
<td>1.34</td>
</tr>
<tr>
<td>SELL-C-σ</td>
<td>1.82</td>
<td>1.20</td>
<td>3.05</td>
<td>1.50</td>
</tr>
</tbody>
</table>

Figure 9: The performance of SpMV. The number on the bar chart is the GFLOPS $G$ of Regu2D calculated by $G = 2 \cdot nze / t$ where $nze$ is the number of elements and $t$ is the execution time.
Experiment

- Effects of different optimizations on performance
  - Set ALBUS as a baseline
  - 2D-partitioning degrades 15% of the performance on average because of serious short vector problem. So simply adopting 2D-partitioning does not accelerate vectorization of SpMV
  - Regular arrangement improves 37% on average because it avoids the short vector problem and makes the computation more regular
    - Decrease the number of branch instructions (Figure 10) and the branch miss rate (Figure 11)
    - The number of store operations is lower than that of ALBUS and CVR because we use both scalar store and vector scatter instructions (Figure 12)
Experiment

- Effects of different optimizations on performance
  - Set ALBUS as a baseline
  - 2D-partitioning degrades 15% of the performance on average because of serious short vector problem. So simply adopting 2D-partitioning does not accelerate vectorization of SpMV
  - Regular arrangement improves 37% on average because it avoids the short vector problem and makes the computation more regular
  - Adaptive chunk merging improves 5% on average, due to the improved data locality of chunks that contain a small number of elements
  - Load balancing is used for the matrices with very irregular element distribution, especially for scale-free matrices and brings 9% acceleration
  - Indices compression brings 13% speedup on average because it decreases the amount of memory IO
- All the optimizations improve 49% overall in seven representative matrices

Figure 8: Performance profiling. 2D=2D-partitioning; R=regular arrangement; M=merging of adaptive 2D-partitioning; LB=load balancing; C=indices compression.
Preprocessing

• The original format is the Matrix Market format (column-major COO)

• Regu2D write back the data of the entire matrix twice after 2D-partitioning & regular arrangement
  • on average 1.56 times and 1.51 times longer than that of ALBUS and CSR5
  • SpMV will iterate hundreds or thousands of times, so the larger preprocessing overhead will be amortized

Figure 15: performance of preprocessing of different works.
Outline

• Introduction
• Previous Work Analyses
• Solutions and Implementation
  • 1 Regular Arrangement
  • 2 Adaptive 2D-partitioning
  • 3 Indices Compression
  • 4 Load Balancing
• Experiment
  • Preprocessing
• Conclusions
Conclusions

• Analyze three issues of accelerating SpMV
  • Irregular data access, the short vector problem, and the memory bandwidth

• Propose regu2D and use four methods to solve them
  • Use adaptive 2D-partitioning to increase
    • The data locality, the range of regular arrangement, the vectorization intensity
  • Use regular arrangement to regularize and simplify the computation process
    • Dynamic programming algorithm is proposed to find the optimal regular arrangement
  • Use indices compression to decrease the amount of memory IO
  • Improve load balancing by consider the number of elements and the number of corresponding rows both

• Compared with four works (ALBUS, CVR, CSR5, and SELL-C-σ) and achieve almost the best performance