INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING
ICPP/2021/CHICAGO/USA
acm In-Cooperation
sighpc
AUGUST 9-12, 2021

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

Xiang Fei & Youhui Zhang*

Tsinghua University, Beijing, China

清華大學
Tsinghua University

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

```
Input: ptr[], x[], data[], col[]
Output: y[]
y[i] = 0 ;
for i = 0 to rows − 1 in parallel do
    for j = ptr[i] to ptr[i + 1] do
        y[i] += data[j] ∗ x[col[j]] ;
    end
end
```

Algorithm 1: The CSR-based parallel SpMV

- 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

```
Input: ptr[], x[], data[], col[]
Output: y[]
y[i] = 0 ;
for i = 0 to rows − 1 in parallel do
    for j = ptr[i] to ptr[i + 1] do
        y[i] += data[j] * x[col[j]] ;
    end
end
```

Algorithm 1: The CSR-based parallel SpMV

- 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

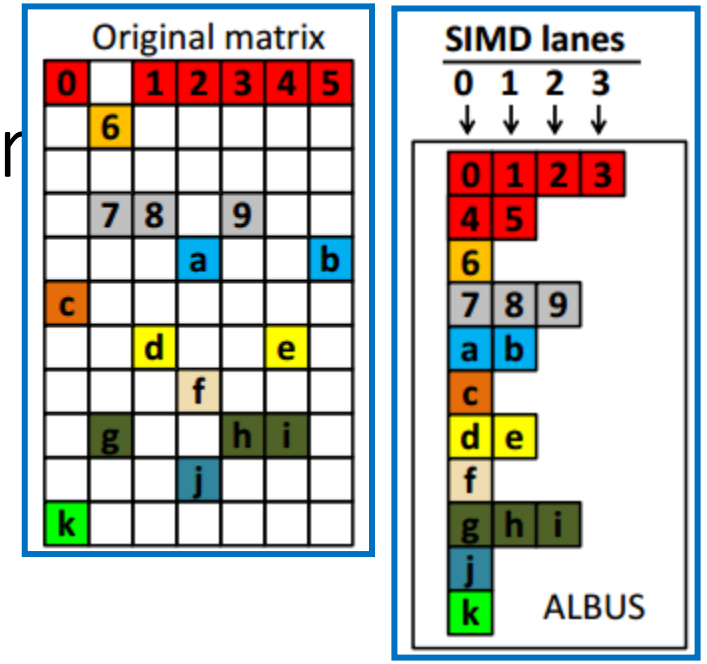
Introduction

```
Input: ptr[], x[], data[], col[]
Output: y[]
y[i] = 0 ;
for i = 0 to rows − 1 in parallel do
    for j = ptr[i] to ptr[i + 1] do
        y[i] += data[j] ∗ x[col[j]] ;
    end
end
```

Algorithm 1: The CSR-based parallel SpMV

- 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

Intr




```
Input: ptr[], x[], data[], col[]
Output: y[]
y[i] = 0 ;
for i = 0 to rows − 1 in parallel do
    for j = ptr[i] to ptr[i + 1] do
        y[i] += data[j] ∗ x[col[j]] ;
    end
end
```

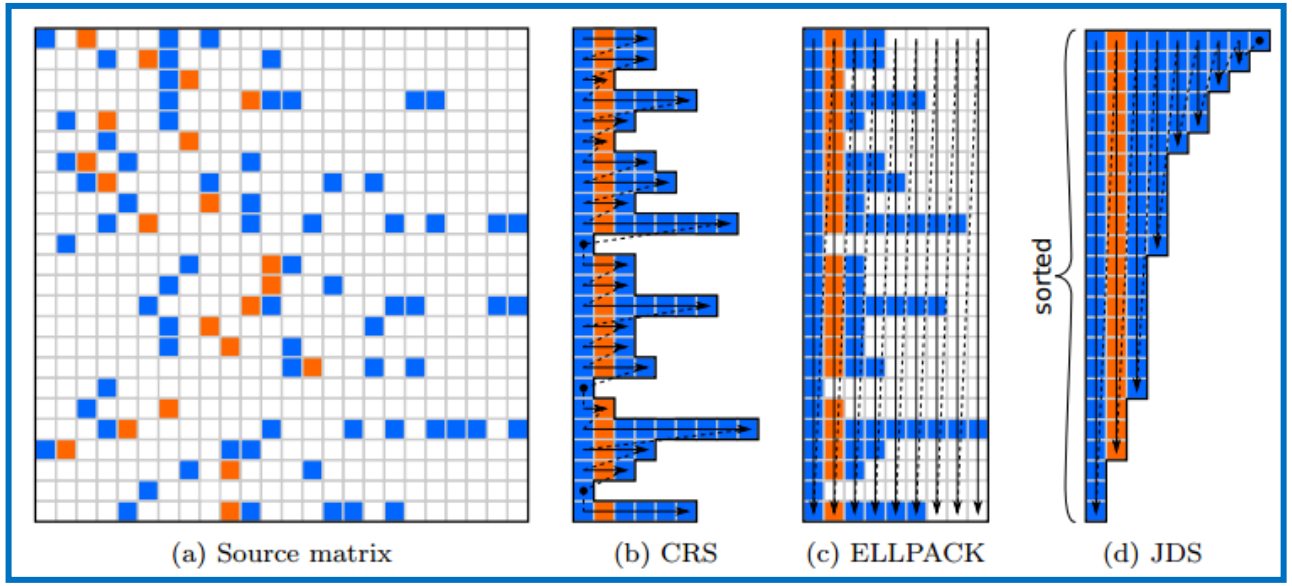
Algorithm 1: The CSR-based parallel SpMV

- Sp iplication (SpMV) is an elementary and n 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

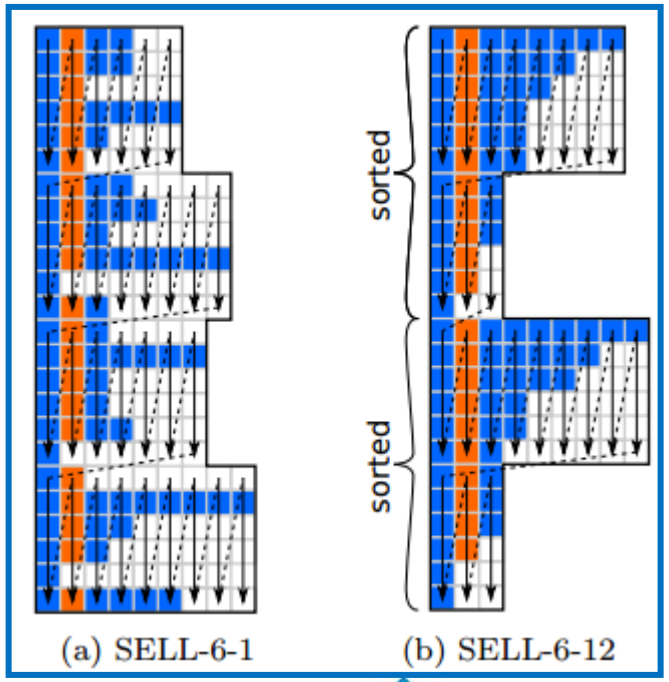
Outline

- 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-σ: (a variant of ELL) still contain many padding zeros, the best value of σ is unknown
- Some new formats are proposed



- Kreutzer, Moritz, et al. "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units." *SIAM Journal on Scientific Computing* 36.5 (2014): C401-C423.

Table 1: Optimization methods of the previous works

	ALBUS	VHCC	CSR5	CVR	Regu2D
2D-partition	x	✓	x	x	✓
Splice rows	x	✓	✓	✓	✓
Use FMA	✓	x	✓	✓	✓
Avoid branches	✓	x	x	x	✓
Use scatter instr.	x	x	✓	x	✓

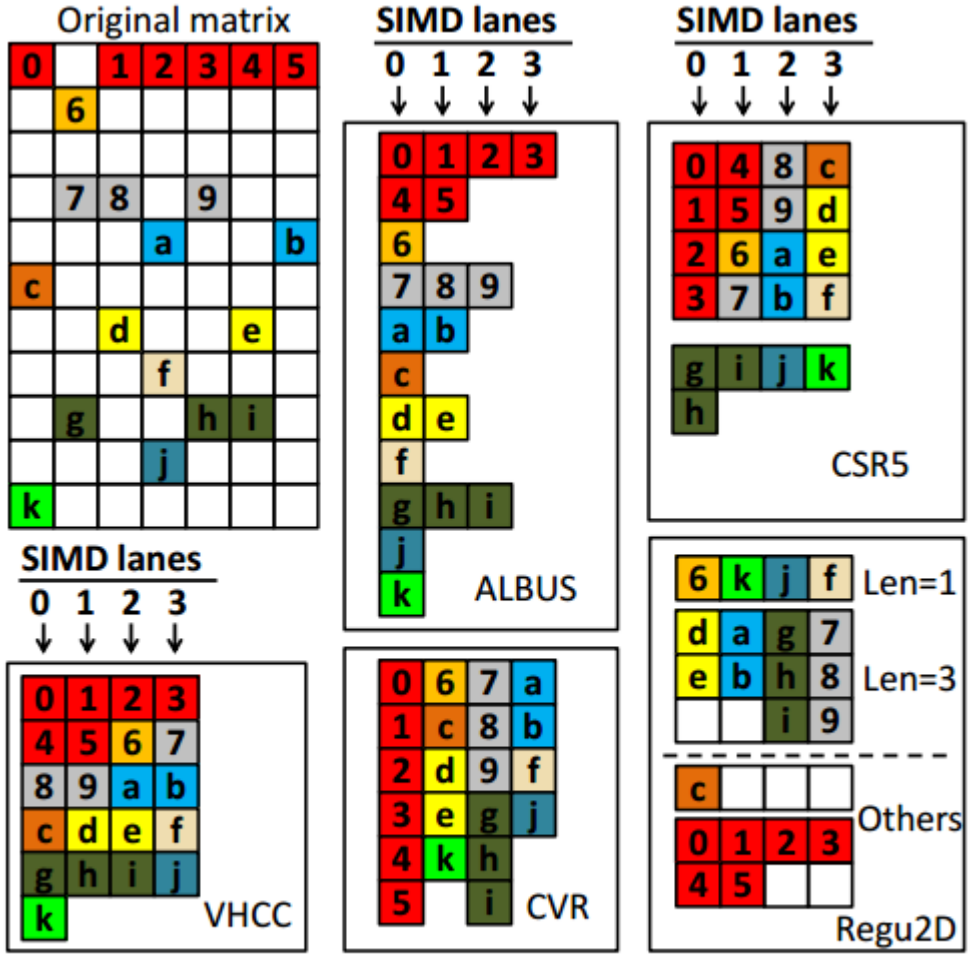


Figure 2: A sparse matrix example and the corresponding data layouts for the previous works and Regu2D

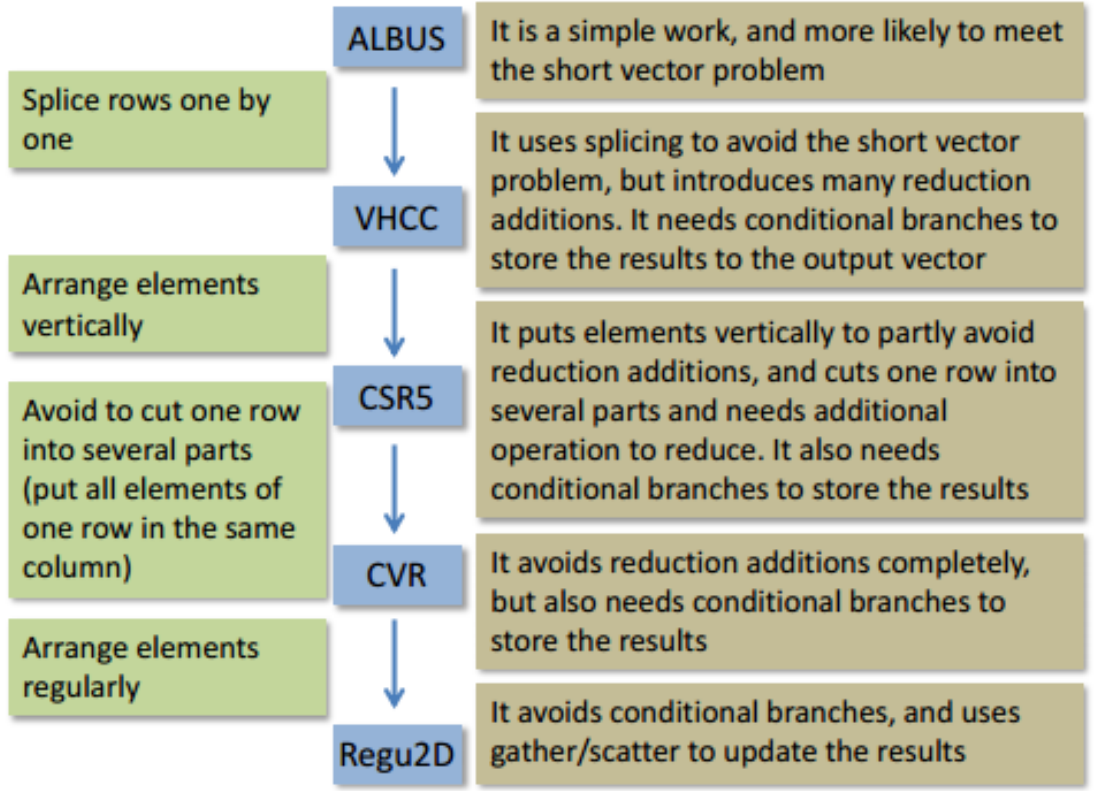


Figure 3: The evolution of the previous works and Regu2D.

Table 1: Optimization methods of the previous works

	ALBUS	VHCC	CSR5	CVR	Regu2D
2D-partition	x	✓	x	x	✓
Splice rows	x	✓	✓	✓	✓
Use FMA	✓	x	✓	✓	✓
Avoid branches	✓	x	x	x	✓
Use scatter instr.	x	x	✓	x	✓

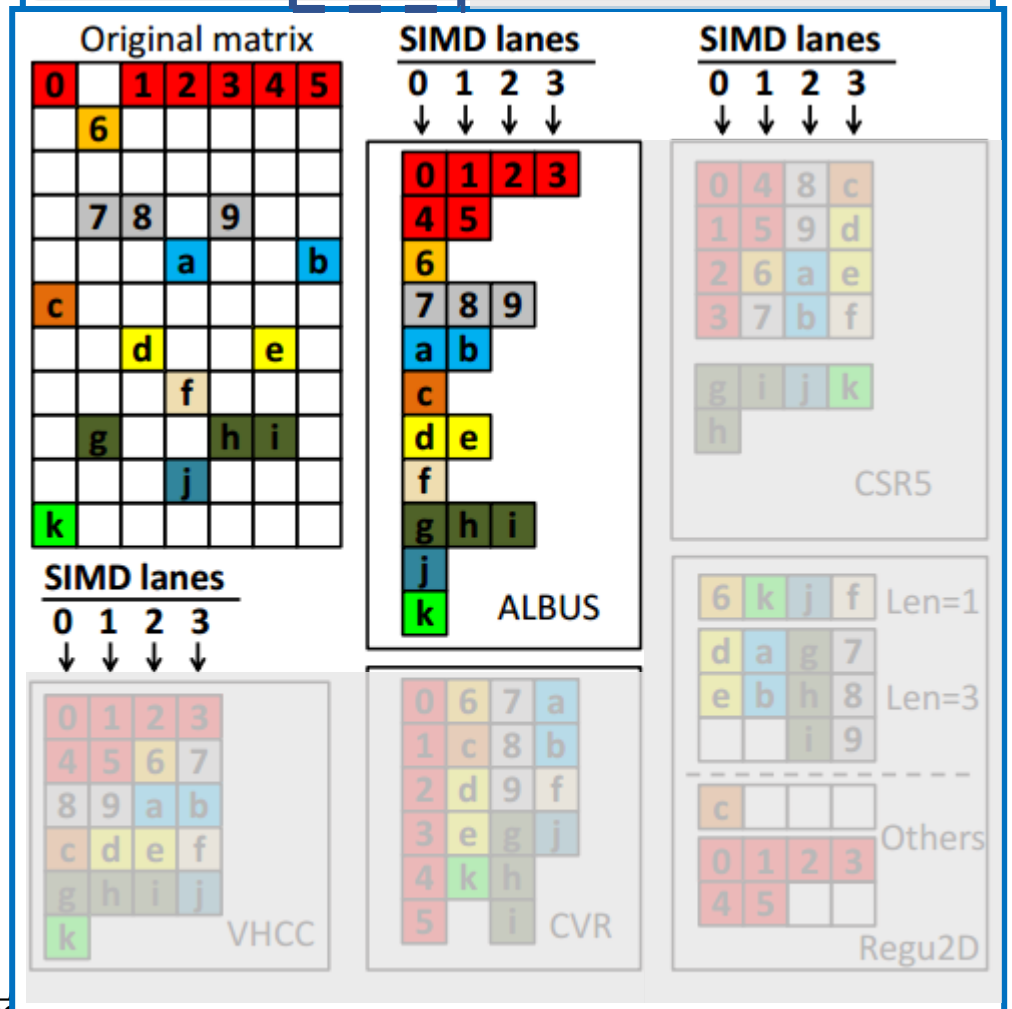


Figure 2: A sparse matrix example and the corresponding data layouts for the previous works and Regu2D

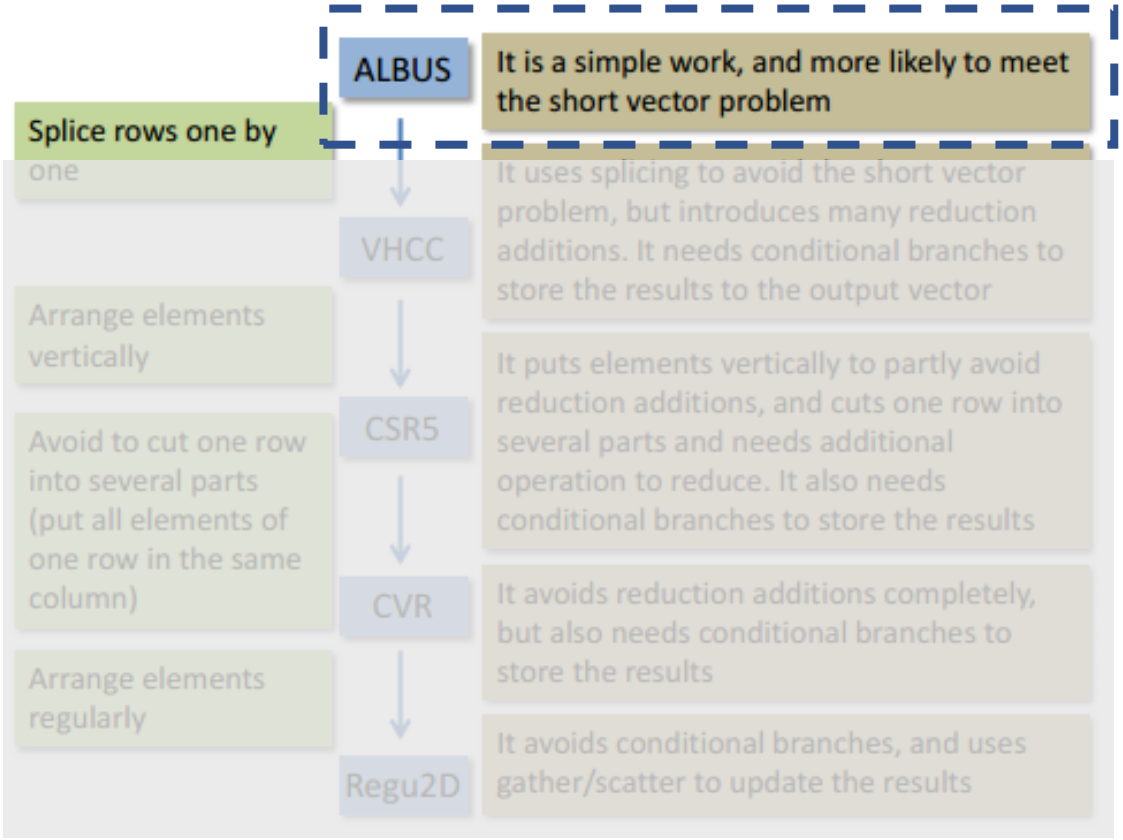


Figure 3: The evolution of the previous works and Regu2D.

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

Algorithm 5: ALBUS kernel

More likely to meet the short vector problem

Table 1: Optimization methods of the previous works

	ALBUS	VHCC	CSR5	CVR	Regu2D
2D-partition	x	✓	x	x	✓
Splice rows	x	✓	✓	✓	✓
Use FMA	✓	x	✓	✓	✓
Avoid branches	✓	x	x	x	✓
Use scatter instr.	x	x	✓	x	✓

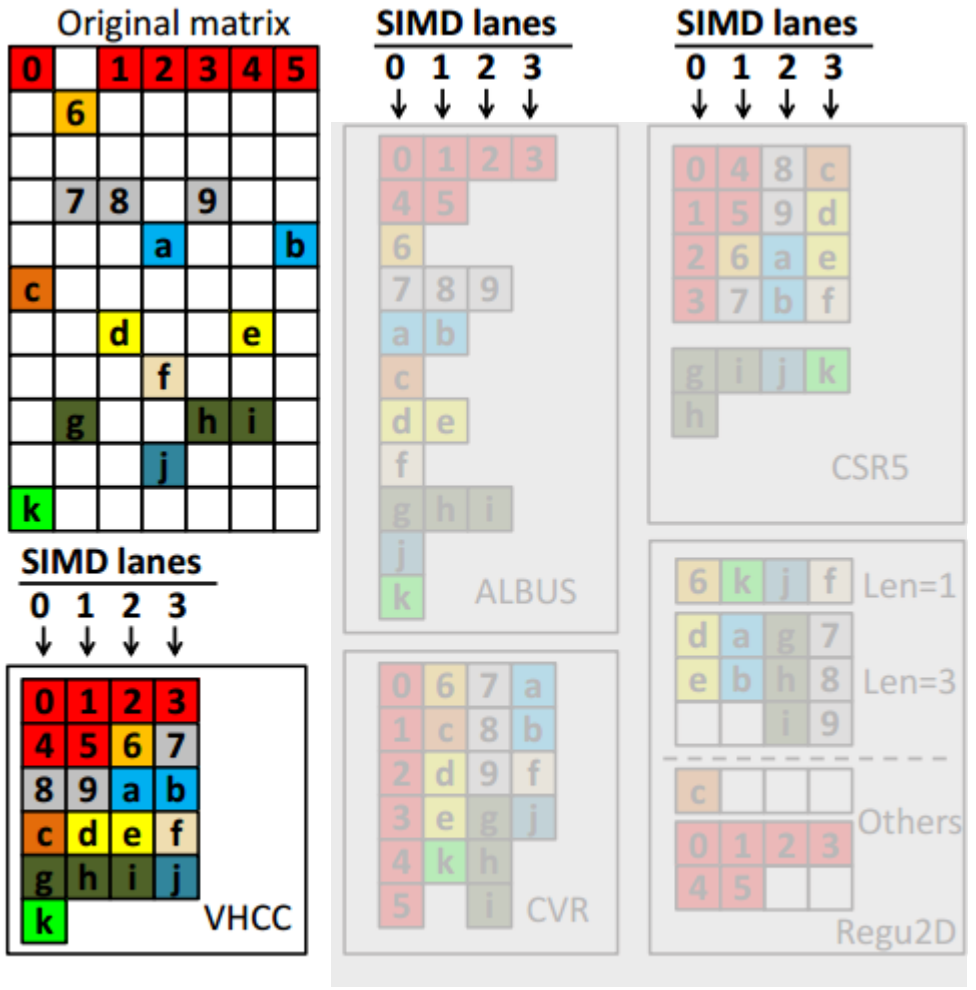
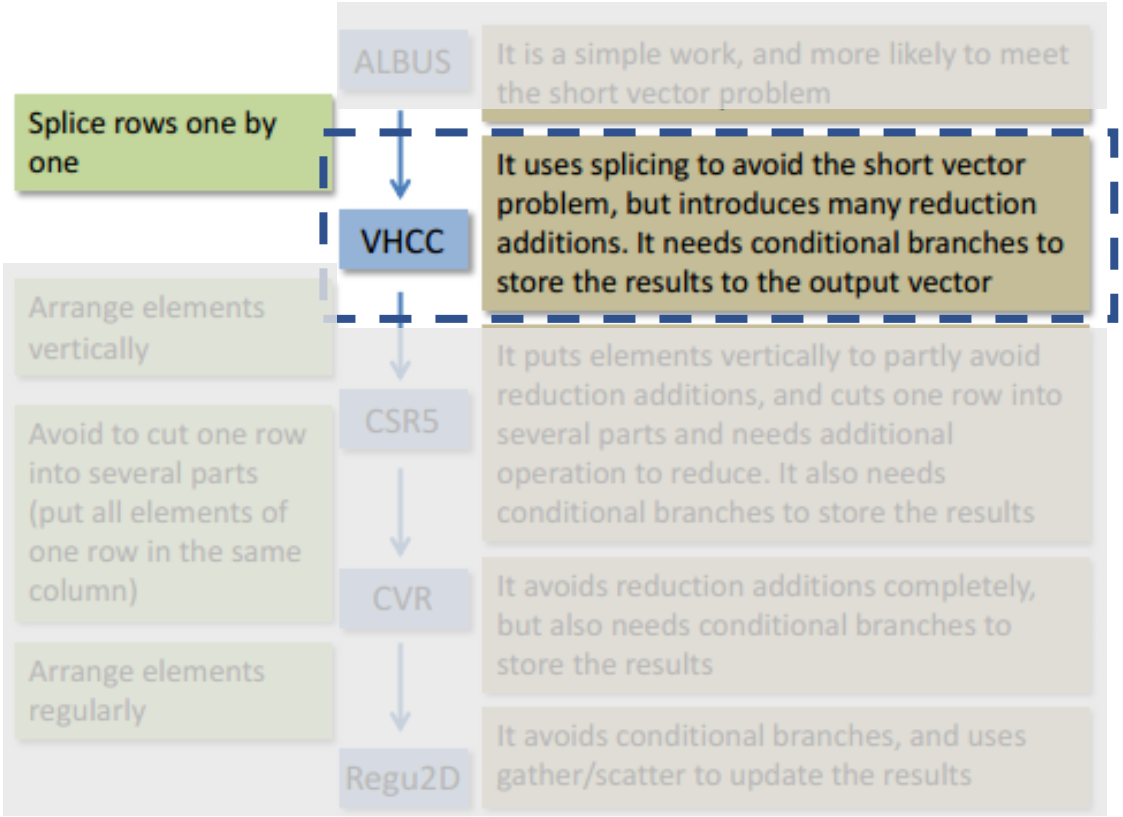


Figure 2: A sparse matrix example and the corresponding data layouts for the previous works and Regu2D



Conditional branches to test whether some vector lanes meet the last element of one row after each multiplication

```
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
```

Reduction additions

Algorithm 4: VHCC kernel

Cooperation sighpc

Table 1: Optimization methods of the previous works

	ALBUS	VHCC	CSR5	CVR	Regu2D
2D-partition	x	✓	x	x	✓
Splice rows	x	✓	✓	✓	✓
Use FMA	✓	x	✓	✓	✓
Avoid branches	✓	x	x	x	✓
Use scatter instr.	x	x	✓	x	✓

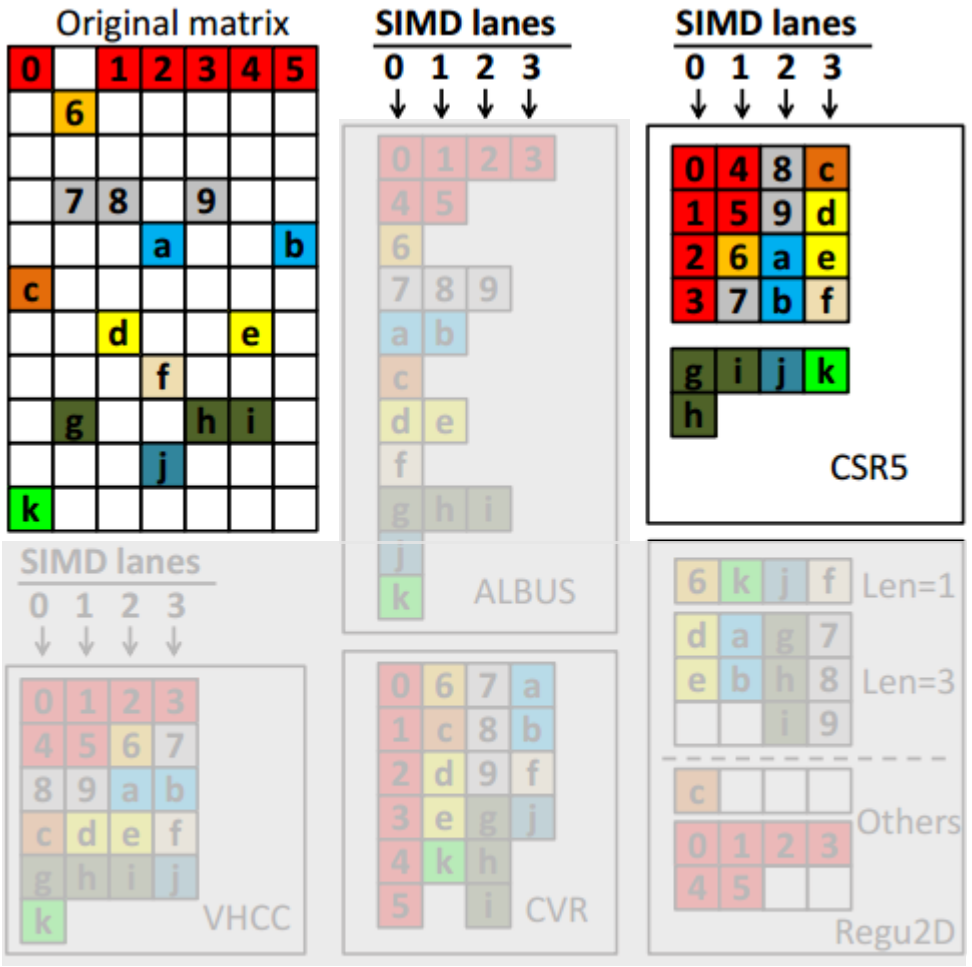
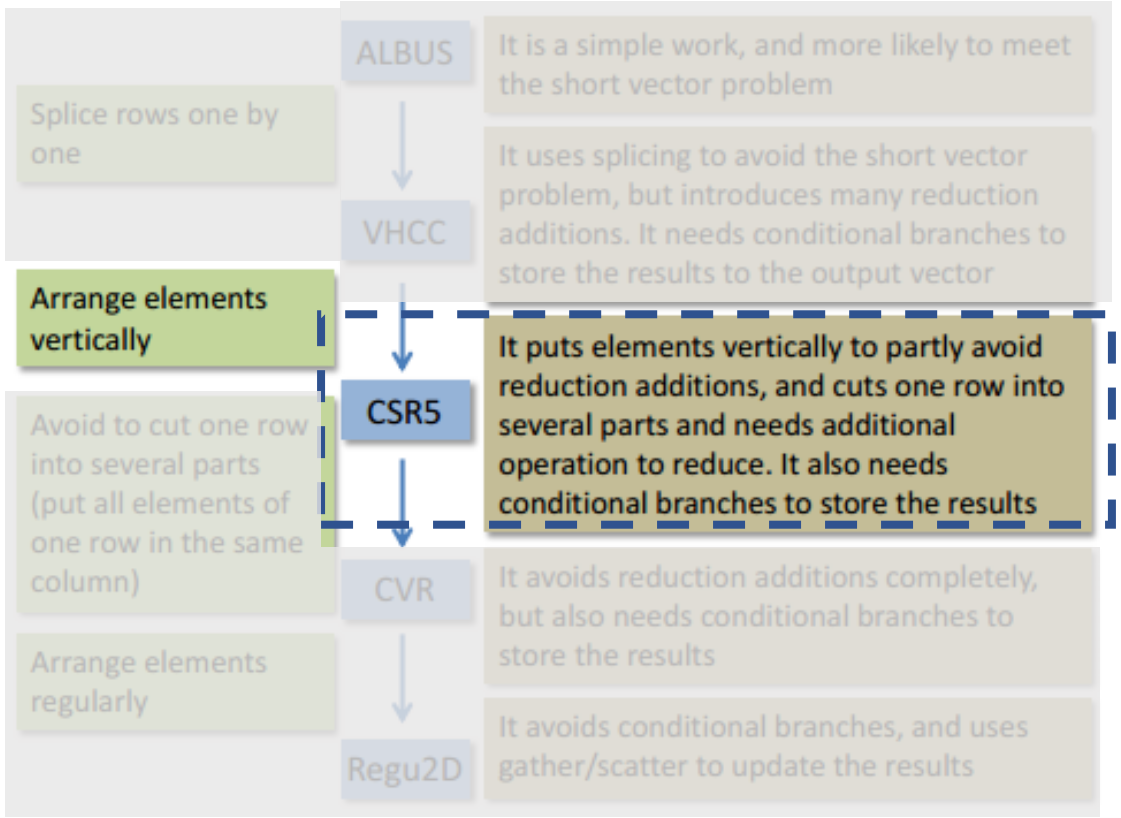


Figure 2: A sparse matrix example and the corresponding data layouts for the previous works and Regu2D



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

	ALBUS	VHCC	CSR5	CVR	Regu2D
2D-partition	x	✓	x	x	✓
Splice rows	x	✓	✓	✓	✓
Use FMA	✓	x	✓	✓	✓
Avoid branches	✓	x	x	x	✓
Use scatter instr.	x	x	✓	x	✓

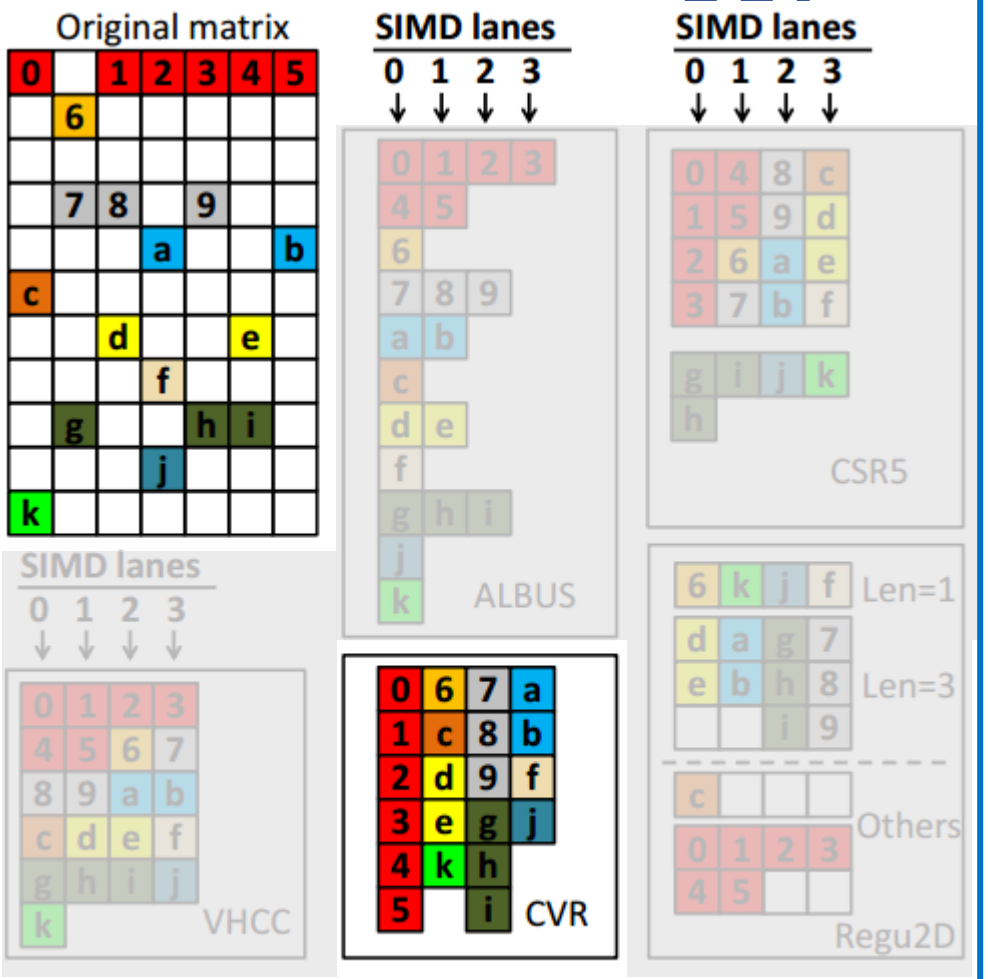
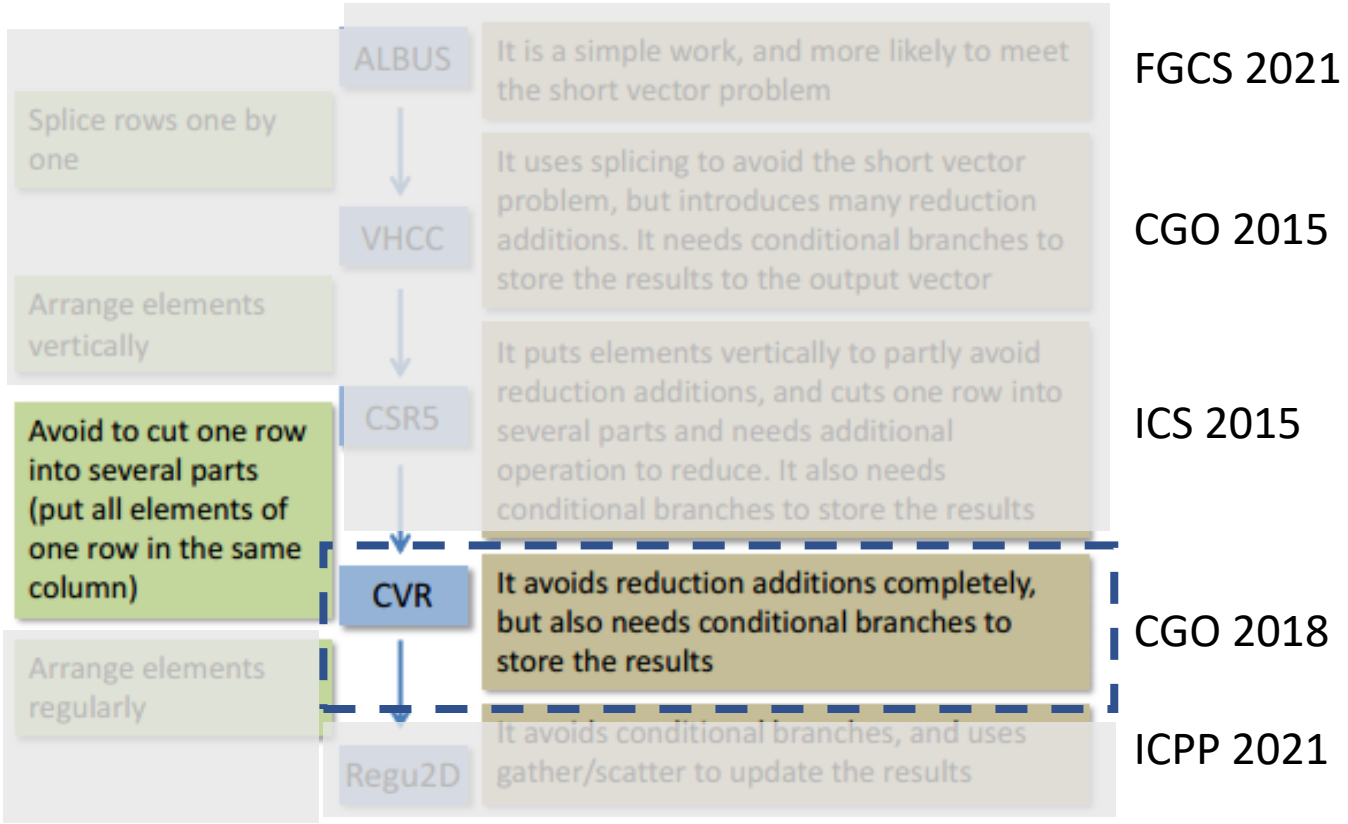


Figure 2: A sparse matrix example and the corresponding data layouts for the previous works and Regu2D



conditional branches

```
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

Algorithm 3: CVR kernel

Table 1: Optimization methods of the previous works

	ALBUS	VHCC	CSR5	CVR	Regu2D
2D-partition	x	✓	x	x	✓
Splice rows	x	✓	✓	✓	✓
Use FMA	✓	x	✓	✓	✓
Avoid branches	✓	x	x	x	✓
Use scatter instr.	x	x	✓	x	✓

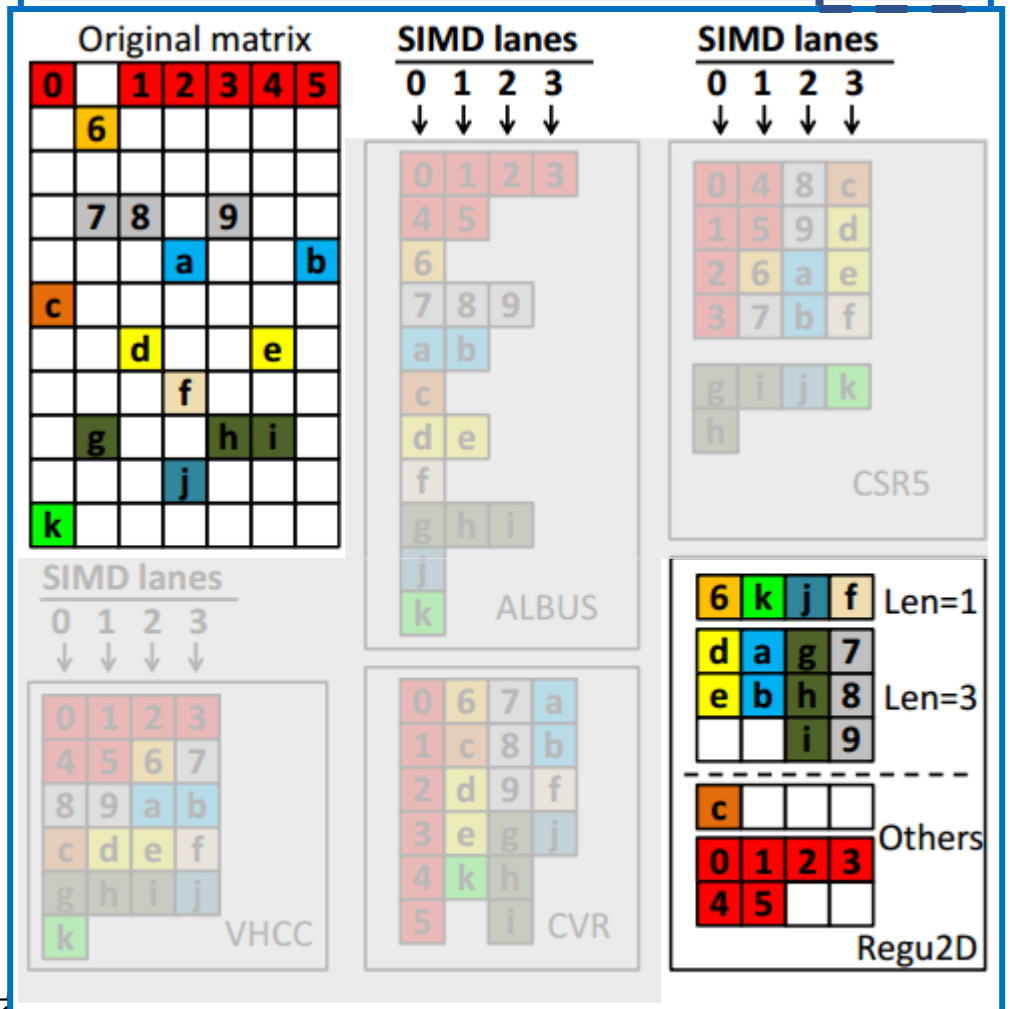
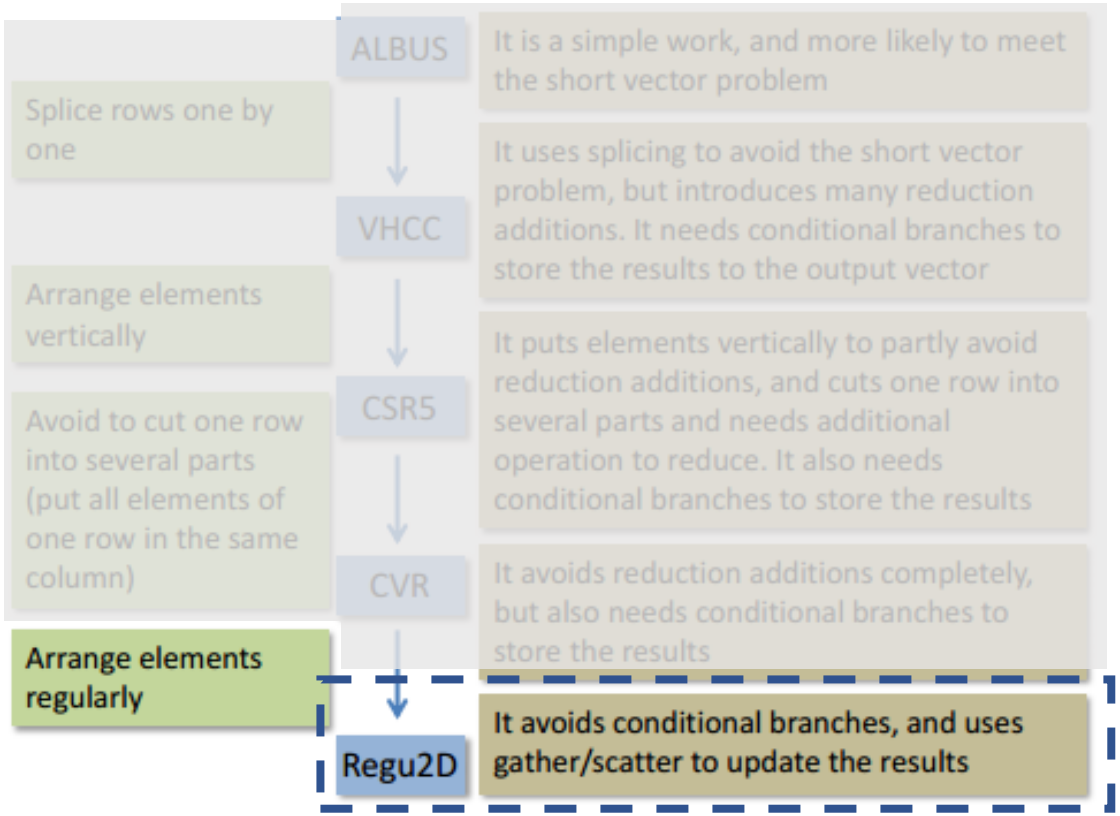


Figure 2: A sparse matrix example and the corresponding data layouts for the previous works and Regu2D

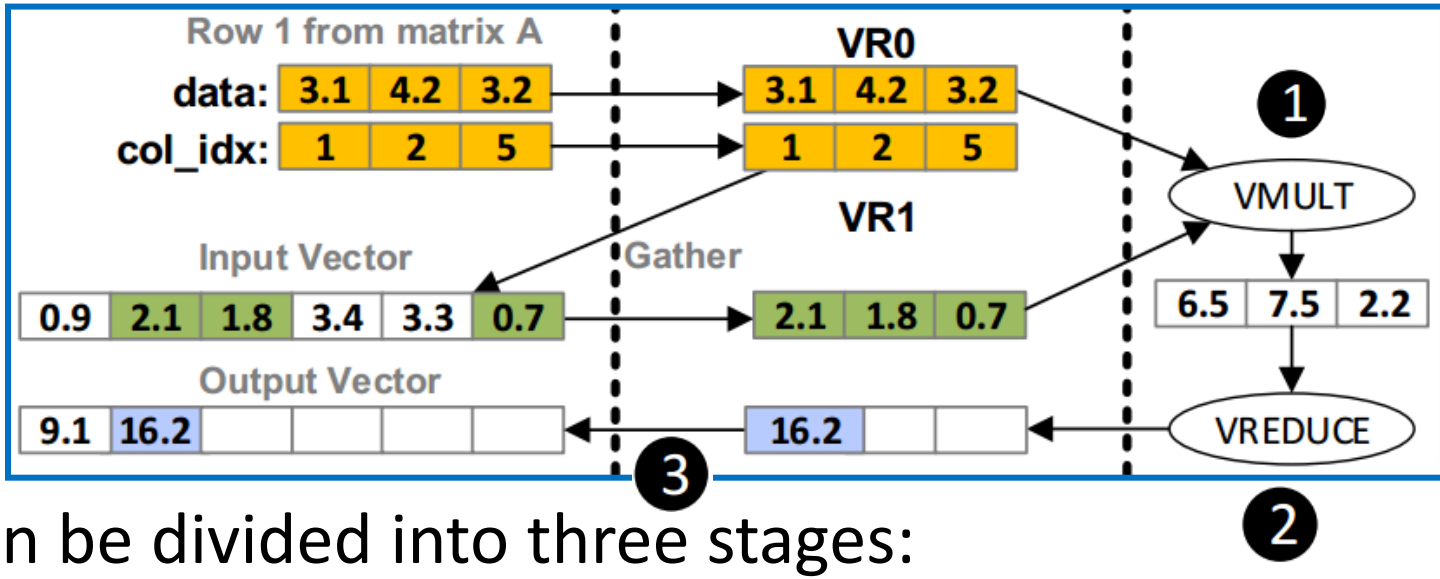



```
for each group do
    len = group length;
    sum = 0;
    for i = 0 to len do
        Load matrix value a and column indices;
        Indexed load input vector b;
        sum+ = a * b;
    end
    Load row indices (if no compression);
    Scatter sum to the output vector;
end
for other rows do
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

Algorithm 7: Computations of Regu2D

Use gather/scatter to update the results

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.

Pavon, Julian, et al. "VIA: A Smart Scratchpad for Vector Units with Application to Sparse Matrix Computations." *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021.

Previous Work Analyses

Table 1: Optimization methods of the previous works

	ALBUS	VHCC	CSR5	CVR	Regu2D
2D-partition	x	✓	x	x	✓
Splice rows	x	✓	✓	✓	✓
Use FMA	✓	x	✓	✓	✓
Avoid branches	✓	x	x	x	✓
Use scatter instr.	x	x	✓	x	✓

- 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

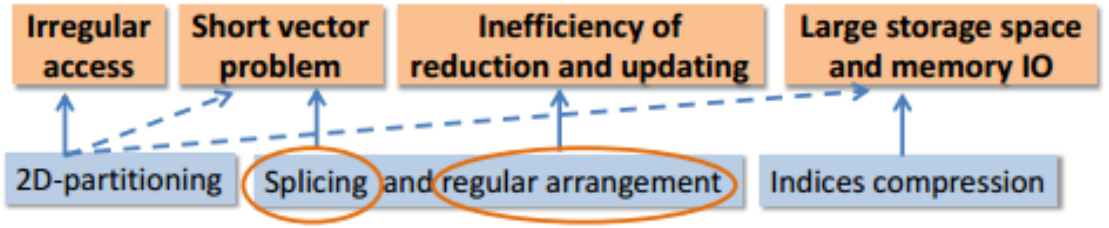
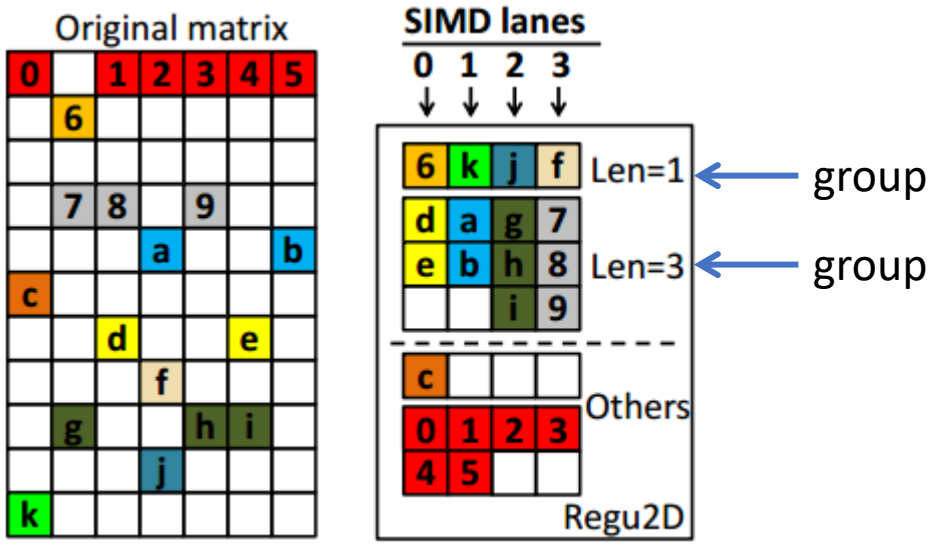


Figure 4: Four challenges and three solutions we adopt. The solid line and the dashed line respectively indicate that the solution has alleviated or aggravated the corresponding problem.

- 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

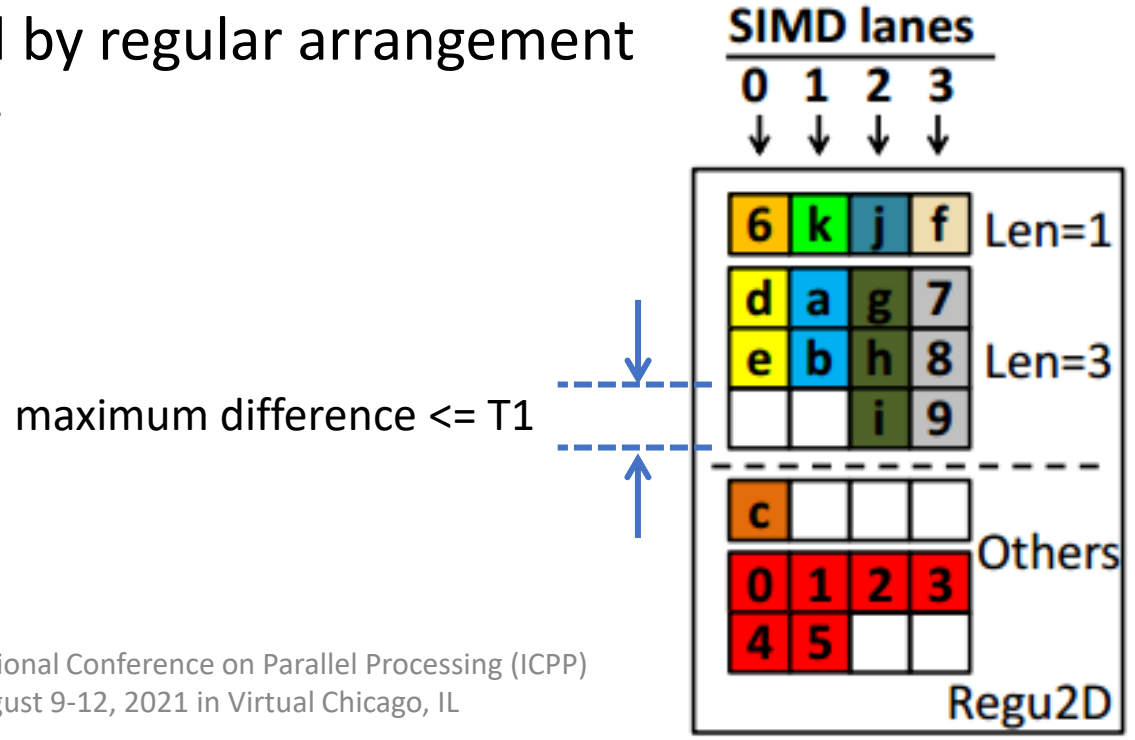
1 Regular Arrangement

- Select *N* rows that have the same number of elements to be a group and then make the *i* th element of row *j* ($0 \le j < N$) occupy lane *j* of vector *i* (arrange non-zero elements ***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

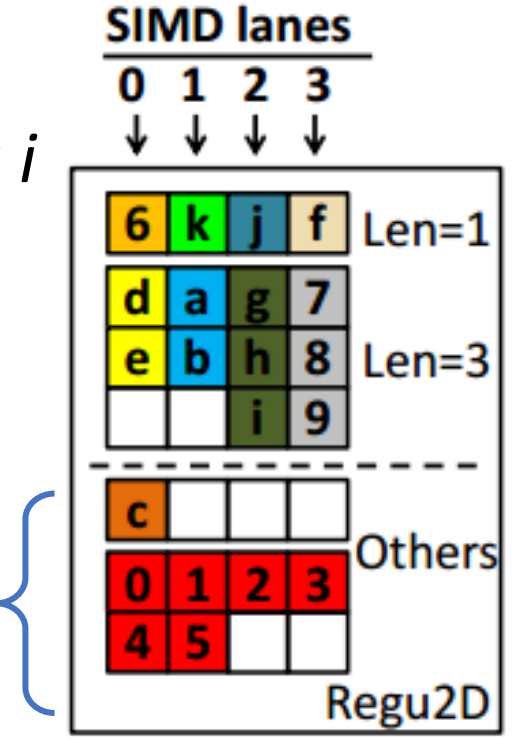


1 Regular Arrangement

- 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.

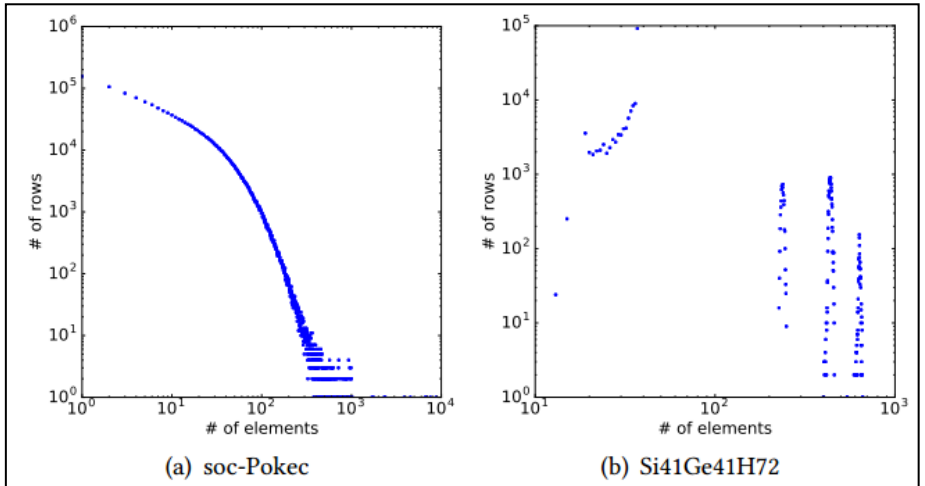
$$f[i] = \begin{cases} f[i-N]+1, & if\ combining\ increases\ f[i]\ or\ decreases\ g[i] \\ f[i-1], & otherwise,\ not\ to\ combine \end{cases}$$

- 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. $P * Q$) 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



(a) soc-Pokec (b) Si41Ge41H72

Figure 1: The distribution of elements of two typical sparse matrices. (Left: irregular or scale-free matrix from the social network field; right: regular matrix from the engineer scientific field)

2 Adaptive 2D-partitioning

- Solutions: after 2D-partitioning with fixed size P ∗ 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 *T2*) 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: T2 = 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

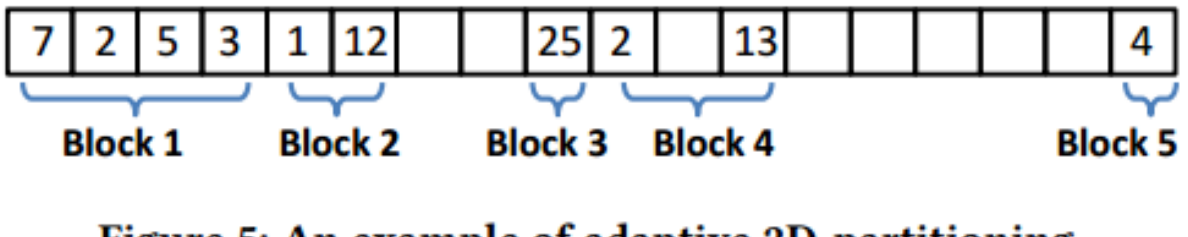


Figure 5: An example of adaptive 2D-partitioning.

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 ⌈logP⌉∗⌈logQ⌉ ≤ 32.
  - This is the compression rule that constrains the merging strategy during 2D-partitioning

4 Load Balancing

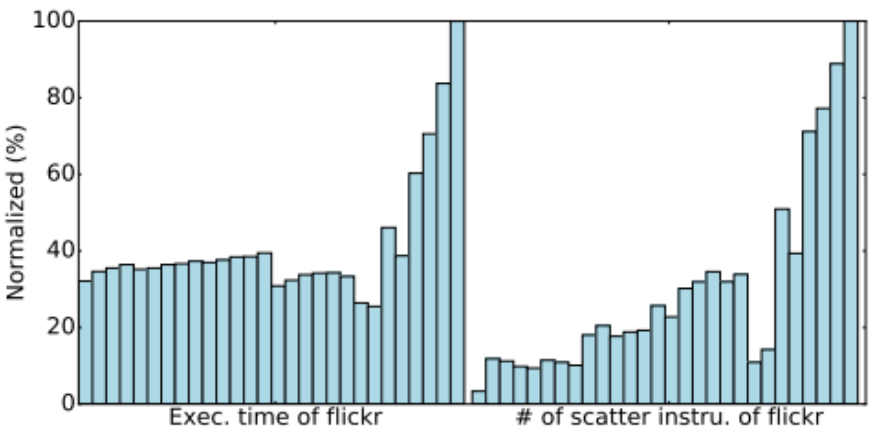


Figure 6: Statistics of execution time and # of scatter instructions of matrix *flickr* with 28 threads.

- 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

Table 2: Hardware & software configurations

CPU	Intel(R) Xeon(R) Gold 6132, 2.60GHz, 2CPU*14cores
Memory	92GB, DDR4, 2666 MT/s
Caches	L1D=32KB, L1I=32KB, L2=1MB, L3=19.25MB
Compiler	icc 19.0.5.281 (-O3 -fopenmp -xCORE-AVX512)
OS	Red Hat 4.8.5-36

- 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

Scenario	ID	matrix name	rows	columns	nonzeros
Engineer scientific (regular matrix)	1	ASIC_680k	682,862	682,862	3,871,773
	2 *	cant	62,451	62,451	4,007,383
	3 *	consph	83,334	83,334	6,010,480
	4 *	crankseg_2	63,838	63,838	14,148,858
	5	dense4k	4,096	4,096	16,777,216
	6	FullChip	2,987,012	2,987,012	26,621,990
	7 *	Ga41As41H72	268,096	268,096	18,488,476
	8 *	inline_1	503,712	503,712	36,816,342
	9 *	ldoor	952,203	952,203	46,522,475
	10	mac_econ_fwd500	206,500	206,500	1,273,389
	11 *	mip1	66,463	66,463	10,352,819
	12 *	mouse_gene	45,101	45,101	28,967,291
	13 *	pdb1HYS	36,417	36,417	4,344,765
	14 *	pwtk	217,918	217,918	11,634,424
	15	rma10	46,835	46,835	2,374,001
	16 *	Si41Ge41H72	185,639	185,639	15,011,265
Web graph**	17	12month1	12,471	872,622	22,624,727
	18	amazon0312	400,727	400,727	3,200,440
	19	amazon-2008	735,323	735,323	5,158,388
	20	cnr-2000	325,557	325,557	3,216,152
	21 *	com-Youtube	1,134,890	1,134,890	5,975,248
	22	eu-2005	862,664	862,664	19,235,140
	23 *	hollywood-2009	1,139,905	1,139,905	113,891,327
	24	IMDB	428,440	896,308	3,782,463
	25	in-2004	1,382,908	1,382,908	16,917,053
	26	NotreDame_actors	392,400	127,823	1,470,404
	27	uk-2002	18,520,486	18,520,486	298,113,762
	28	web-Google	916,428	916,428	5,105,039
	29	web-Stanford	281,903	281,903	2,312,497
Wiki**	30	wiki-Talk	2,394,385	2,394,385	5,021,410
	31	wiki-topcats	1,791,489	1,791,489	28,511,807
Citation**	32 *	citationCiteseer	268,495	268,495	2,313,294
	33 *	com-DBLP	317,080	317,080	2,099,732
	34 *	coPapersCiteseer	434,102	434,102	32,073,440
	35	patents	3,774,768	3,774,768	14,970,767
Road**	36	rail4284	4,284	1,096,894	11,284,032
	37 *	road_usa	23,947,347	23,947,347	57,708,624
	38 *	roadNet-CA	1,971,281	1,971,281	5,533,214
Routing**	39 *	as-Skitter	1,696,415	1,696,415	22,190,596
Social network**	40 *	com-Orkut	3,072,441	3,072,441	234,370,166
	41	flickr	820,878	820,878	9,837,214
	42	higgs-twitter	456,626	456,626	14,855,842
	43	soc-LiveJournal1	4,847,571	4,847,571	68,993,773
	44	soc-Pokec	1,632,803	1,632,803	30,622,564
	45	soc-sign-epinions	131,828	131,828	841,372
	46	twitter-2010	41,652,230	41,652,230	1,468,365,182

* Symmetric matrix; ** Scale-free matrix

Figure 7: The benchmark suite.

Experiment

Speedup of Regu2D compared with	Max(scale-free)	Avg(scale-free)	Max(HPC)	Avg(HPC)
ALBUS	5.23	1.69	2.40	1.34
CVR	2.93	1.93	2.85	1.89
CSR5	2.19	1.40	2.17	1.34
SELL-C-σ	1.82	1.20	3.05	1.50

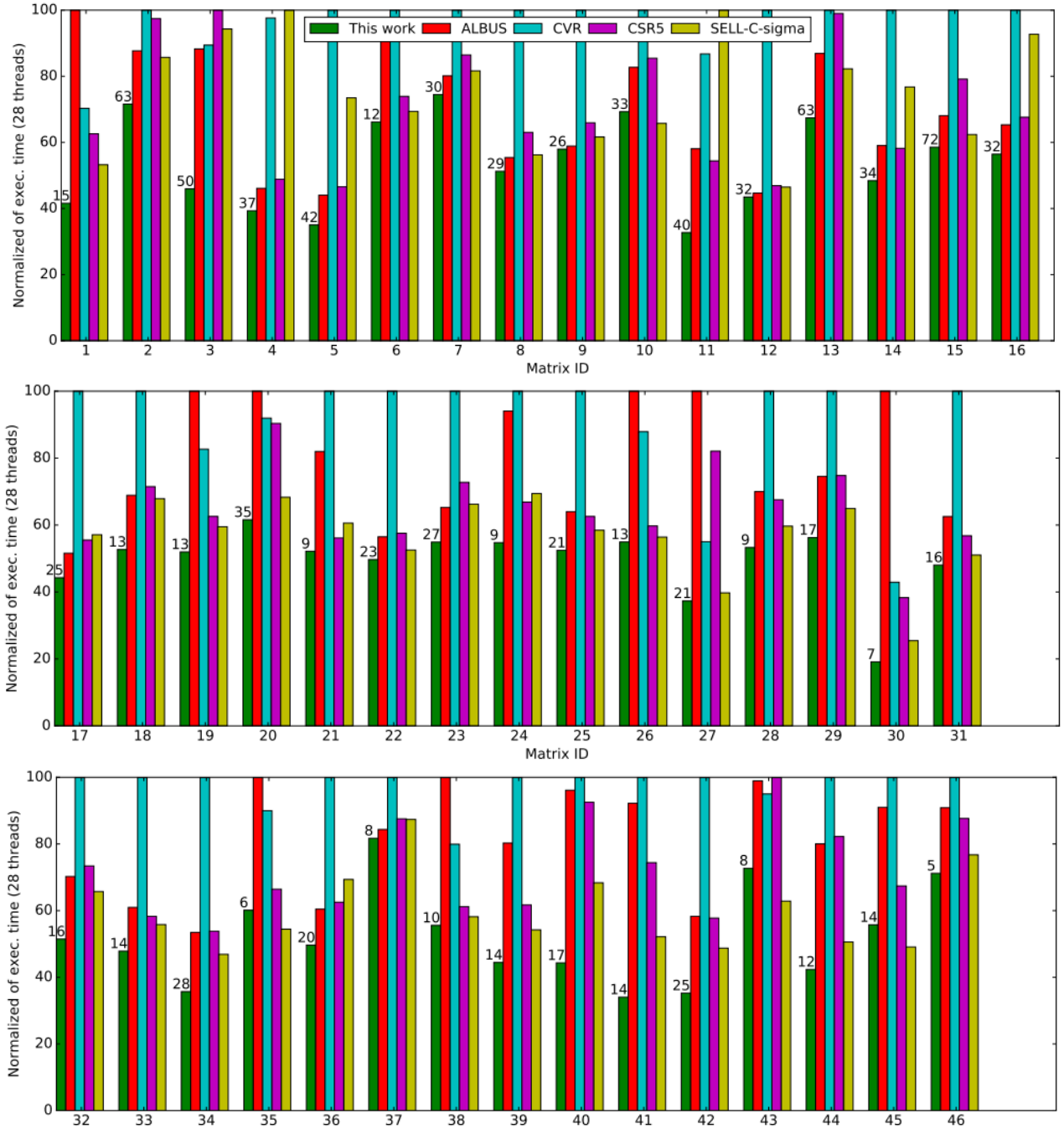


Figure 9: The performance of SpMV. The number on the bar chart is the GFLOPS G of Regu2D calculated by $G = 2 * nnz/t$ where nnz is # of elements and t is the execution time.

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)

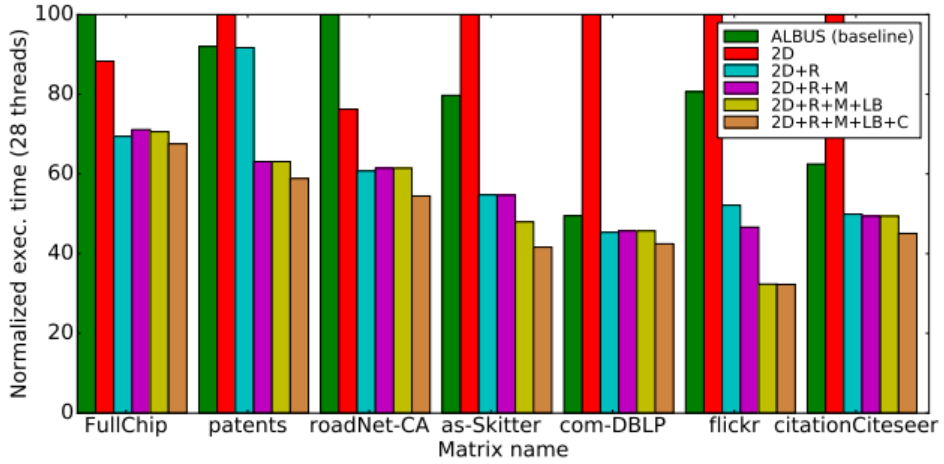


Figure 8: Performance profiling. 2D=2D-partitioning; R=regular arrangement; M=merging of adaptive 2D-partitioning; LB=load balancing; C=indices compression.

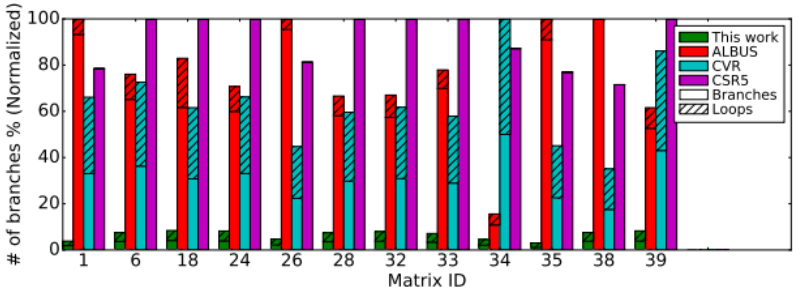


Figure 10: # of branches of different works. One loop is counted as one branch regardless of the number of iterations.

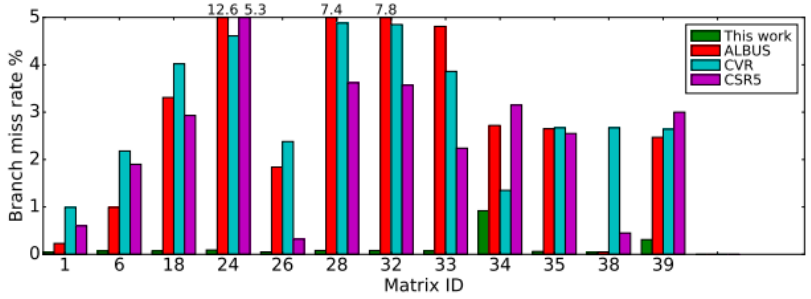


Figure 11: Branch miss rates of different works.

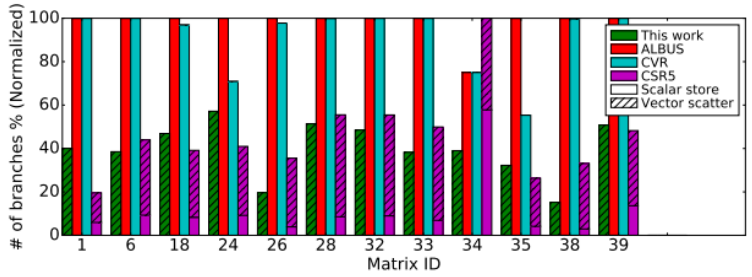


Figure 12: # of store instructions of different works.

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

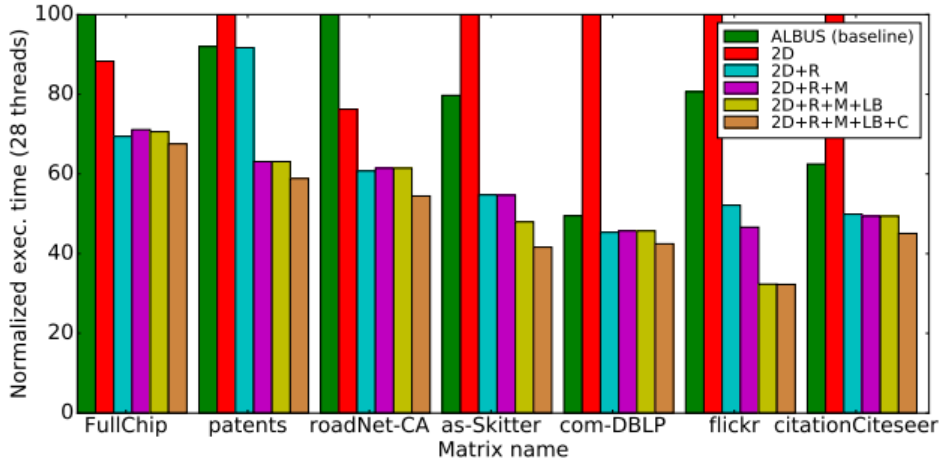


Figure 8: Performance profiling. 2D=2D-partitioning; R=regular arrangement; M=merging of adaptive 2D-partitioning; LB=load balancing; C=indices compression.

Preprocessing

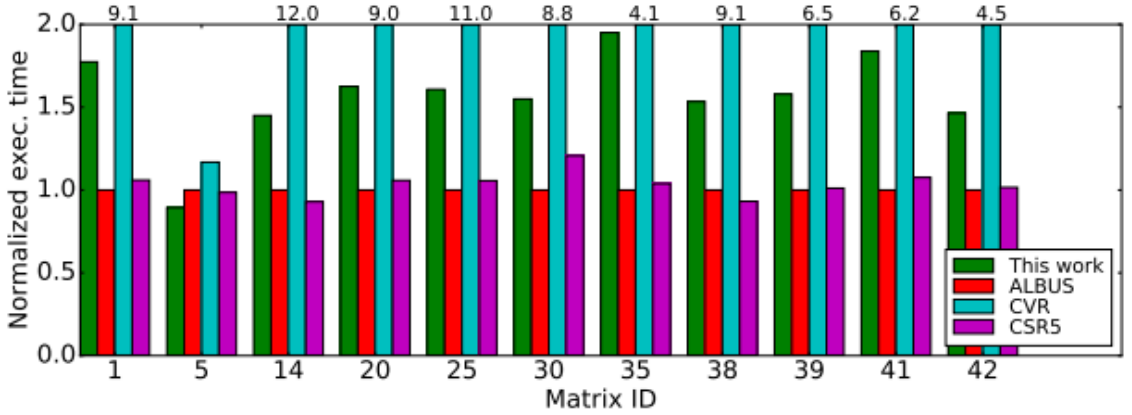


Figure 15: performance of preprocessing of different works.

- 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

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