Adaptive Multi-level Blocking Optimization for Sparse Matrix Vector Multiplication on GPU

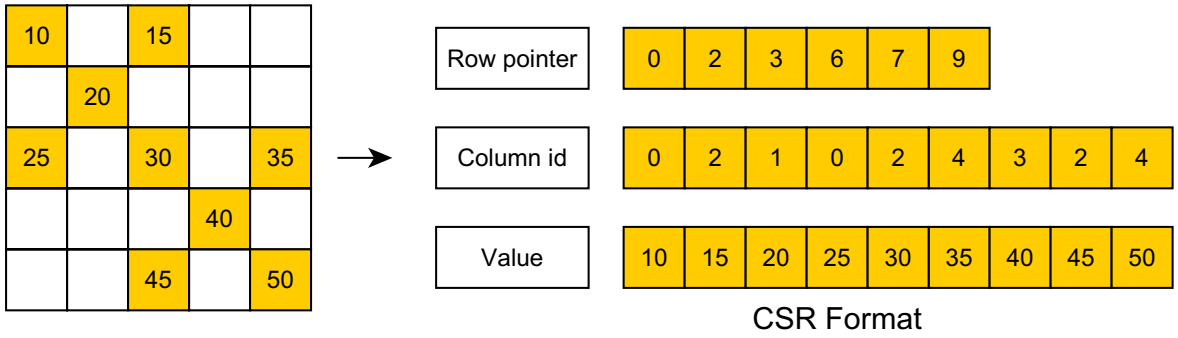
Yusuke Nagasaka, Akira Nukada, Satoshi Matsuoka

Tokyo Institute of Technology

Sparse Matrix Computation

- Large and sparse equations
  - Generated by FEM
  - Sparse matrix vector multiplication (SpMV) requires much execution time in solving equation
    - Using sparse format, which removes needless zero elements
      - Indirect memory access to input vector element
        - » Random memory access : Frequent cache misses
      - Require index of row and column for each non-zero element
        - » Aggravate memory boundness of Mat-vec

      => Performance degradation of SpMV



SpMV on Many-core Processors

- Acceleration by high parallelism and memory bandwidth
  - Small cache ⇔ Enormous executing threads
    - More frequent cache misses
  - Performance is strongly limited by memory bandwidth
    - Not exploiting many core processors

Sparse Format

- Compressing needless zero elements
  - Storing only non-zero elements
  - Reducing memory usage and computation
- Each format has variety of characteristics
  - Being suited to architectures and given matrices

	Memory Layout	Load-balance	Memory Access to Matrix Data	Locality of access to input vector
CSR	Row-major	✓	-	-
ELLPACK	Column-major	-	-	-
SELL-C-σ	Column-major	-	✓	-
Segmented, NUS	Column-major	✓	-	✓✓
BCCOO (yaSpMV)	Row-major	✓✓	✓✓	-

Sparse Format

- Reducing total memory access is important to alleviate the memory-boundness
  - However, existing work does not reduce enough
  - We should consider the **access to both matrix data and input vector elements**
    - Reduction of total memory access

	Memory Layout	Load-balance	Memory Access to Matrix Data	Locality of access to input vector
CSR	Row-major	✓	-	-
ELLPACK	Column-major	-	-	-
SELL-C-σ	Column-major	-	✓	-
Segmented, NUS	Column-major	✓	-	✓✓
BCCOO (yaSpMV)	Row-major	✓✓	✓✓	-

Contribution

- We propose new sparse matrix format for GPU
  - Adaptive Multi-level Blocking (AMB) format
    - Several optimization techniques such as division and blocking
    - Aggressively reducing total memory access including both matrix data and input vector in SpMV to improve performance
    - Precisely predicting total memory access and performance of SpMV
  - For 32 matrix datasets taken from Florida sparse matrix collection, achieve speedups of
    - Up to x2.92 compared to cuSPARSE (x1.74 on average)
    - Up to x1.40 compared to yaSpMV (x1.13 on average)

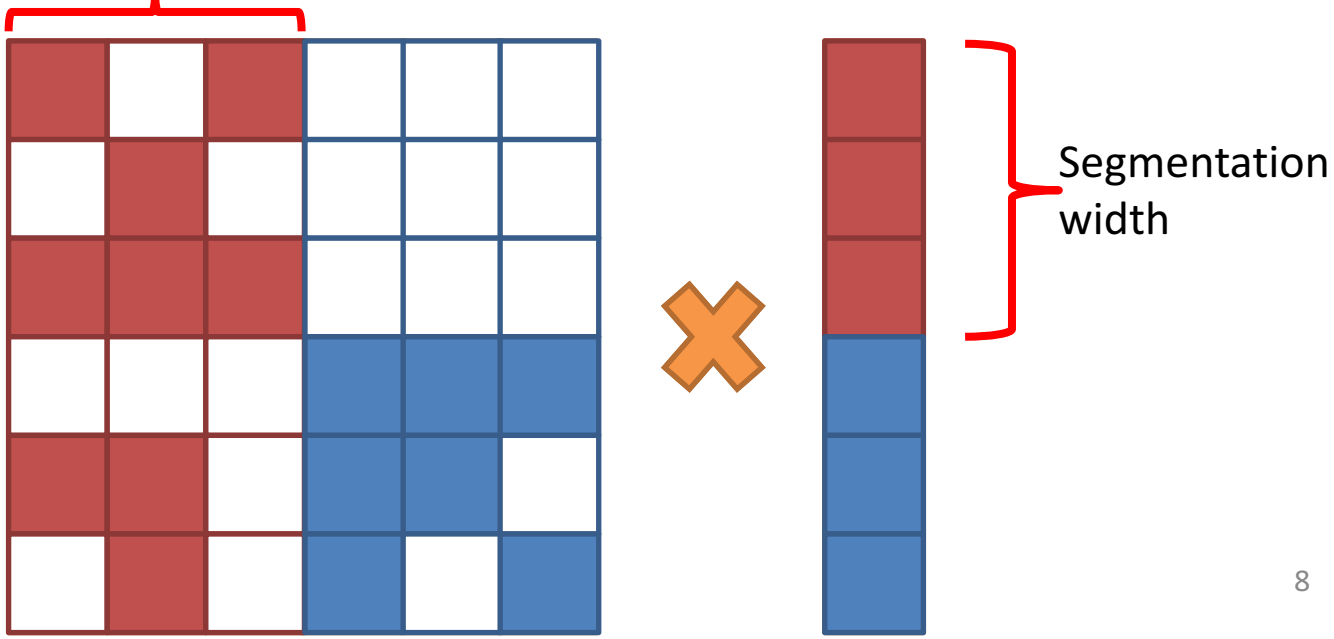
Adaptive Multi-level Blocking (AMB)

- Largely reducing total memory access in SpMV
- Constructed in multi-level division and blocking
  - Improving the locality of the access to input vector by dividing matrix and input vector
  - Mainly focusing on compression of column index

	Memory Layout	Load-balance	Memory Access to Matrix Data	Locality of access to input vector
CSR	Row-major	✓	-	-
ELLPACK	Column-major	-	-	-
SELL-C-σ	Column-major	-	✓	-
Segmented, NUS	Column-major	✓	-	✓✓
BCCOO (yaSpMV)	Row-major	✓✓	✓✓	-
AMB（Proposal）	Column-major	✓	✓✓	✓✓

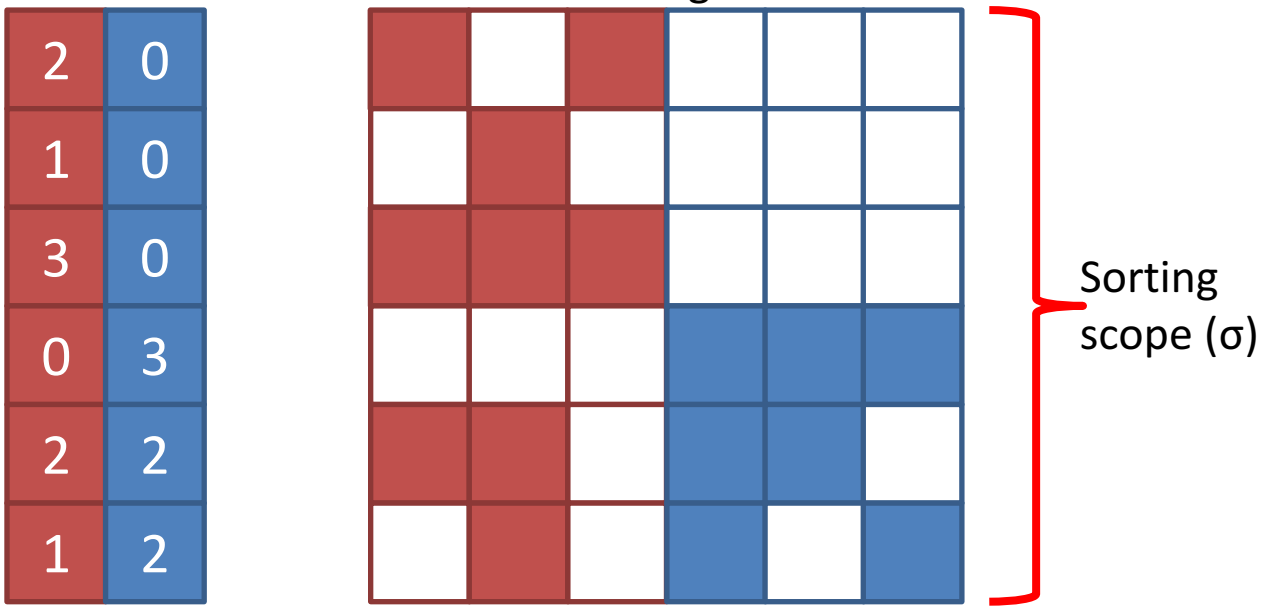
AMB (Adaptive Multi-level Blocking)

- **Level 1 : Column-wise Segmentation**
  - Improving the locality of the access to input vector element in SpMV
  - Segmentation width is less than 65536 (= 2^16) columns
    - For the compression of column index in the second level division



AMB (Adaptive Multi-level Blocking)

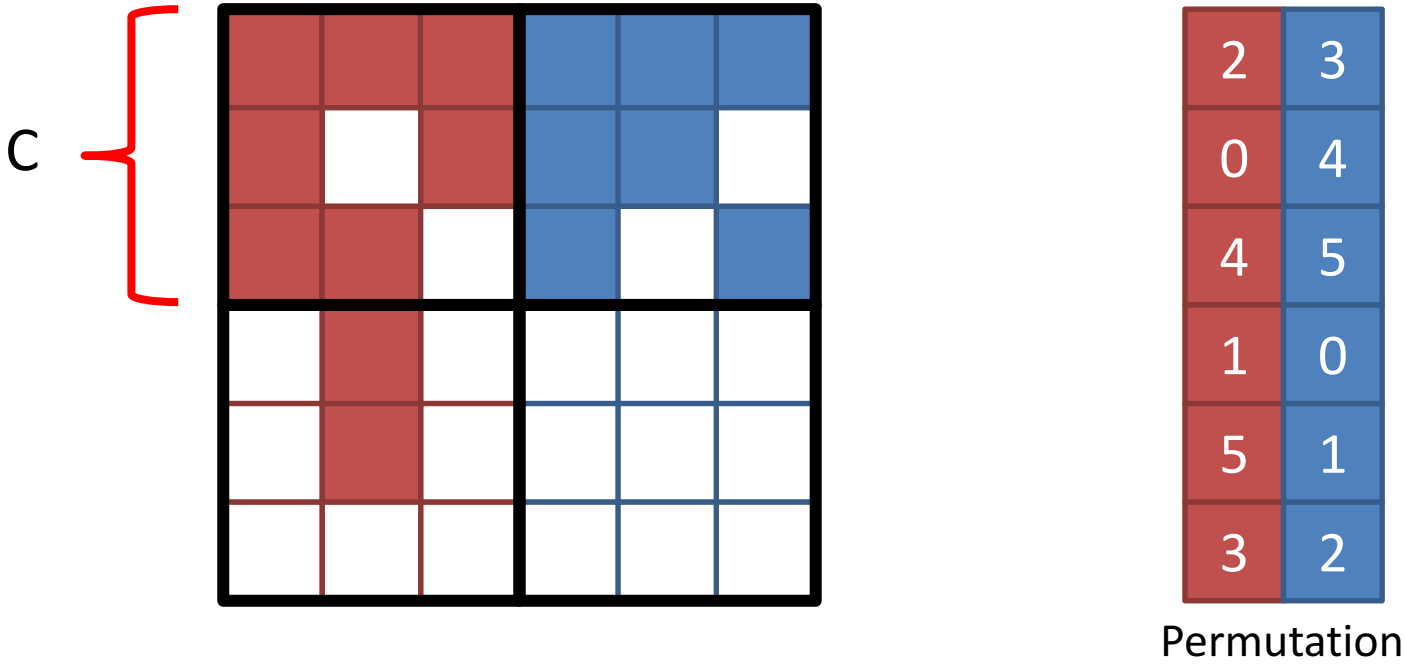
- **Level 2 : <u>Row-wise segmentation</u> and compression**
  - Converting each sub-matrix to SELL-C-σ
    - Sorting rows by the number of non-zero elements per row
      - Sorting scope (σ) is limited : less than 32768 (2^15)
        - » σ =6 in this figure



9

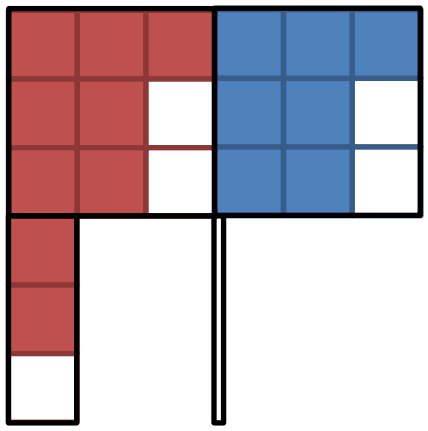
AMB (Adaptive Multi-level Blocking)

- **Level 2 : <u>Row-wise segmentation</u> and compression**
  - Converting each sub-matrix to SELL-C-σ
    - Row-wise division by C rows (C = 3 in this figure)
      - C is set based on architecture (C=32 in GPU case)

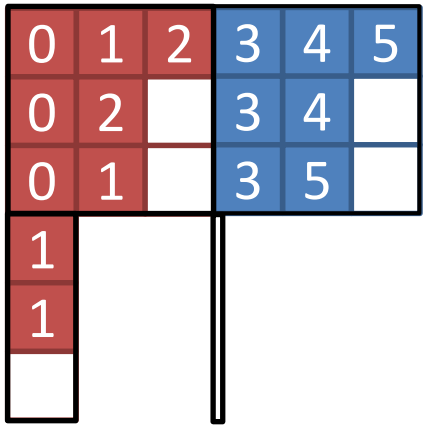


AMB (Adaptive Multi-level Blocking)

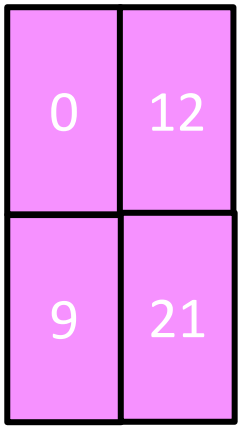
- **Level 2 : <u>Row-wise segmentation</u> and compression**
  - Converting each sub-matrix to SELL-C-σ
    - Converting each chunk to ELLPACK
    - More memory access to CS, CL and Permutation is required compared to original SELL-C-σ (w/o Level 1 segmentation)



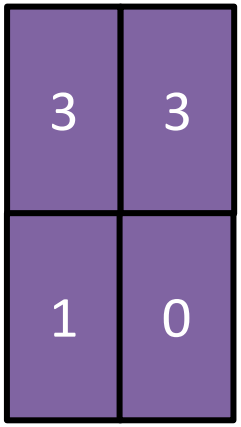
Value data



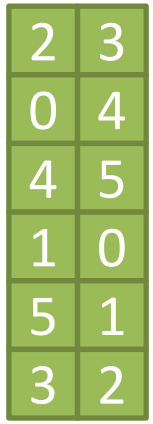
Column Index



CS (Chunk Starting offset)



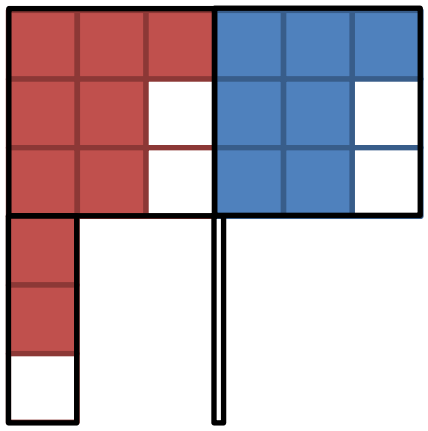
CL (Chunk Length)



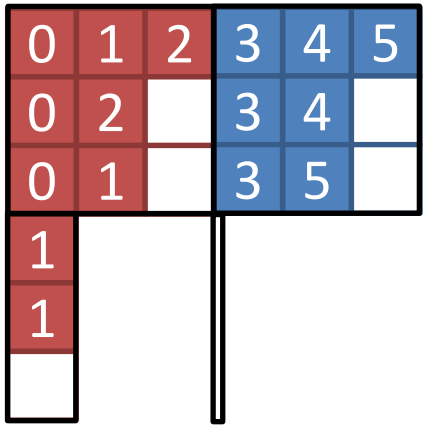
Permutation

AMB (Adaptive Multi-level Blocking)

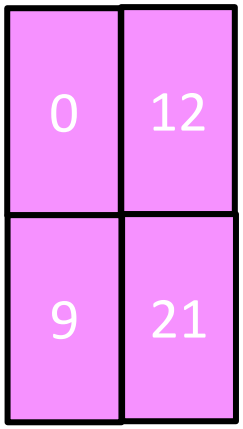
- **Level 2 : Row-wise segmentation and <u>compression</u>**
  - Elimination of the empty chunk
    - Removing needless elements of CS, CL and Permutation



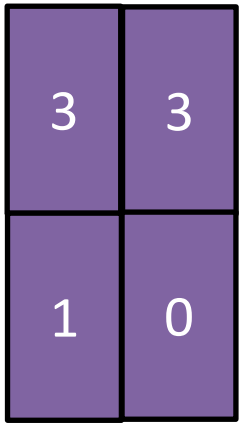
Value data



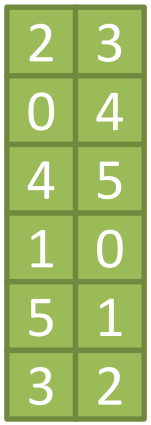
Column Index



CS (Chunk Starting offset)



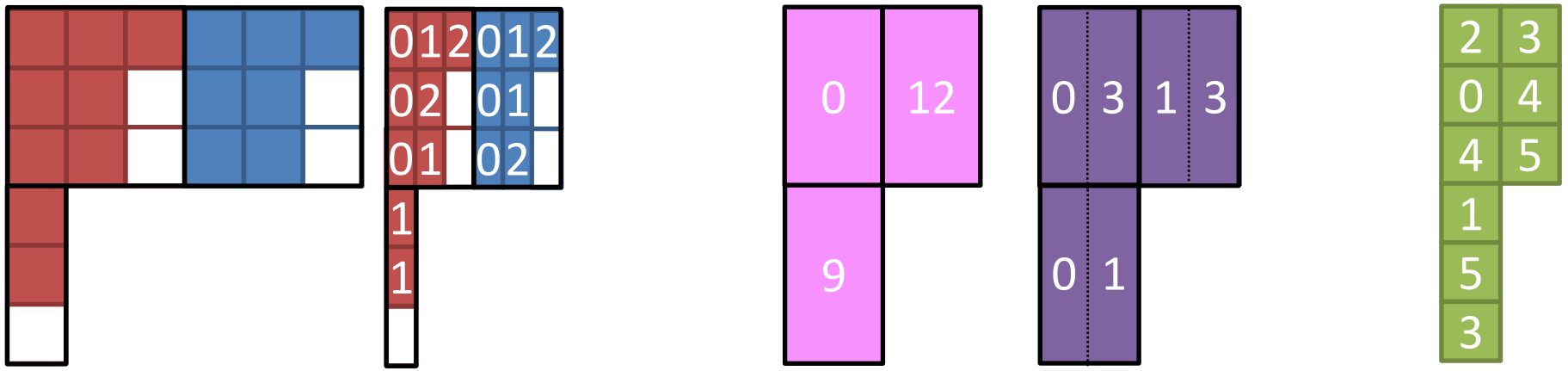
CL (Chunk Length)



Permutation

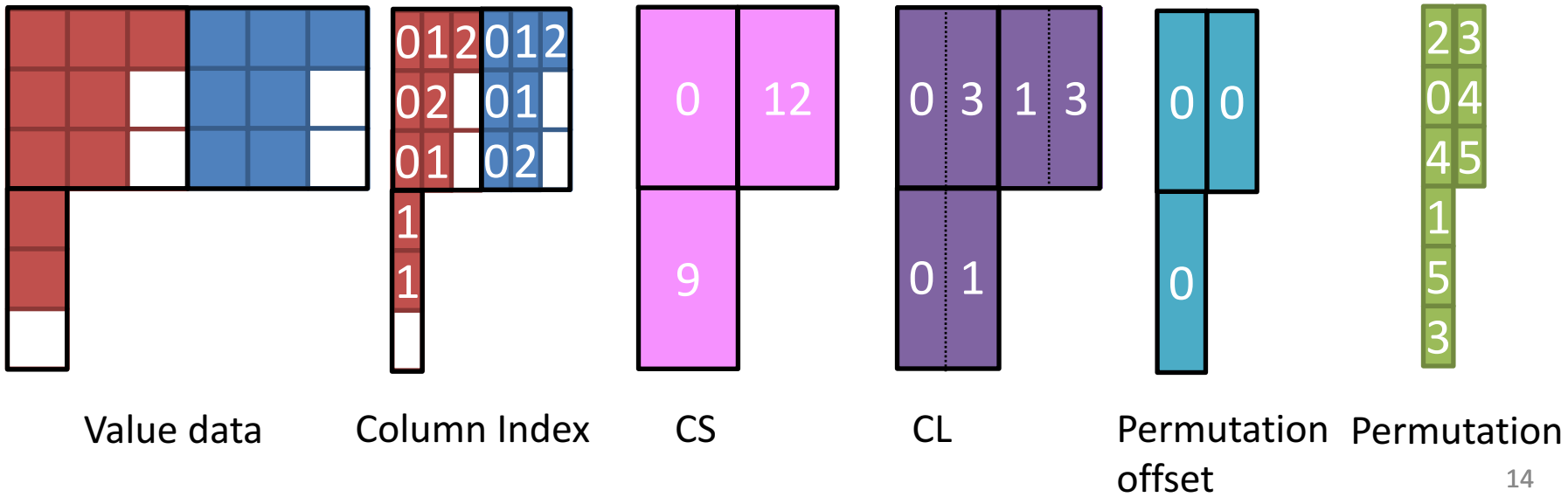
AMB (Adaptive Multi-level Blocking)

- **Level 2 : Row-wise segmentation and <u>compression</u>**
  - Column indices are represented by 16-bit integer (unsigned short)
    - Original index is divided by segmentation width and remainder is stored as column index
      - Quotient is stored to upper 16-bit of the 'CL' (Chunk Length) array



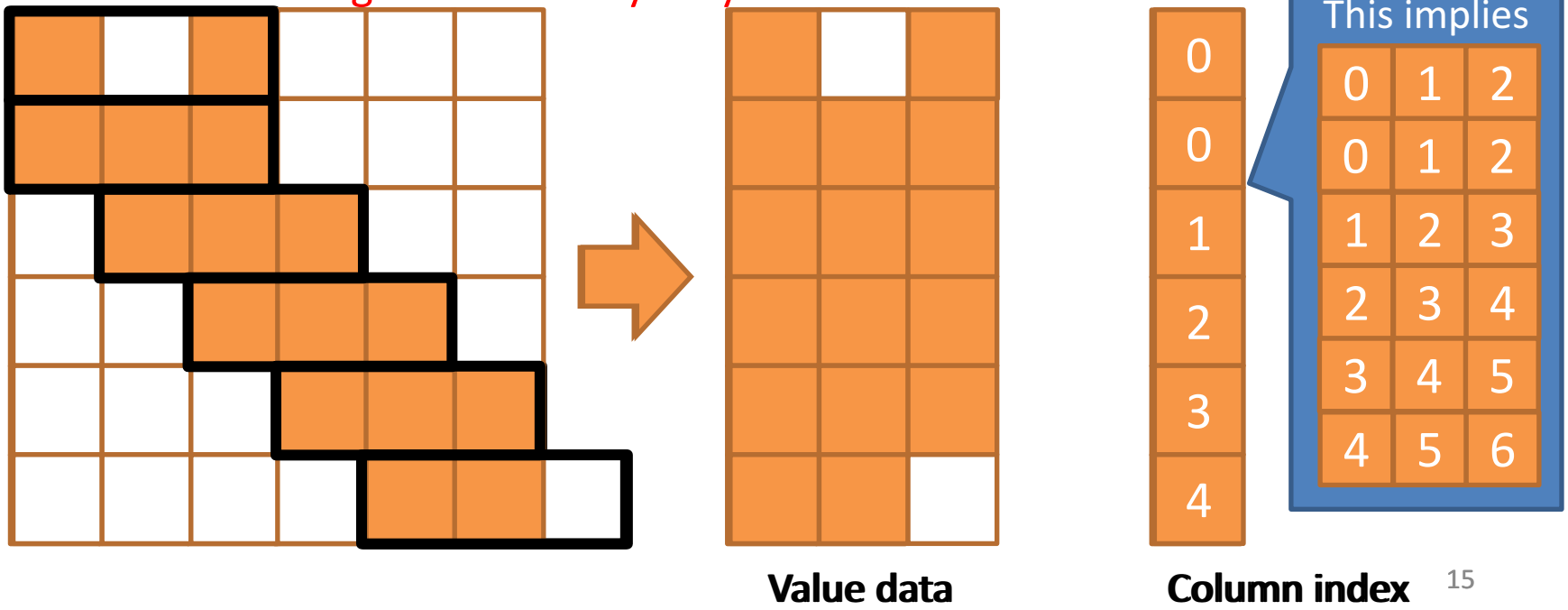
AMB (Adaptive Multi-level Blocking)

- **Level 2 : Row-wise segmentation and <u>compression</u>**
  - Compression of original row index (permutation) by using 16-bit interger ($\sigma \leq 32768 (= 2^{15})$)
    - Original row index is divided by sorting scope (σ=8 in this figure) and remainder is stored as permutation
      - Quotient is stored to Permutation offset array (16-bit integer)



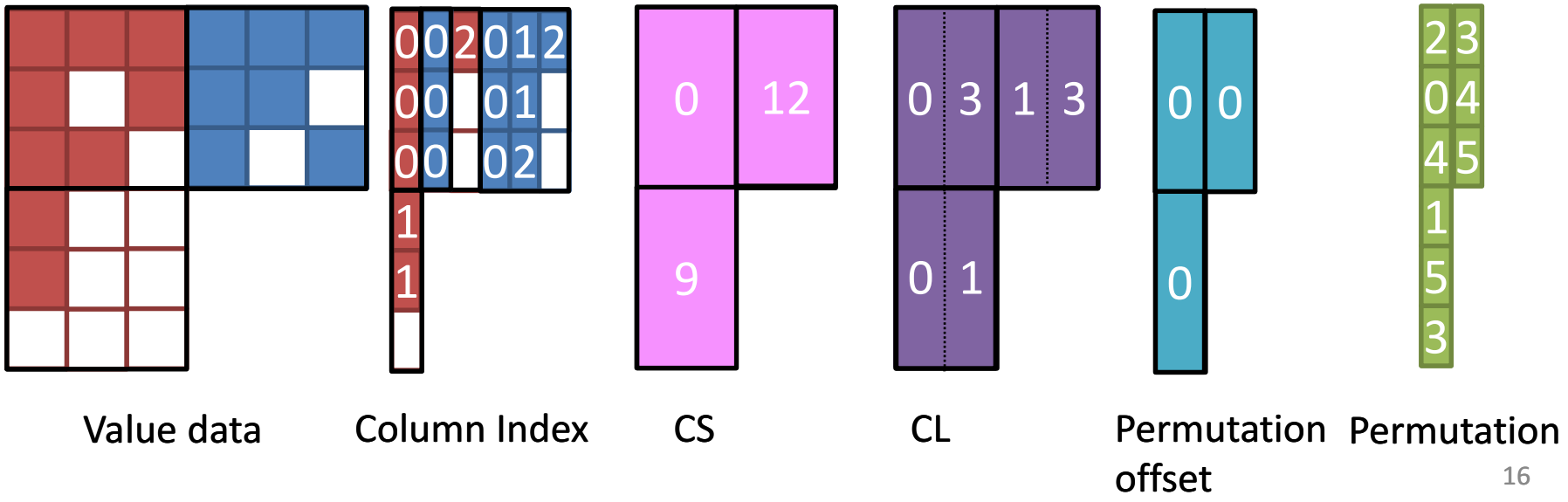
AMB (Adaptive Multi-level Blocking)

- **Level 3 : Blocking**
  - Regarding multiple non-zero elements as one element
    - Compression of contiguous column indices (block size=3 in figure)
    - Large block size : Compression rate is high, but the number of zero-filling in value array may increase



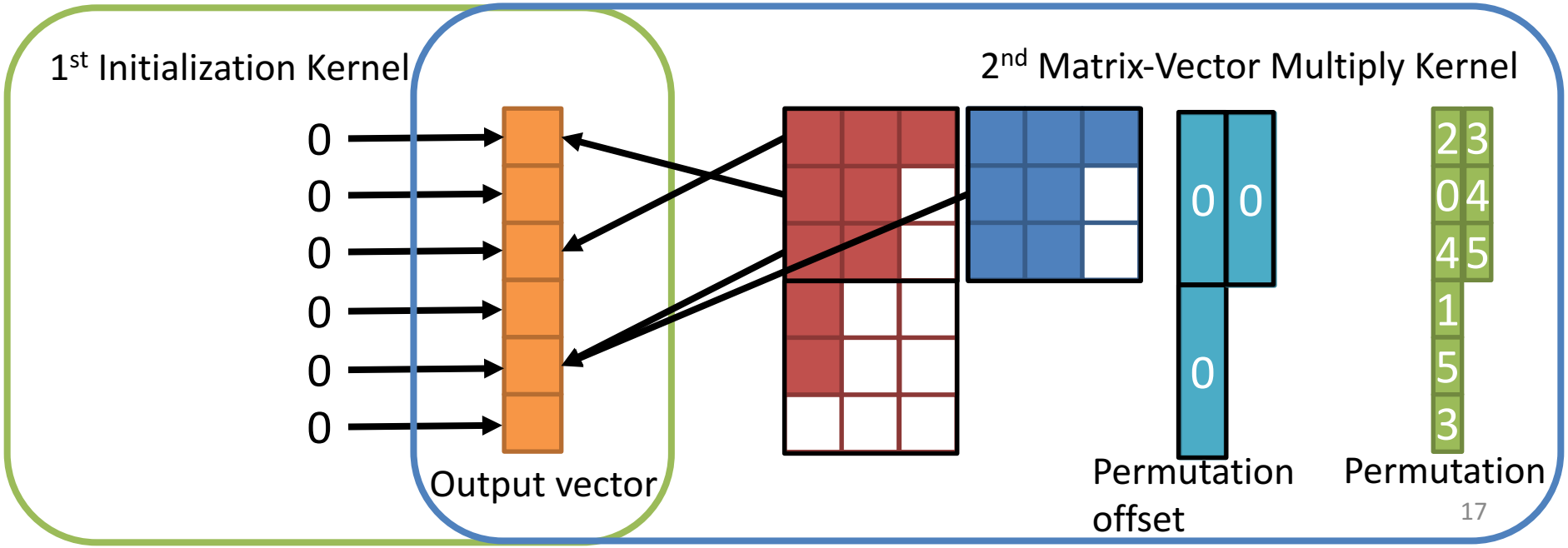
AMB (Adaptive Multi-level Blocking)

- **Level 3 : Blocking**
  - Compression of contiguous column indices
  - Block size : 1~10 (block size=3 in the figure)
    - Block size is chosen considering tradeoff between compression of column index and zero filling in value data



SpMV Kernel for AMB format in CUDA

- Constructed in two CUDA kernels
  - Initializing the output vector with zero
  - Matrix vector multiplication kernel
    - One thread is assigned to each row and the result of each row is accumulated in the output vector by atomic operation



Estimation of Total Memory Access in SpMV

- Performance prediction by estimating total memory access
  - AMB format has high locality in cache level
    - We can estimate total memory access including random access

$$nz_{zf} * \left(v + \frac{2}{b}\right) + n_C * (2 * v * C + 2 * C + 10) + (M + N) * v$$

M, N : Row size, column size
v : Byte of floating point (v=4 in single precision and v=8 in double precision)
nz_{zf} : Number of elements including zero-filling
b : Block size in third level
C : Chunk size in second level
n_c : Number of chunk

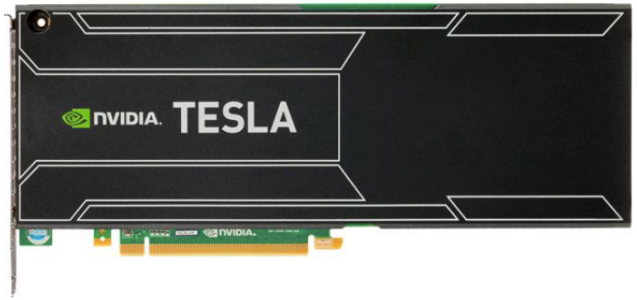
- We can predict the performance without computation

  => Utilizing for parameter tuning

Performance Evaluation

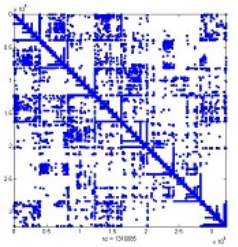
Experiment Environment

- TSUBAME-KFC
  - GPU : NVIDIA Tesla K20X
    - Memory size : 6[GB]
    - Memory bandwidth : 250[GB/s]
    - ECC off
    - L2 cache size : 1.5[MB]
    - Read-only cache size : 12[KB] * 4 / SMX
  - CUDA : Version 7.0



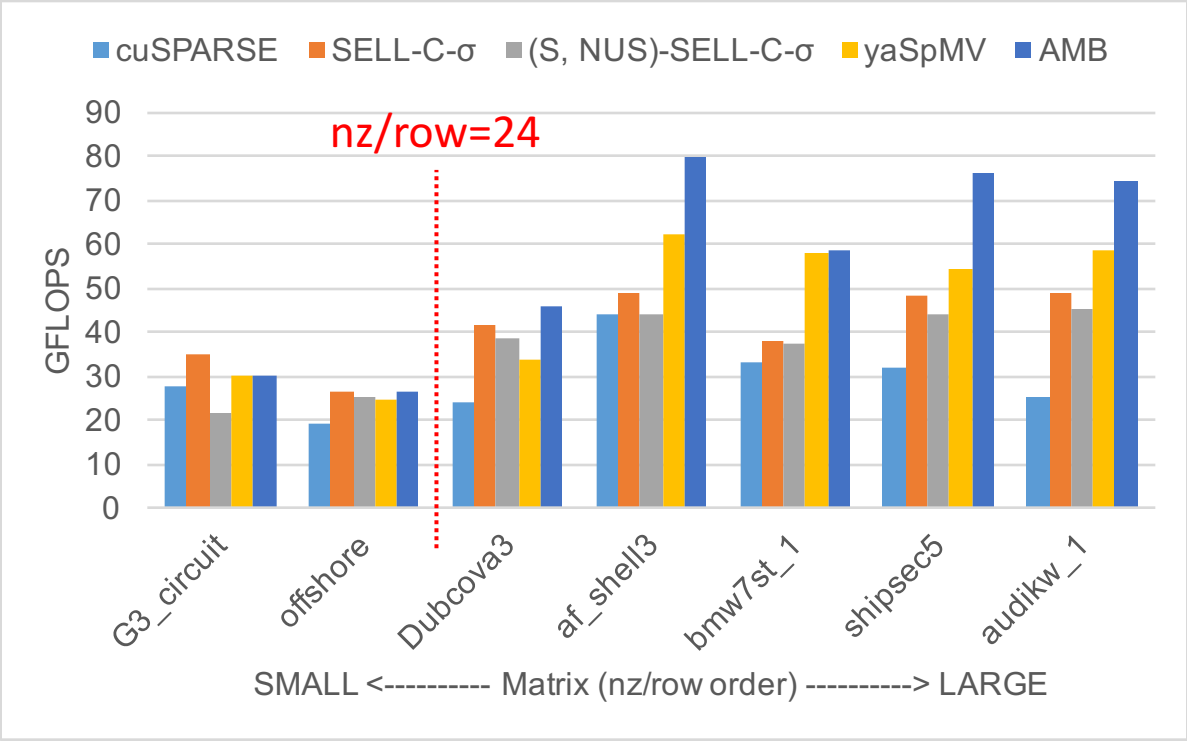
Experiment Environment

- Matrix data
  - The University of Florida Sparse Matrix Collection
    - Selecting 32 positive definite symmetric matrices whose row sizes are larger than 131072 (=2^17)
- Evaluated sparse formats
  - cuSPARSE : CSR, HYBRID, BSR (Block CSR)
  - SELL-C-σ
  - {Segmented, Non-Uniformly-Segmented}-SELL-C-σ
  - yaSpMV: BCCOO（only single precision）



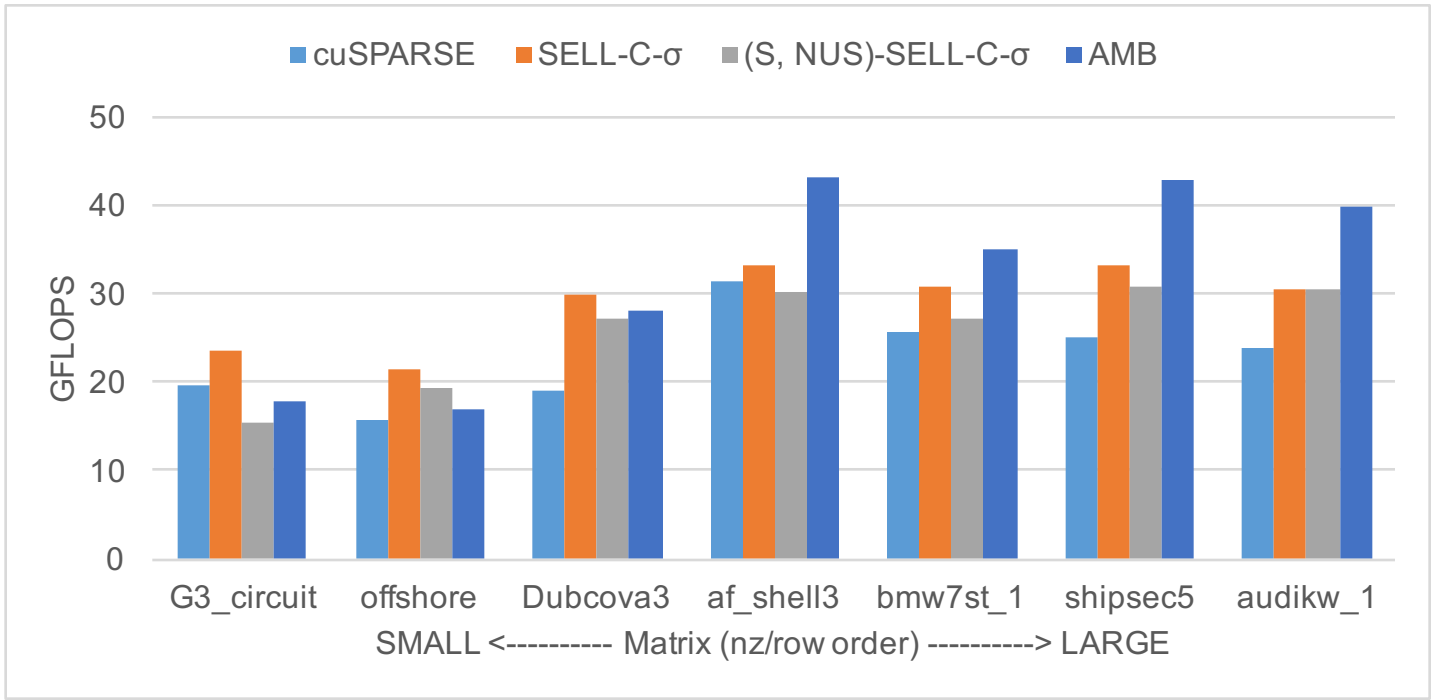
Performance Evaluation
SpMV in Single-precision

- Speedups of
  - x2.92 on maximum and x1.74 on average compared to cuSPARSE
  - x1.40 on maximum and x1.13 on average compared to yaSpMV
    - Performance of AMB becomes worse when average number of non-zero elements per row (nz/row) is small



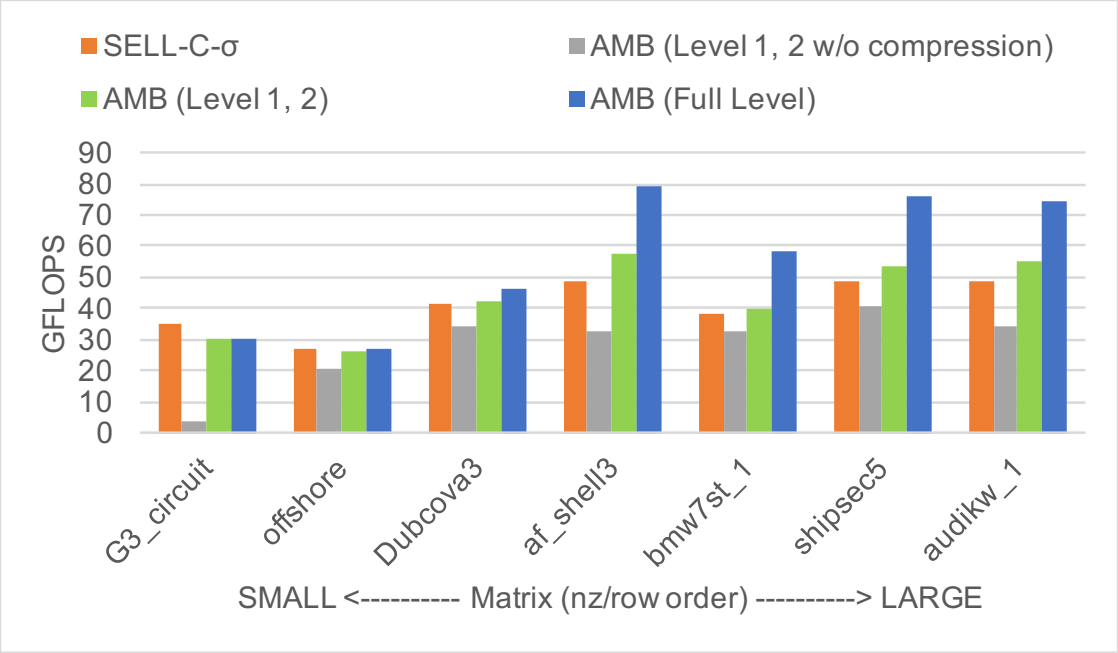
Performance Evaluation
SpMV in Double precision

- Speedups of
  - x1.86 on maximum and x1.29 on average compared to cuSPARSE
  - x2.59 on maximum and x1.83 on average compared to CSR
- Memory access to value data is larger and compression rate becomes worse compared to single precision



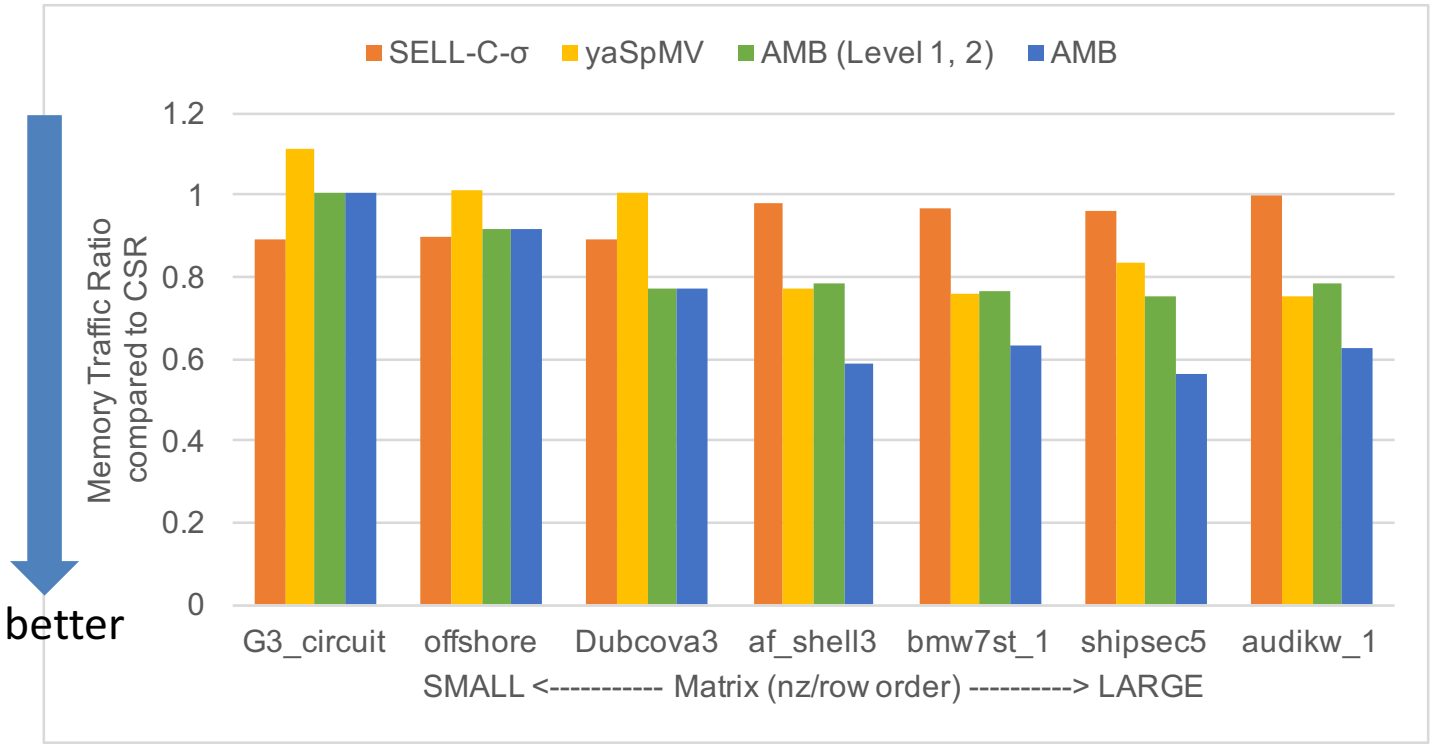
Performance Evaluation SpMV in Single-precision

- Performance of each level of AMB format
  - Memory access to matrix data increases in Level 1 while the locality of the access to input vector is improved
  - Reducing memory access to matrix data in Level 2 and Level 3 contributes the speedups



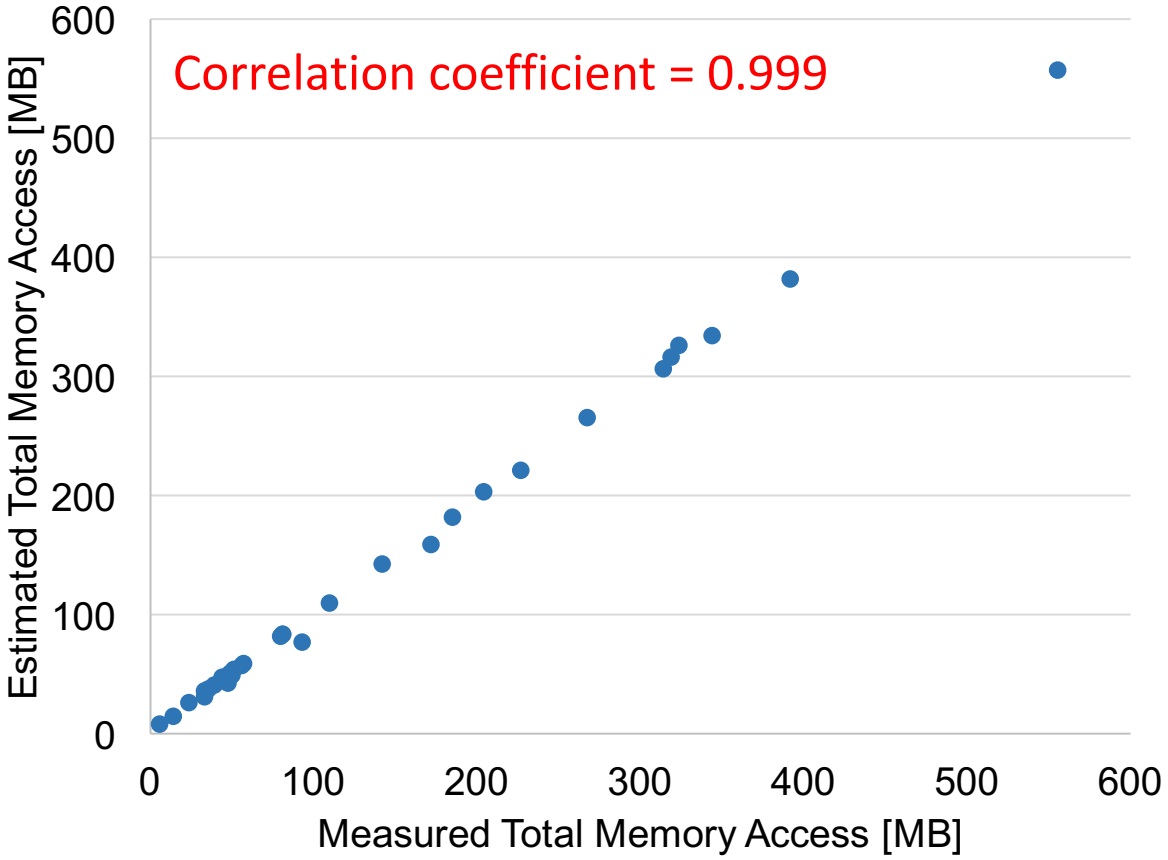
Total Memory Access in SpMV

- Evaluating total memory access in SpMV by using nvprof
  - Format with minimum total memory access shows best performance of SpMV for most of matrix



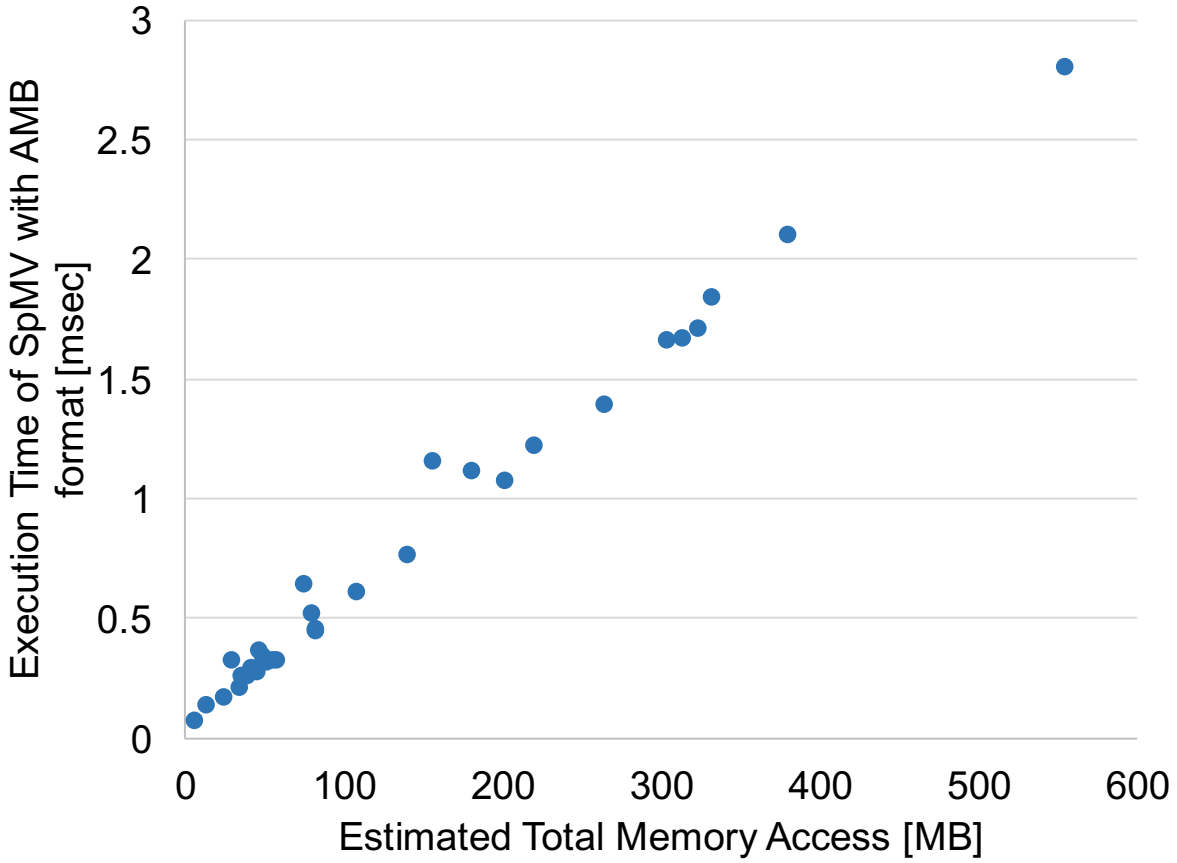
Estimation of Total Memory Access and Performance Prediction

- Very accurate estimation of total memory access in SpMV with AMB format



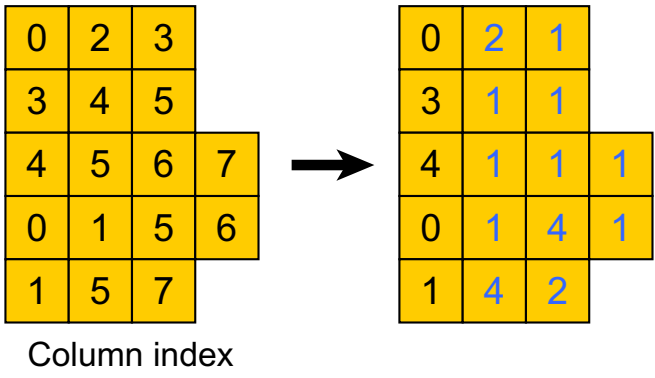
Estimation of Total Memory Access and Performance Prediction

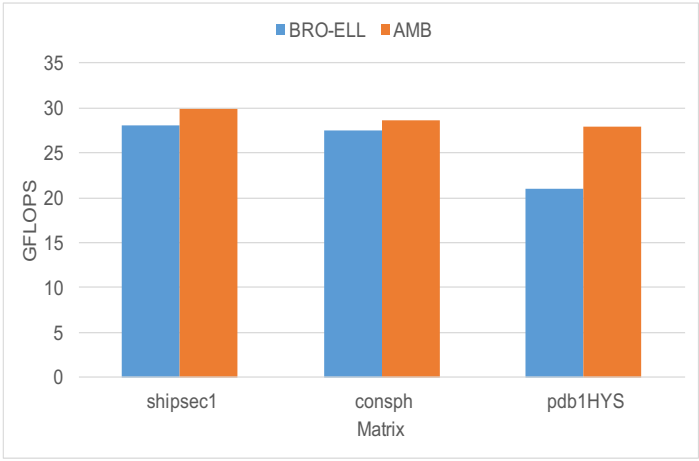
- Performance of SpMV depends on memory access
  - Performance degradation by load-imbalance for some data



Related Work (1)

- CoAdELL [Maggioni, IPDPS2014]
  - Delta between column indices is compressed to 16-bit or 8-bit
  - No compression when delta is large
- BRO format [Tang, SC13]
  - Delta between column indices is represented in more precise number of bits
  - Optimized for GPU by removing sequentially execution
  - Although BRO successfully reduces the memory footprint, it requires additional decompression schemes





Related Work (2)

- BCCOO (yaSpMV) [Yan, PPoPP'14]
  - Storing the matrix data in extended COO with blocking the matrix to reduce the memory usage of the index of column and row
    - Local memory such as shared memory on GPU is effectively utilized when the computation result of each block is accumulated in SpMV

    => Highly accelerating SpMV from existing library such as cuSPARSE
  - Parameter auto-tuning mechanism
    - Find best set of parameters by conversions and executions
    - Large parameter space
      - Cost of parameter tuning is extremely high

Conclusion

- To improve the performance of SpMV on GPU, we propose AMB (Adaptive Multi-level Blocking) format which reduces total memory access
  - Multi-level division and blocking improve the locality of the access to input vector and compresses column index
  - High performance improvement from existing SpMV libraries such as cuSPARSE and yaSpMV
- Future work
  - Investigating the effectiveness of our AMB format on other many-core processors such as AMD GPUs and Xeon Phi

Backup

Zero-filling in Value data

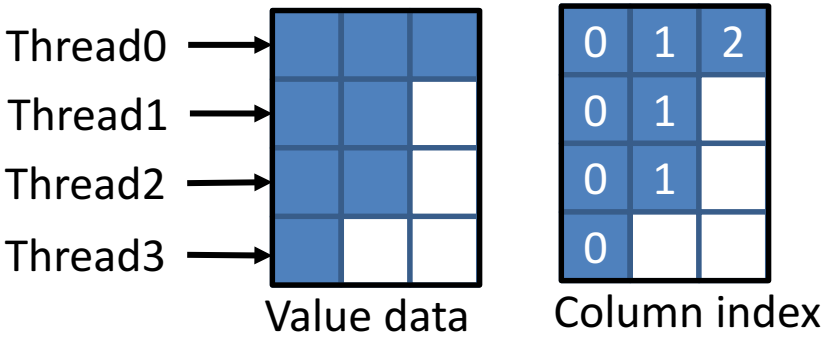
- Zero-filling in value data is not so many
  - Table shows the result of single precision

Matrix	Best block size	#zero-fill/#non-zero[%]	Total memory access of (Best block size) / (block size=1) [%]
af_shell3	5	0.096	75.62
audikw_1	3	0.451	79.77
bmw7st_1	6	4.709	82.22
Dubcova3	2	21.691	99.97
G3_circuit	1	0.070	100.00
offshore	1	0.212	100.00
shipsec5	6	1.731	74.92

Loop unrolling

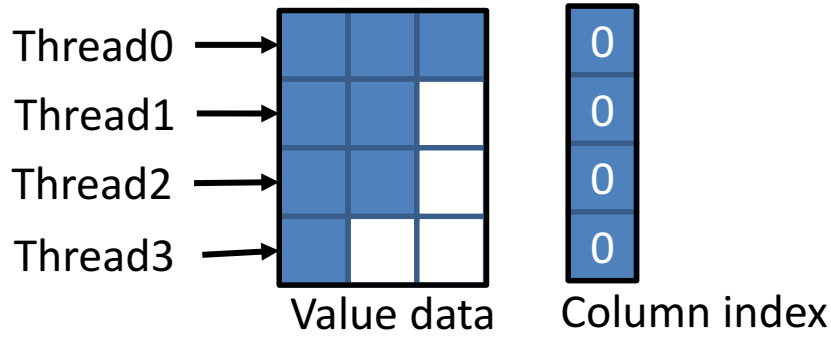
- Loop unrolling technique is used for SpMV
  - Divergence effect is large on GPU

Block size = 1




```
sum = 0;  adr=thread_id;
for (i = 0; i < 3; i++) {
   sum += value[adr] * vec[col[adr]];
   adr += 4;
}
output += sum;
```

Block size = 3




```
sum = 0;  adr = thread_id
for (i = 0; i < 3 / 3; i++) {
   c = col[adr / 3 + adr % 4];
   sum += value[adr] * vec[c];
   sum += value[adr + 4 * 1] * vec[c + 1];
   sum += value[adr + 4 * 2] * vec[c + 2];
   adr += 4;
}
output += sum;
```

Total Memory Access and Execution time

- Exectution time depends on total memory access in SELL-C- σ, yaSpMV and AMB
  - AMB also achieves high throughput although total memory access is less than other formats