H-GCN: A Graph Convolutional Network Accelerator on Versal ACAP Architecture

Chengming Zhang (Washington State University)
Tong Geng (University of Rochester)
Anqi Guo (Boston University)
Jiannan Tian (Washington State University)
Martin Herbordt (Boston University)
Ang Li (Pacific Northwest National Laboratory)
Dingwen Tao (Washington State University)

Background: Graph Convolutional Network (GCN)

➢ **Computation procedure**

- Aggregation and Combination paradigm
- Layer wise forward propagation:

$$X^l = \sigma(\tilde{A} \cdot X^{l-1} \cdot W^l)$$

$$\tilde{A} = D^{-\frac{1}{2}} \cdot \bar{A} \cdot D^{-\frac{1}{2}}$$

$$\bar{A} = A + I$$

D is Laplacian matrix with $D_{ii} \sum_j \bar{A}_{ij}$

- Two-layer GCN model:

$$X^2 = \sigma(\tilde{A} \cdot (\tilde{A} \cdot X^0 \cdot W^1) \cdot W^2)$$

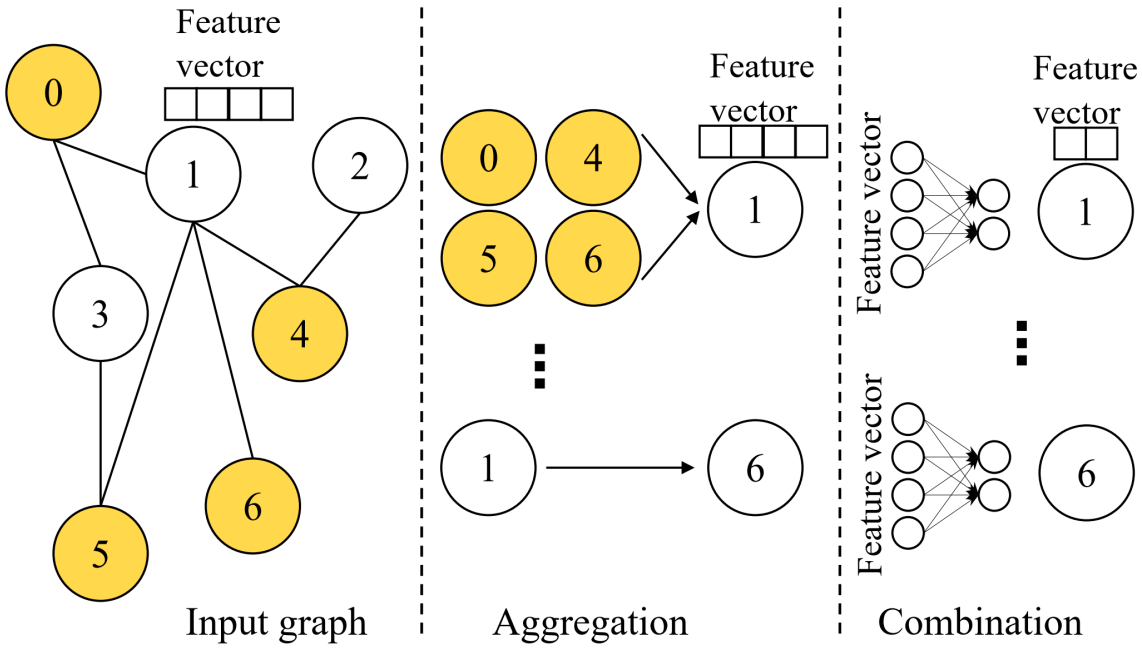


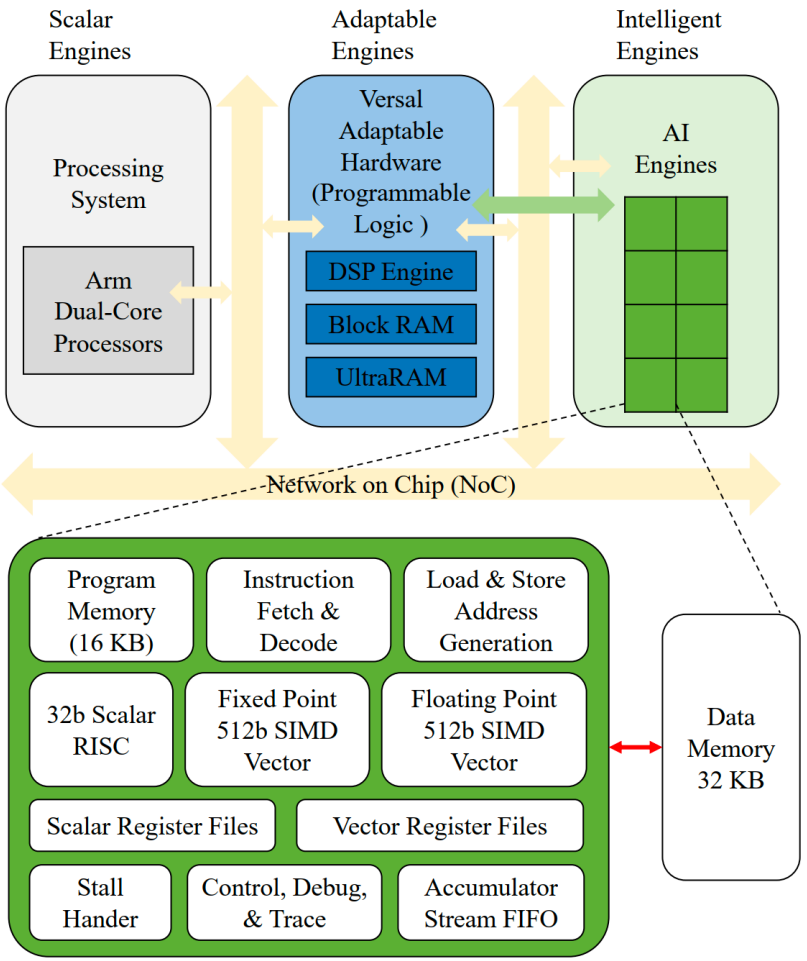
Illustration of typical GCN models

Background: Versal ACAP

- **ACAP Architecture**
- Fully software-programmable, heterogeneous compute platform.
- Heterogeneity:

1. Processor System (PS): Scalar Engines that include the Arm processors.
2. Programmable Logic (PL): Adaptable Engines that include the programmable logic blocks and memory.
3. Artificial intelligence Engine (AIE): with leading-edge memory and interfacing technologies.

- The PL kernels are C/C++ or RTL (traditional FPGA).

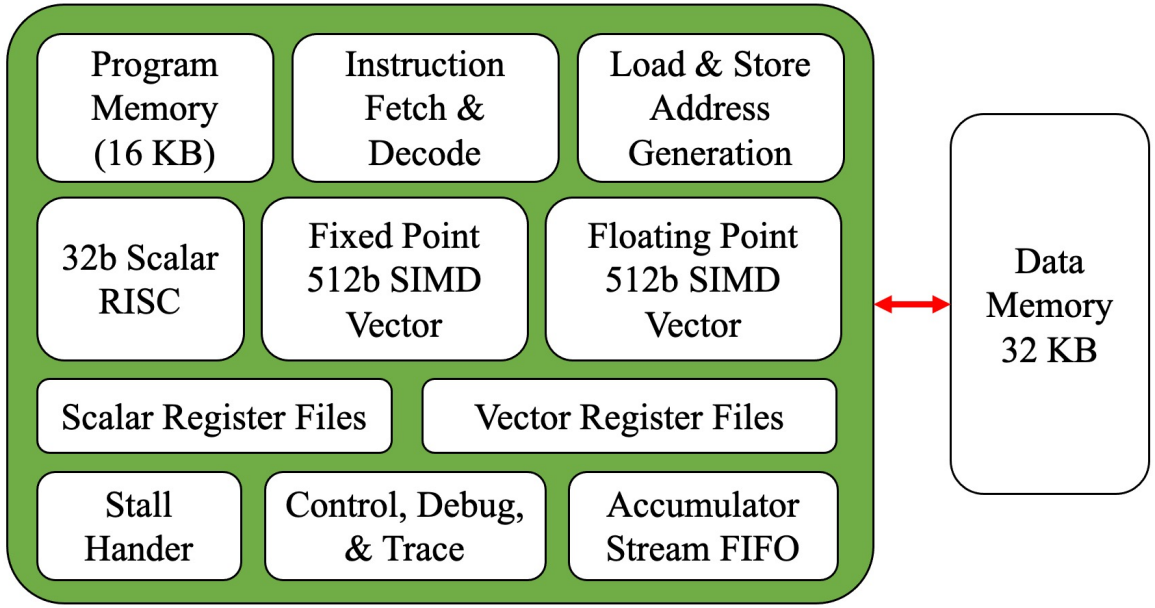


Versal Adaptive Compute Acceleration Platforms (ACAPs).

Background: Versal ACAP

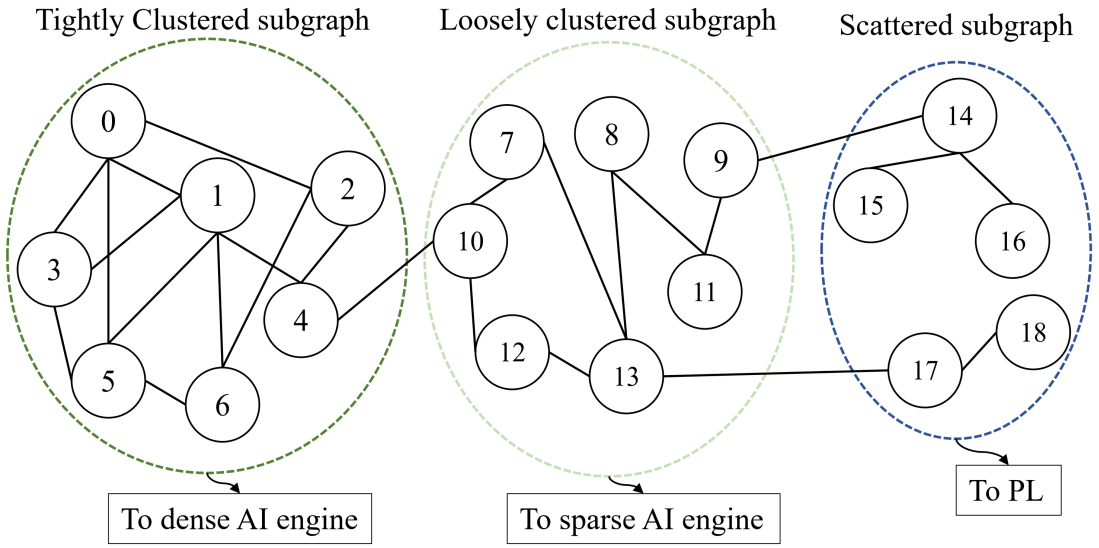
➢ AI Engines

- AI engines are an array of very-long instruction word (VLIW) processors with single instruction multiple data (SIMD) vector units.
- Three levels of parallelism
  (1) SIMD, (2) instruction level, (3) multi-core.
- The AI engine kernels are C/C++ programs written using specialized intrinsic calls or AI engine APIs.



Motivation

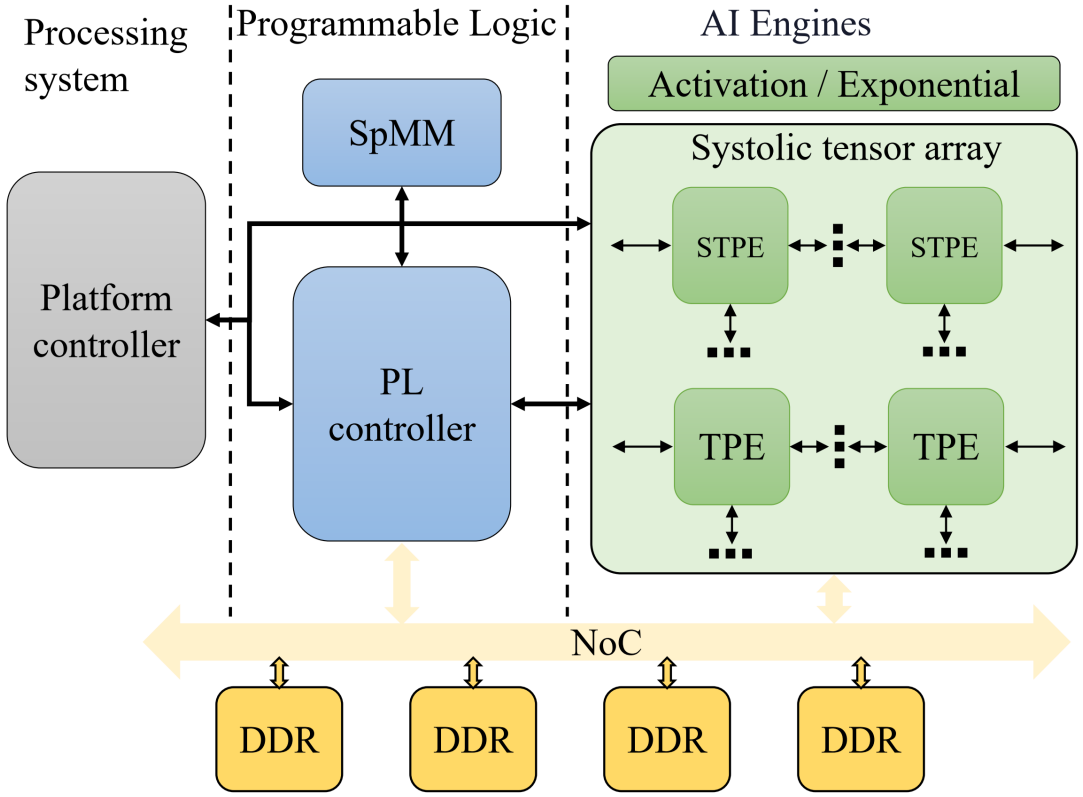
- **Heterogeneity of graph limits performance**
  - A graph has tightly clustered components, loosely clustered components, and scattered nodes.
  - It is NOT possible to use a unified hardware architecture to accelerate all parts of a graph.
- **Performance of FPGA accelerator is bounded**
  - Overall performance of FPGA-based accelerator is bounded by the low frequency.
  - SIMD can provide high frequency and computation power. However, its use scenario is limited (e.g., dense computation) because of fixed computation pattern.



Overview of three types of subgraph.

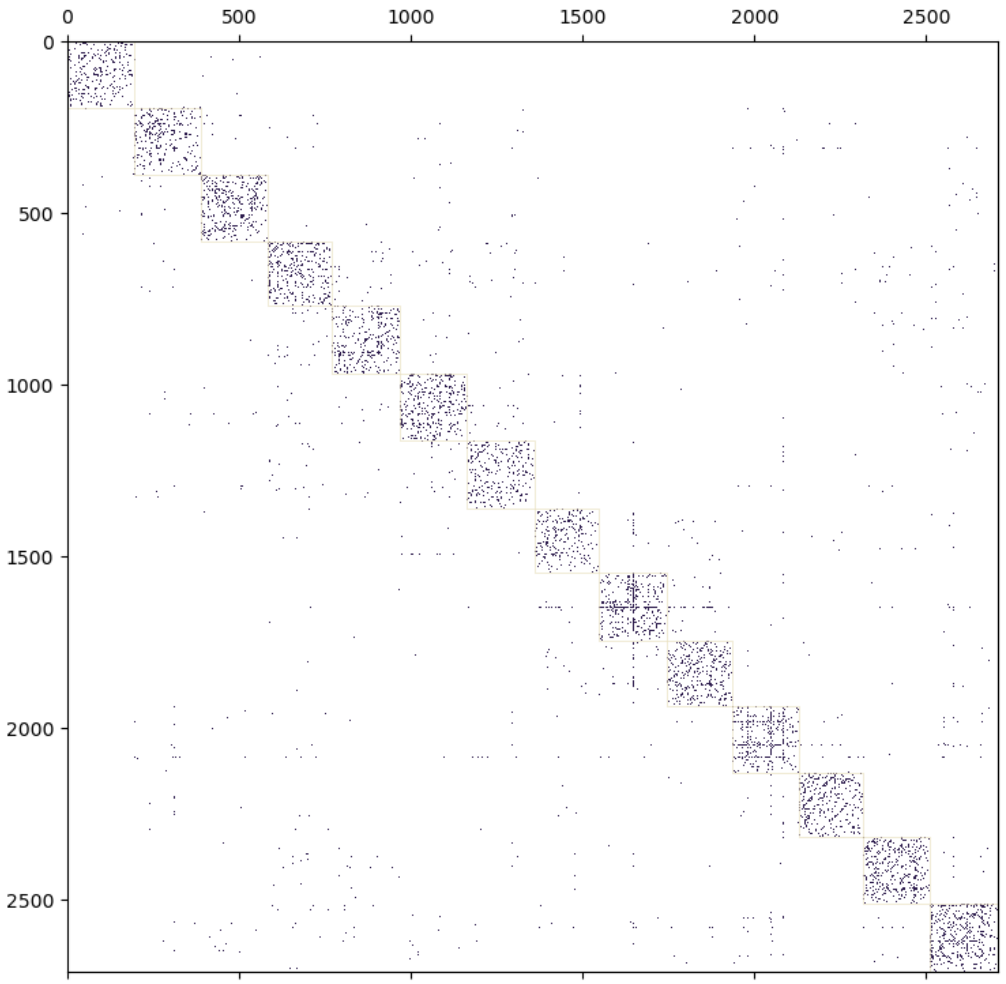
Design: Proposed Architecture

- Consists of a platform controller, a sparse-dense matrix-matrix multiplications (SpMM) unit and a PL controller, a sparse/dense systolic tensor array and activation/exponential unit, a network on chip (NoC), four DDR4 SDRAM.
- Platform controller controls the whole system.
- PL controller controls SpMM unit to cooperate with the sparse/dense systolic tensor arrays to perform all GCN computations.
- Sparse/dense systolic tensor array accelerates of both dense and sparse matrix addition and multiplication.



Design: Input Graph Ordering

- Real-world graphs exhibit a "community" structure, some vertices may share neighbors or have a closer relationship to each other.
- Improve data locality by modifying the order of vertices.
- Perform the graph reordering at the pre-processing stage for only once using mt-metis.



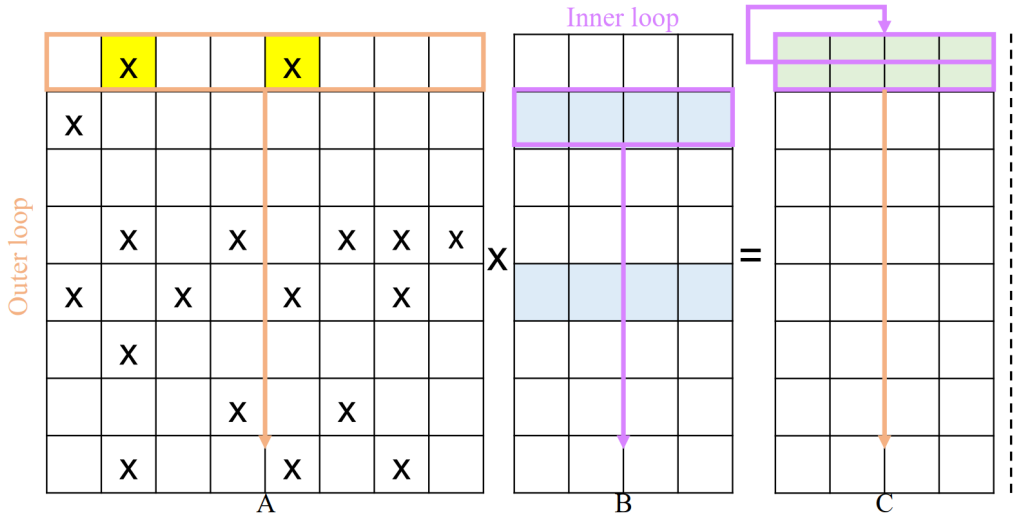
Design: Sparse Tensor Engine

➢ **Matrix multiplication**

- The essence of matrix multiplication is multiply-accumulate (MAC) operations.
- Matrix multiplication can be further decomposed into vector operations.
- AI engines provide a floating-point 512 bits SIMD vector unit, fpmac & fpmul.

➢ **SpMM**

- Row-wise SpMM and compressed sparse row (CSR) increase the generality.




```
ind_ptr[9] = {0, 2, 3, 3, 8,
12, 13, 15, 18}
// Outer loop
for i = 0 to 8
nnz_row_i = ind_ptr[i+1] - ind_ptr[i]
 // Inner loop
  for j = 0 to nnz_row_i
  val = window_readincr(A)
  buf_mat_b = window_read_v8(B)
  acc0 = fpmac(acc0, val,  buf_mat_b)
```

Design: Sparse Tensor Engine

➢ Limitations of SpMM

1. Compiler cannot optimize

- The number of innermost loops is not fixed since the number of non-zeros in each row is not fixed.
- The compiler cannot use pipeline or loop flatten to optimize such loops with a variable number of loops.
- Performance being worse than the dense matrix multiplication with the same size.

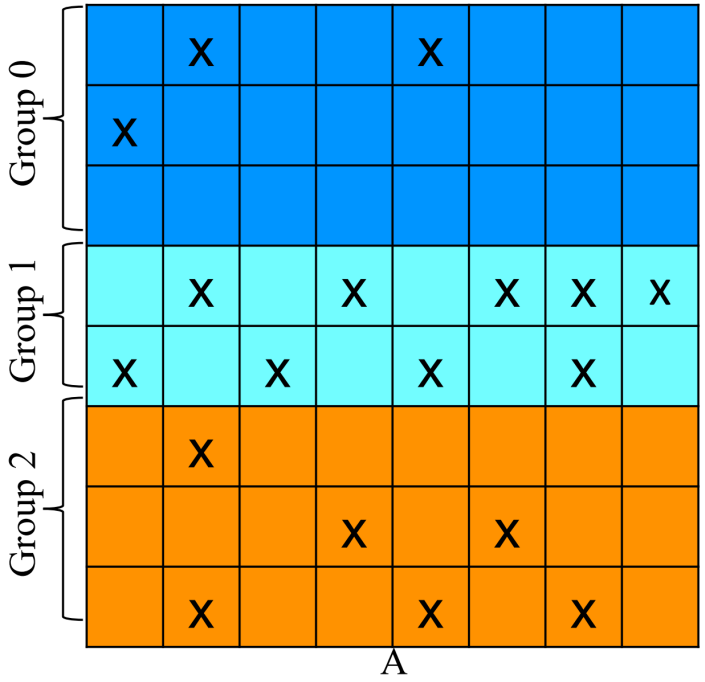
2. Low memory bandwidth utilization: CSR format leads to random row data accesses.

Design: Sparse Tensor Engine

➢ Group SpMM

- Directly flatten outermost loop possibly solves limitations.
- Direct expansion causes insufficient programming space error due to limited programming space.
- "moving average" divides the rows of matrix A into multiple groups.

- **Goals**
  1. Each group contains as many rows as possible to save programming space.
  2. Each group has as little padding as possible to reduce unnecessary calculations on zeros.




```
// Group 0
// Outer loop
for i = 0 to 2
// Inner loop
  for j = 0 to 1
    . . .
// Group 1
for i = 0 to 1
  for j = 0 to 4
    . . .
// Group 2
for i = 0 to 2
  for j = 0 to 2
    . . .
```

Design: Sparse Tensor Engine

- **Grouping algorithm**

- *Lines 8-9*: use *pre_ave* to record the previous moving average, and *cur_ave* saves the current moving average.

- *Lines 13-18*: if the change of the moving average exceeds threshold τ, row j to row i-1 into a group, pad each row in this group to ensure the same number of non-zero elements in each row.

Algorithm 1: Proposed grouping algorithm.

Inputs : A: input array; $nnzs\_rows$: non-zeros of each row; $rows$: the number of rows of A; τ: threshold of changing group

Outputs: $group\_dic$: dictionary of group information; $density$: density after padding

```
1  moving_ave ← MovingAverage(); group_dic ← dict();
     idx_g ← 0
2  for i ← 0 to rows − 1 do
3      if not exist(nnzs_rows) then
4          nnz_row_i ← count_nonzero(A[i, :])
5      else
6          nnz_row_i ← nnzs_rows[i]
7      end
8      pre_ave ← cur_ave
9      cur_ave ← moving_ave.update(nnz_row_i)
10     if pre_ave == 0 then
11         pre_ave ← cur_ave   # Prevent division by 0.
12     end
13     if abs(cur_ave − pre_ave)/pre_ave ≥ τ then
14         group_dic[idx_g] ← g; g ← [ ]    # update group.
15         moving_ave.reset(); moving_ave.update(nnz_row_i)
16     else
17         g.append(i)
18     end
19 end
20 widensity ← calc_density(group_dic)
```

Design: Sparse Systolic Tensor Array

- Two-dimensional (2D) systolic arrays is a pipelined 2D array of processing elements (PEs).
- Efficient local data movement and energy-efficient execution.
- Systolic tensor array with tensor PEs (TPEs).

- Difficulty of performing SpMM using systolic tensor array.

1. TPEs in the same row are required to perform exactly the same calculation mode (e.g., MAC).
2. But for SpMM, each tile has a completely different number of non-zero element and computational model.

Design: Sparse Systolic Tensor Array

➢ **Proposed automatic tensor PEs generation algorithm.**

- *Lines 6-8*: count the non-zeros of tiles in the same row.
- *Lines 9-10*: calculate the average non-zeros (ave_nnz) and maximum non-zeros (max_nnz) of all tiles in the same row.
- *Lines 12-14*: find a suitable number of non-zeros for all tiles in the same row if the difference between ave_nnz and max_nnz is larger than the pre-defined ratio δ; if cannot find a suitable number, select max_nnz as ideal non-zeros for all tiles in the same row.

Algorithm 2: Proposed automatic tensor PEs generation algorithm.

Inputs : A: input sparse matrix; $rows$: the number of rows of A; $cols$: the number of columns of A; $tile\_size$: tile size; δ: ratio by which the number of non-zeros changes. p: coverage percentage; d: density threshold of generating dense tensor PE.

Outputs: Sparse or dense code for systolic tensor PEs in the same row.

```
1  tiles_row = rows/tile_size; tiles_col = cols/tile_size
2  for i ← 0 to tiles_row do
3      nnzs_rows ← [0] × tile_size
4      for j ← 0 to tile_size do
5          nnzs_row_j ← [0] × tiles_col
6          for k ← 0 to tiles_col do
7              nnzs_row_j[k] ←
                 count_nonzero(A[i × tile_size + j, k × tile_size :
                 (k + 1) × tile_size])
8          end
9          ave_nnz ← sum(nnzs_row_j)/len(nnzs_row_j)
10         max_nnz ← max(nnzs_row_j)
11         if max_nnz/ave_nnz ≥ δ then
12             nnzs_rows[j] ← find_nnz(nnzs_row, p)
13         else
14             nnzs_rows[j] ← max_nnz
15         end
16     end
17     group_dic, density ← grouping(nnzs_rows)
18     if density ≥ d then
19         gen_dense_tensor_PE(i)
20     else
21         gen_sparse_tensor_PE(i, group_dic)
22     end
23 end
```

Design: Sparse Systolic Tensor Array

➢ **Proposed automatic tensor PEs generation algorithm.**

- *Line 17*: use the grouping algorithm again to group the rows (enable efficient SpMM on each AIE) after generating the number of non-zeros in each row, and obtain the final density after padding.

- *Lines 18-22*: directly use dense tensor PE for those tiles if their final density is larger than d; otherwise, we use sparse tensor PE.

Algorithm 2: Proposed automatic tensor PEs generation algorithm.

Inputs : A: input sparse matrix; $rows$: the number of rows of A; $cols$: the number of columns of A; $tile\_size$: tile size; δ: ratio by which the number of non-zeros changes. p: coverage percentage; d: density threshold of generating dense tensor PE.

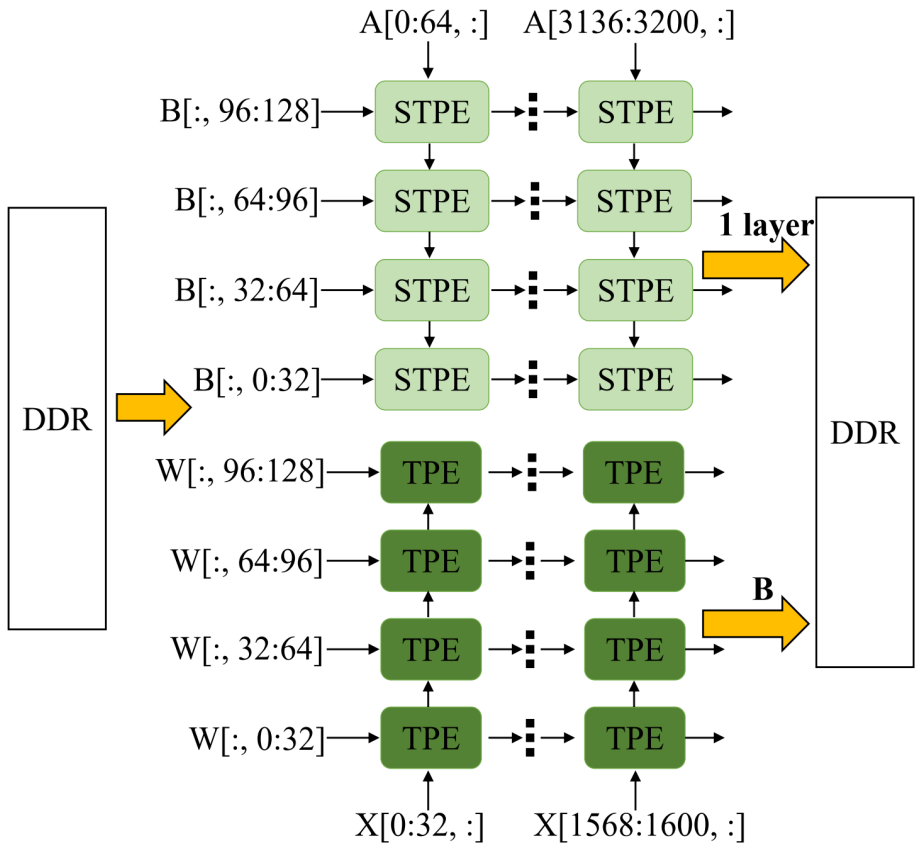
Outputs: Sparse or dense code for systolic tensor PEs in the same row.

```
1  tiles_row = rows/tile_size; tiles_col = cols/tile_size
2  for i ← 0 to tiles_row do
3      nnzs_rows ← [0] × tile_size
4      for j ← 0 to tile_size do
5          nnzs_row_j ← [0] × tiles_col
6          for k ← 0 to tiles_col do
7              nnzs_row_j[k] ←
                 count_nonzero(A[i × tile_size + j, k × tile_size :
                 (k + 1) × tile_size])
8          end
9          ave_nnz ← sum(nnzs_row_j)/len(nnzs_row_j)
10         max_nnz ← max(nnzs_row_j)
11         if max_nnz/ave_nnz ≥ δ then
12             nnzs_rows[j] ← find_nnz(nnzs_row, p)
13         else
14             nnzs_rows[j] ← max_nnz
15         end
16     end
17     group_dic, density ← grouping(nnzs_rows)
18     if density ≥ d then
19         gen_dense_tensor_PE(i)
20     else
21         gen_sparse_tensor_PE(i, group_dic)
22     end
23 end
```

Design: Sparse Systolic Tensor Array

➢ **Pipelining SpMM Chains**

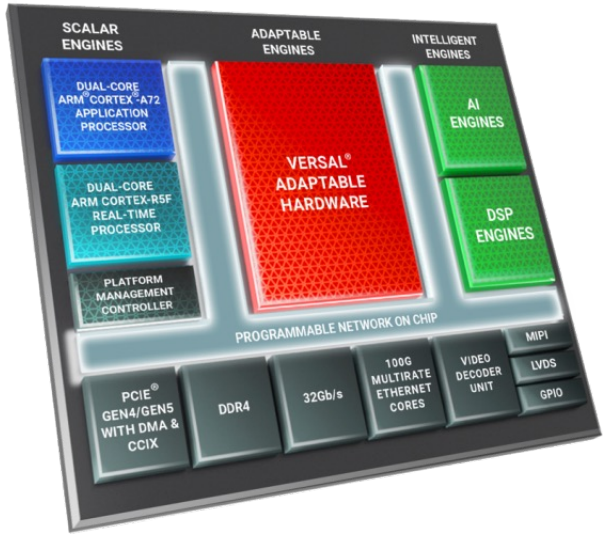
- SpMM chains $A \cdot (X \cdot W)$ are executed on three different hardware, i.e., dense systolic tensor array, sparse systolic tensor array, and PL for SpMM.
- 400 AIEs distributed in 8 rows and 50 columns.
- The upper 4 lines are for mixed sparse or dense systolic tensor PEs (STPEs/TPEs) to perform the computation of $A \cdot B$, where B is the intermediate variable generated by $X \cdot W$.
- The remaining 4 lines are for dense systolic tensor PEs to perform the computation of $X \cdot W$.



Experimental Setup

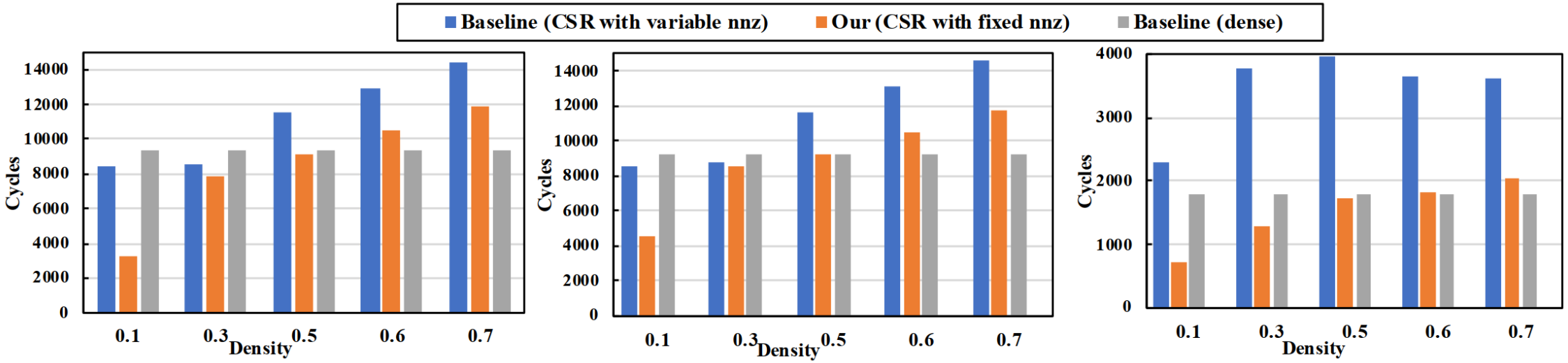
- **Dataset**: 7 real-world datasets
- **GCN Model**: 2-layer Vanilla-GCN model with the hidden dimension of 128.
- **Platform**: Versal VCK5000 board which features with Versal ACAP XCVC1902 device, and four DDR4 with 72-bit memory interface.

Dataset	# Vertices	A's Density	# Features
Cora	2,708	0.14%	1,433
Flickr	89,250	0.011%	500
Citeseer	3,327	0.08%	3,703
Reddit	232,965	0.04%	602
Pubmed	19,717	0.023%	500
Yelp	716,847	0.0027%	300
Amazon	1,569,960	0.011%	200



Speedup of Sparse Tensor Engine

- Our grouping algorithm "CSR-fixed-nnz" provides 2.9x, 2.1x, and 2.5x speedup on matrices of size 64, 32, and 16 (density = 0.1).
- Row-wise SpMM with variable loops "CSR-variable-nnz" is much slower than dense method.



(a) Matrix size: 64 x 64 64 x 8

(b) Matrix size: 32 x 32 32 x 32

(c) Matrix size: 16 x 16 16 x 16

Comparison with SOTA

➢ **Comparison with other GCN accelerators**

- Inference latency
  - Outperforms I-GCN by 1.1x, BoostGCN by 1.5x~2.3x, AWB-GCN by 1.2x, and HyGCN by 6.9x.
  - Due to (1) better data locality, (2) full use of AIEs, and (3) our proposed scheduling approach.

- Energy-efficient
  - 1.12x and 1.64x more energy-efficient than I-GCN and AWB-GCN.
  - Due to the ACAP's more efficient dynamic power management.

COMPARISON OF INFERENCE TIMES (T) IN μS AND ENERGY EFFICIENCY (E) IN GRAPHS/KJ. OOM IS SHORT FOR "OUT OF MEMORY".

Dataset	PyG-CPU		DGL-CPU		PyG-GPU		DGL-GPU		HyGCN		AWB-GCN		I-GCN		BoostGCN		H-GCN (our work)	
	T	E	T	E	T	E	T	E	T	E	T	E	T	E	T	E	T	E
Flickr	3.5E5	17.37	2.4E5	25.43	1.6E4	3.51E2	1.1E4	5.1E2	N/A	N/A	N/A	N/A	N/A	N/A	2.01E4	N/A	1.02E4	1.0E3
Reddit	6.5E6	0.83	5.4E5	11.26	OoM	N/A	6.6E4	87.07	2.89E5	5.17E2	5.0E4	1.5E2	4.6E4	2.2E2	9.81E4	N/A	4.18E4	2.46E2
Yelp	5.9E6	1.03	8.6E5	7.09	OoM	N/A	2.5E5	23.12	N/A	N/A	N/A	N/A	N/A	N/A	1.93E5	N/A	1.2E5	85.85
Amazon	OoM	N/A	2.9E6	2.1	OoM	N/A	OoM	N/A	N/A	N/A	N/A	N/A	N/A	N/A	7.94E5	N/A	5.15E5	19.93E

Comparison with SOTA

➢ **Comparison with CPU/GPU software**

- H-GCN significantly outperforms PyG and DGL on both CPU and GPU
  - Speedup of 79.5x over PyG-CPU
  - Speedup of 12.2x over DGL-CPU
  - Speedup of 1.59x over PyG-GPU
  - Speedup of 1.58x over DGL-GPU

COMPARISON OF INFERENCE TIMES (T) IN μS AND ENERGY EFFICIENCY (E) IN GRAPHS/KJ.

Method	Cora		Citeseer		Pubmed	
	T	E	T	E	T	E
PyG-CPU	1.1E4	5.36E2	1.7E4	3.65E2	5.7E4	1.07E2
DGL-CPU	7.5E3	8.08E2	2.4E4	2.50E2	2.9E4	2.07E2
PyG-GPU	2.2E3	2.55E3	2.7E3	2.16E3	3.7E3	1.53E3
DGL-GPU	4.1E3	1.39E3	4.6E3	1.23E3	4.96E3	1.15E3
H-GCN	1.1E2	9.18E4	2.9E2	3.56E4	1.03E3	9.93E3

Graph Reordering Overhead

- The graph reordering is integrated into the training process.
- The OpenMP version of Metis takes advantage of multiple cores in the CPU (56 CPU cores).

GRAPH REORDERING TIME (mS).

Cora	Citeseer	Pubmed	Flickr	Reddit	Yelp	Amazon
11.5	11.2	33.6	193	648	1650	7310

Thank you!

Any questions and ideas are welcomed

Contact: Dingwen Tao: dingwen.tao@wsu.edu
Chengming Zhang: chengming.zhang@wsu.edu