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:
  \[ X^l = \sigma(\tilde{A} \cdot X^{l-1} \cdot W^l) \]
  \[ \tilde{A} = D^{-\frac{1}{2}} \cdot \tilde{A} \cdot D^{-\frac{1}{2}} \]
  \[ \tilde{A} = A + I \]

  \[ D \text{ is Laplacian matrix with } D_{ii} \sum_j \tilde{A}_{ij} \]

• Two-layer GCN model:
  \[ X^2 = \sigma(\tilde{A} \cdot (\tilde{A} \cdot X^0 \cdot W^1) \cdot W^2) \]
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.
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 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.

```plaintext
ind_ptr[i] = {0, 2, 3, 4, 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_readvec(A)
        buf_mat_b = window_read_v8(B)
        acc0 = fpmac(acc0, val, buf_mat_b)

ind_ptr[i] = {0, 2, 3, 4, 8, 12, 13, 15, 18}
```

![Diagram of matrix multiplication and SpMM](image-url)
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.
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 $\tau$, 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.

```plaintext
Inputs: $A$: input array; $\text{nnzs}_\text{rows}$: non-zeros of each row;
        $\text{rows}$: the number of rows of $A$; $\tau$: threshold of changing group
Outputs: $\text{group}_\text{dic}$: dictionary of group information; $\text{density}$: density after padding
1 $\text{moving}_\text{ave} \leftarrow \text{MovingAverage}(); \text{group}_\text{dic} \leftarrow \text{dict}();$
   $\text{idx}_g \leftarrow 0$
2 for $i \leftarrow 0$ to $\text{rows} - 1$ do
3     if $\text{not exist}($nnzs_rows$)$ then
4         $\text{nnz}_\text{row}_i \leftarrow \text{count}_\text{nonzero}(A[i,:])$
5       else
6         $\text{nnz}_\text{row}_i \leftarrow \text{nnzs}_\text{rows}[i]$
7       end
8     $\text{pre}_\text{ave} \leftarrow \text{cur}_\text{ave}$
9     $\text{cur}_\text{ave} \leftarrow \text{moving}_\text{ave}.\text{update}(\text{nnz}_\text{row}_i)$
10    if $\text{pre}_\text{ave} = 0$ then
11       $\text{pre}_\text{ave} \leftarrow \text{cur}_\text{ave}$  # Prevent division by 0.
12    end
13    if $\text{abs}($cur_ave $-$ pre_ave$)/$pre_ave $\geq \tau$ then
14       $\text{group}_\text{dic}[\text{idx}_g] \leftarrow g; g \leftarrow []$  # update group.
15       $\text{moving}_\text{ave}.\text{reset}(); \text{moving}_\text{ave}.\text{update}(\text{nnz}_\text{row}_i)$
16     else
17       $g.\text{append}(i)$
18     end
19 end
20 widensity $\leftarrow \text{calc}_\text{density}(\text{group}_\text{dic})$
```
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.
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 $\delta$; if cannot find a suitable number, select max_nnz as ideal non-zeros for all tiles in the same row.
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; $\delta$: 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.

```
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.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Vertices</th>
<th>A’s Density</th>
<th># Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cora</td>
<td>2,708</td>
<td>0.14%</td>
<td>1,433</td>
</tr>
<tr>
<td>Flickr</td>
<td>89,250</td>
<td>0.011%</td>
<td>500</td>
</tr>
<tr>
<td>Citeseer</td>
<td>3,327</td>
<td>0.08%</td>
<td>3,703</td>
</tr>
<tr>
<td>Reddit</td>
<td>232,965</td>
<td>0.04%</td>
<td>602</td>
</tr>
<tr>
<td>Pubmed</td>
<td>19,717</td>
<td>0.023%</td>
<td>500</td>
</tr>
<tr>
<td>Yelp</td>
<td>716,847</td>
<td>0.0027%</td>
<td>300</td>
</tr>
<tr>
<td>Amazon</td>
<td>1,569,960</td>
<td>0.011%</td>
<td>200</td>
</tr>
</tbody>
</table>
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.
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”.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Flickr</td>
<td>3.5E5</td>
<td>17.37</td>
<td>2.4E5</td>
<td>25.43</td>
<td>1.6E4</td>
<td>3.51E2</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>2.01E4</td>
<td>N/A</td>
<td>1.02E4</td>
<td>1.0E3</td>
</tr>
<tr>
<td>Reddit</td>
<td>6.5E6</td>
<td>0.83</td>
<td>5.4E5</td>
<td>11.26</td>
<td>6.6E4</td>
<td>87.07</td>
<td>2.89E5</td>
<td>5.17E2</td>
<td>5.0E4</td>
<td>1.5E2</td>
<td>9.81E4</td>
<td>N/A</td>
<td>4.18E4</td>
<td>2.46E2</td>
</tr>
<tr>
<td>Yelp</td>
<td>5.9E6</td>
<td>1.03</td>
<td>8.6E5</td>
<td>7.09</td>
<td>2.5E5</td>
<td>23.12</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>1.93E5</td>
<td>N/A</td>
<td>1.2E5</td>
<td>85.85</td>
</tr>
<tr>
<td>Amazon</td>
<td>OoM</td>
<td>N/A</td>
<td>2.9E6</td>
<td>2.1</td>
<td>OoM</td>
<td>N/A</td>
<td>OoM</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>7.94E5</td>
<td>N/A</td>
<td>5.15E5</td>
<td>19.93E</td>
</tr>
</tbody>
</table>


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

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

<table>
<thead>
<tr>
<th>Graph Reordering Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cora</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>11.5</td>
</tr>
</tbody>
</table>
Thank you!
Any questions and ideas are welcomed

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