

# **NDMiner: Accelerating Graph Pattern Mining Using Near Data Processing**

**Nishil Talati, Haojie Ye, Yichen Yang, Leul Belayneh, Kuan-Yu Chen,  
David Blaauw, Trevor Mudge, and Ronald Dreslinski**

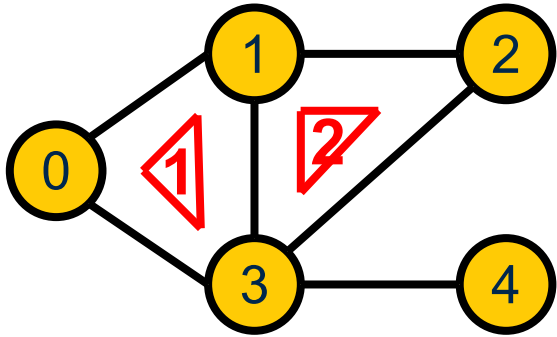
**International Symposium on Computer Architecture (ISCA) 2022**

**Session 2B: Graph Applications**

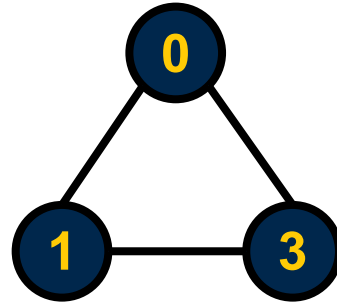


# Graph Pattern Mining (GPM)

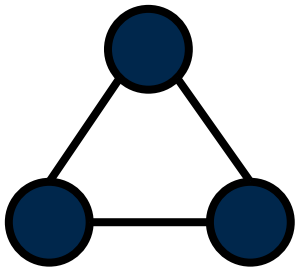
Input Graph



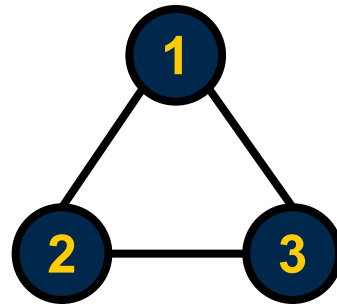
Mined Patterns in Input Graph



Input Pattern

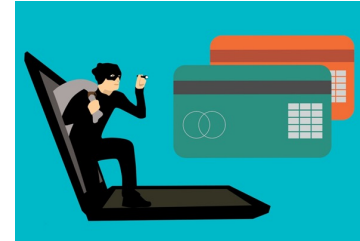


**Goal:** Find the unique instances of input patterns in an input graph



Applications of GPM

Cyber-security



Bioinformatics



Social Network Analysis



Spam Detection



Many more...

# Graph Pattern Mining (GPM)

Input Graph

Mined Patterns in  
Input Graph

Applications of GPM

**Are today's hardware platforms adequate?**

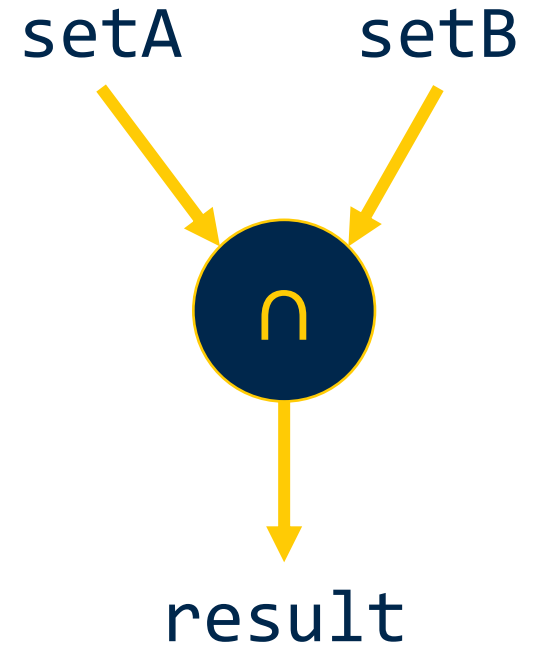


**Goal:** Find the unique instances of input patterns in an input graph

Many more...

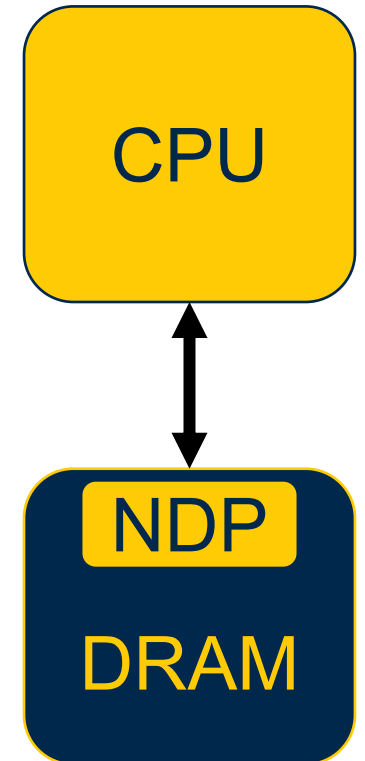
# Performance Bottleneck In GPM

- Set operations dominate execution time
  - Set intersection and set difference
- Control flow and memory instructions
  - Data-dependent branch instructions
  - Memory accesses to irregular graph data structure
- Prior hardware works
  - Design domain-specific accelerator architectures
  - Employ generic Near Data Processing (NDP) architectures



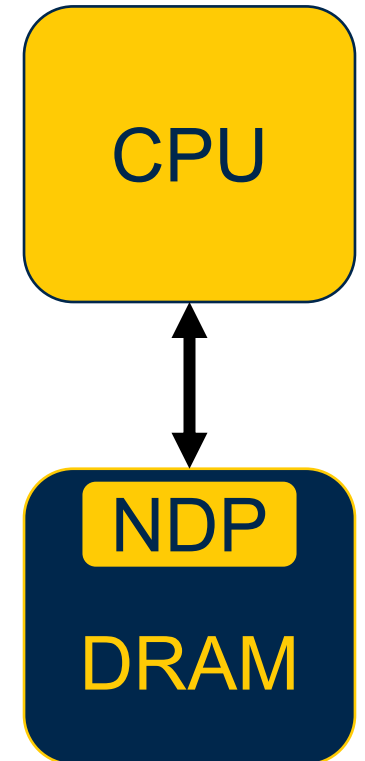
# Contributions of NDMiner

- NDMiner combines domain specialization and NDP
- **Load elision unit**
  - Reduces unnecessary data transfer
- **Loop nest flattening**
  - Improves algorithmic efficiency
- **Set operation reordering**
  - Enhances bank-level parallelism



# In This Talk

- NDMiner combines domain specialization and NDP
- **Load elision unit**
  - Reduces unnecessary data transfer
- ~~Loop nest flattening~~
  - ~~Improves algorithmic efficiency~~
- **Set operation reordering**
  - Enhances bank-level parallelism



# Why NDP For GPM?

Workload Property	Feasibility of NDP

## Triangle Counting

```
for u0 in input_graph:
    N_u0 = G.out_neigh(u0)
    for u1 in N_u0:
        if u1 >= u0: break
        N_u1 = G.out_neigh(u1)
        N_u0u1 = Intersection(N_u0, N_u1)
        for u2 in N_u0u1:
            if u2 >= u1: break
            {u0, u1, u2}: triangle
```

# Why NDP For GPM?

## Workload Property

Wasteful data transfers

## Feasibility of NDP

In-memory data processing

## Triangle Counting

```
for u0 in input_graph:
    N_u0 = G.out_neigh(u0)
    for u1 in N_u0:
        if u1 >= u0: break
        N_u1 = G.out_neigh(u1)
        N_u0u1 = Intersection(N_u0, N_u1)
        for u2 in N_u0u1:
            if u2 >= u1: break
            {u0, u1, u2}: triangle
```

# Why NDP For GPM?

Workload Property	Feasibility of NDP
Wasteful data transfers	In-memory data processing
Employ simple arithmetic ops	Cost-effective logic integration

## Triangle Counting

```
for u0 in input_graph:
    N_u0 = G.out_neigh(u0)
    for u1 in N_u0:
        if u1 >= u0: break
        N_u1 = G.out_neigh(u1)
        N_u0u1 = Intersection(N_u0, N_u1)
        for u2 in N_u0u1:
            if u2 >= u1: break
            {u0, u1, u2}: triangle
```

# Why NDP For GPM?

## Workload Property

## Feasibility of NDP

Wasteful data transfers

In-memory data processing

Employ simple arithmetic ops

Cost-effective logic integration

Read-only data structures

No coherence issue with CPU caches

## Triangle Counting

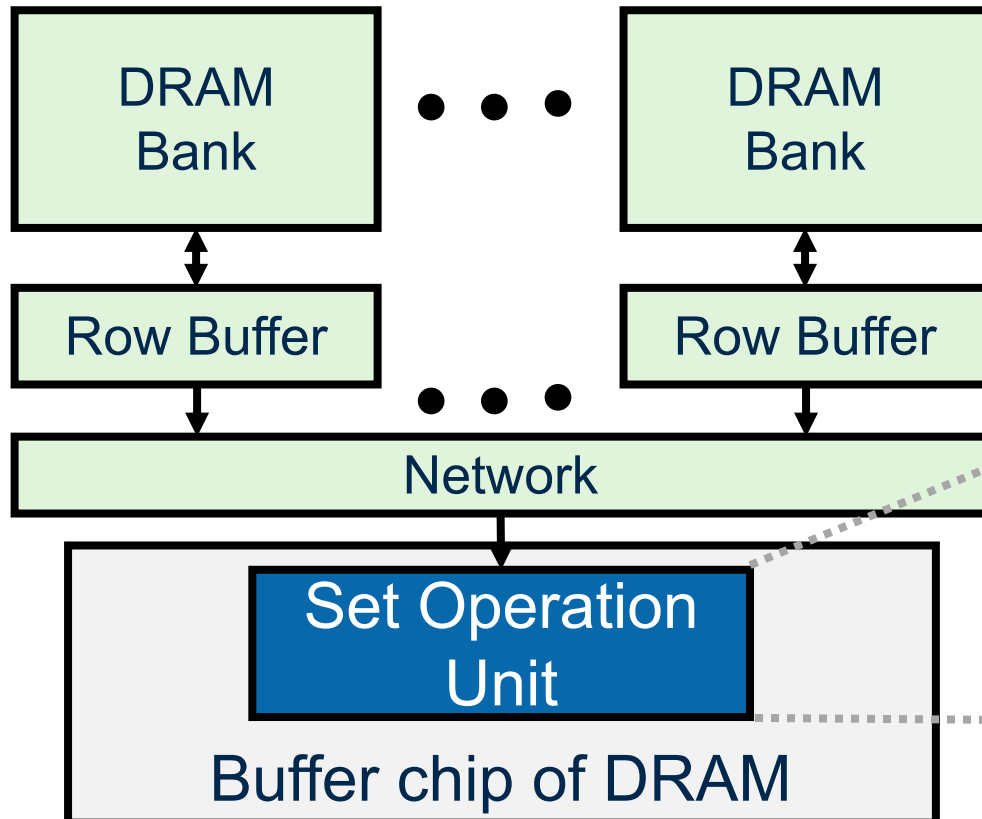
```
for u0 in input graph:
    N_u0 = G.out_neigh(u0)
    for u1 in N_u0:
        if u1 >= u0: break
        N_u1 = G.out_neigh(u1)
        N_u0u1 = Intersection(N_u0, N_u1)
        for u2 in N_u0u1:
            if u2 >= u1: break
            {u0, u1, u2}: triangle
```

NDP is an attractive candidate to accelerate GPM

# Baseline NDP Architecture

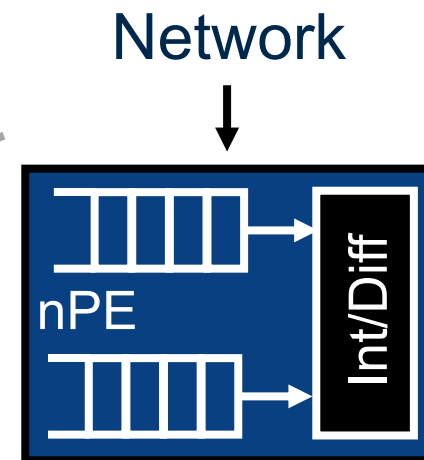
## Type Of Computation

Offload **set operations** to NDP



## Where To Compute

**Outside** memory banks



nPE – near-data  
Processing Element

Set operation unit

Global visibility of  
operands

# Wasteful Data Transfer Due To Symmetry Breaking Constraints

## Example Values

$u_0 = 15, u_1 = 12$

$N_{u_0} = \{5, 7, 11, 19, 22, 40\}$

$N_{u_1} = \{5, 11, 19, 22, 40\}$



Intersection( $N_{u_0}, N_{u_1}$ )  
 $\{5, 11, \text{X}, \text{X}, \text{X}\}$

**60% data wasted**

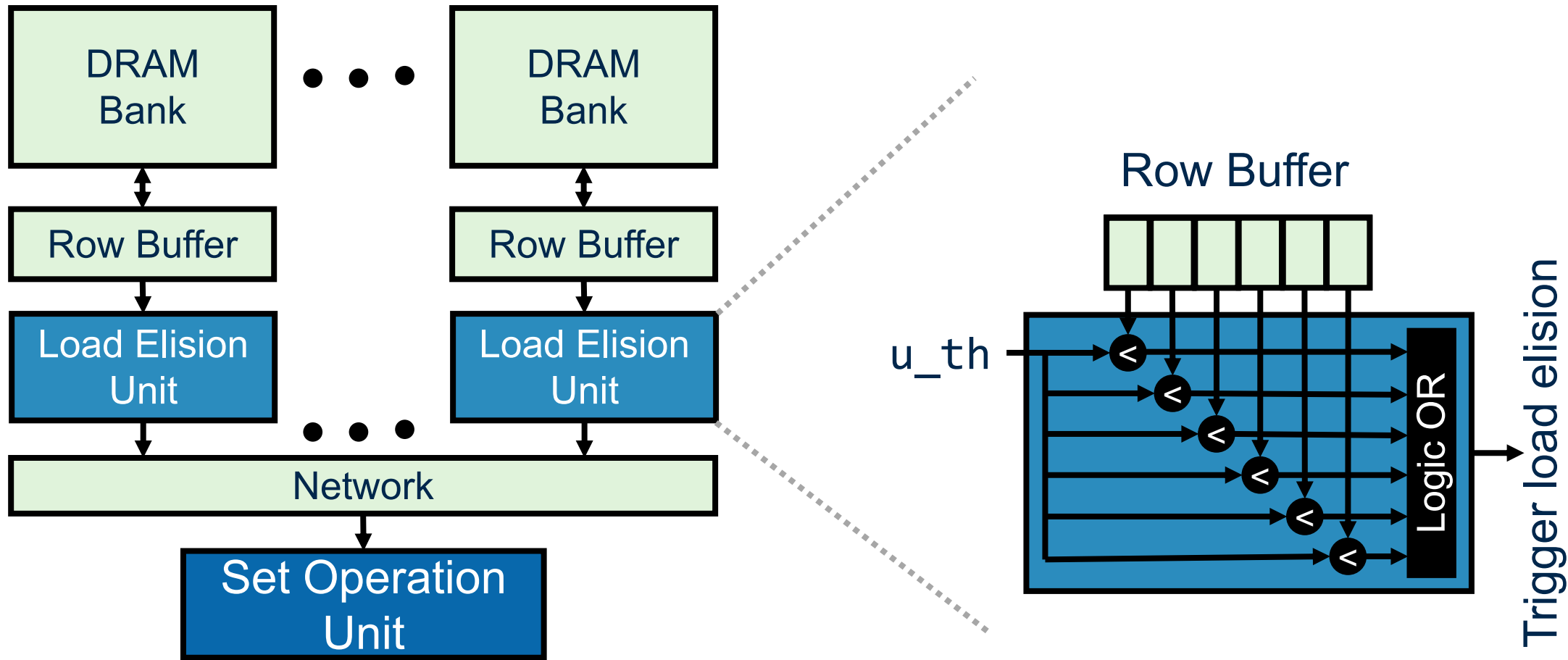
## Typical GPM Algorithm

```
...  
Nu0u1 = Intersection(Nu0, Nu1)  
for u2 in Nu0u1:  
    if u2 >= u1: break  
...
```

**Symmetry Breaking Constraint**  
Ordering on vertex ordering to  
avoid redundant counting

**Insight:** Symmetry breaking constraints results in 66.5% data wastage

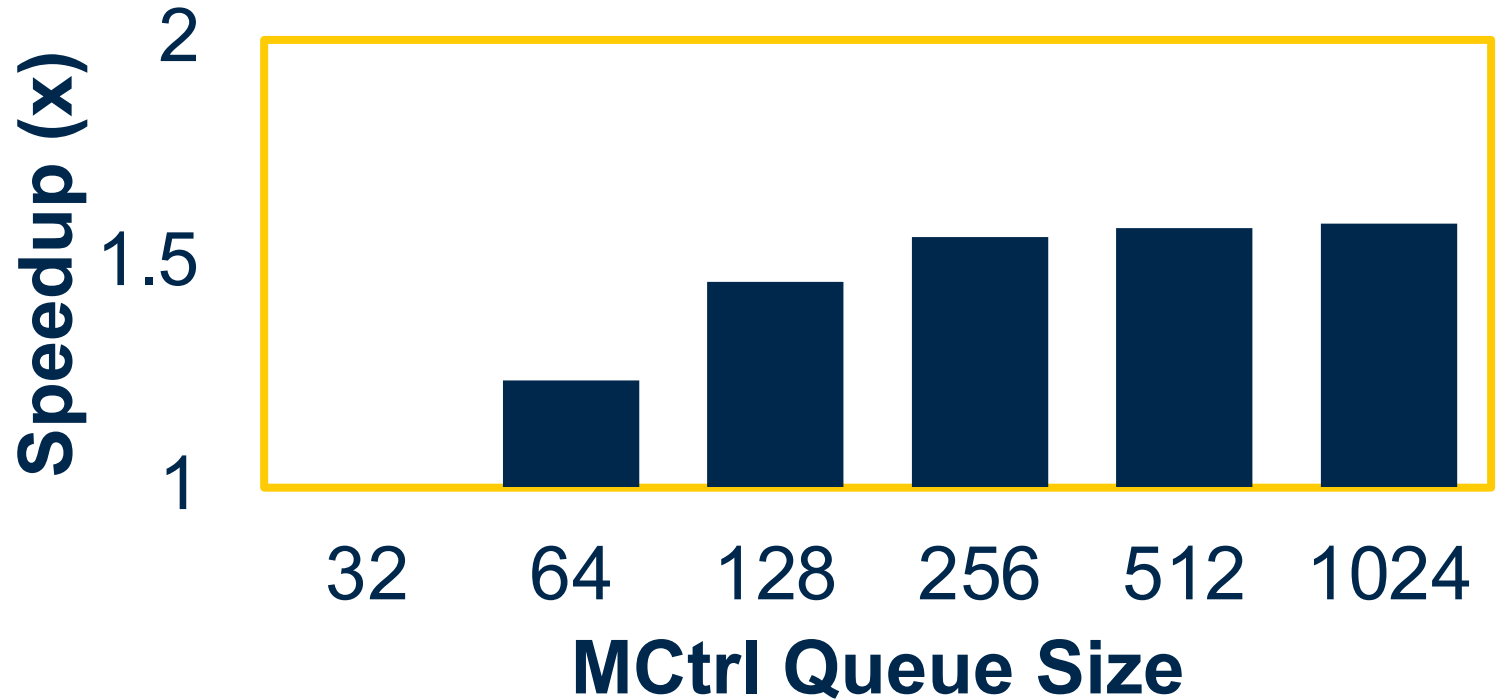
# Load Elision Unit



**Proposal:** Load elision unit prevents unnecessary loads by breaking symmetry in hardware

# Unexploited In-Memory Bandwidth

Effect of increasing  
memory controller  
queue size  
on GPM performance



Larger queue size:  
Better reordering opportunity

**Insight:** Size-limited memory controller queues prevent GPM workloads from exploiting in-DRAM parallelism

# Unexploited In-Memory Bandwidth

Effect of increasing

?

**How to reorder set operations from a large window size without hurting performance?**

Better reordering opportunity

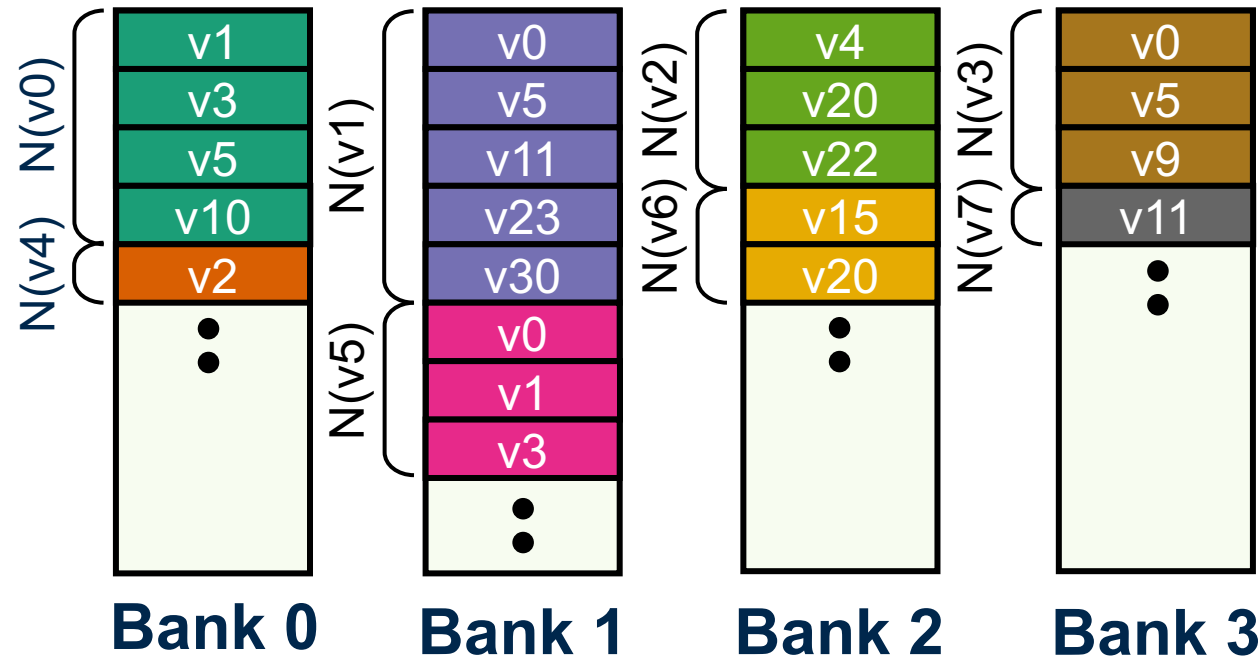
MCtrl Queue Size

**Insight:** Size-limited memory controller queues prevent GPM workloads from exploiting in-DRAM parallelism

# Set Operation Reordering

## Proposed Graph Neighborhood Mapping

(Vertex ID) % (Num Banks: 4) = Bank ID



## Set Operation Reordering

### Op. Seq.

int v0, v4  
 int v0, v5  
 diff v4, v5  
 diff v4, v1  
 int v4, v7  
 int v2, v4  
 diff v3, v6

### Bank IDs

v0, v4	%4 → 0,0
v0, v5	%4 → 0,1
v4, v5	%4 → 0,1
v4, v1	%4 → 0,1
v4, v7	%4 → 0,3
v2, v4	%4 → 2,0
v3, v6	%4 → 3,2

Addr	Bank#
N(v0)	0
N(v4)	0
N(v3)	3
N(v6)	2

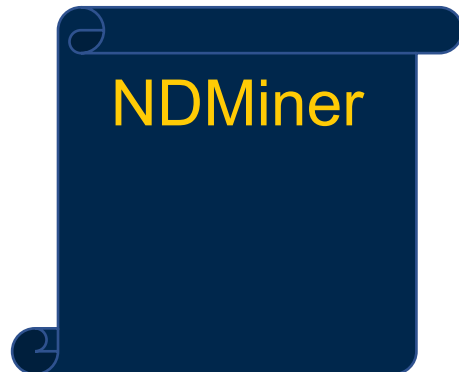
Read Queue (size = 4)

Reorder operations from a larger window to optimize bank-level parallelism

**Proposal:** Set operation reordering based on vertex IDs to improve bank-level parallelism

# More Details In The Paper

- Loop nest flattening for sparse GPM
- ISA extensions to support NDP
- NDMiner programming model
- Detailed NDP unit hardware design
- Modifications on the memory controller to support NDP
- Command scheduling
- . . .



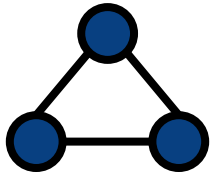
# Evaluation Methodology



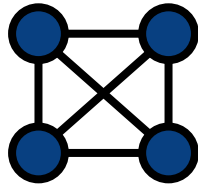
- Cycle-accurate simulation using Ramulator
  - Modeling of NDP logic based on detailed RTL models
- Comparison with state-of-the-art baselines
  - GraphPi [SC 2020] (software baseline)
  - GraphPi algorithms on GAPBS data structures (software baseline)
  - Pangolin [VLDB 2020] (software baseline)
  - FlexMiner [ISCA 2021] (hardware baseline)
- Five input graph datasets with diverse sizes and connectivity

# Evaluation Methodology

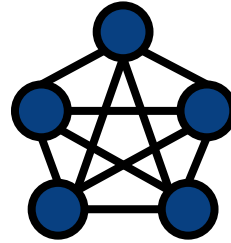
## Cliques (Dense Shapes)



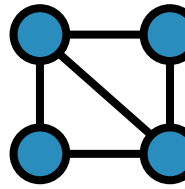
Triangle  
Counting



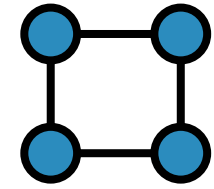
4-Clique Mining



5-Clique  
Mining



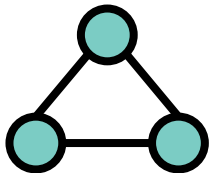
Diamond



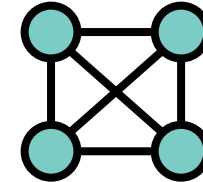
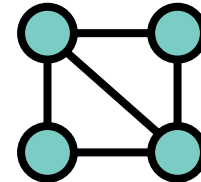
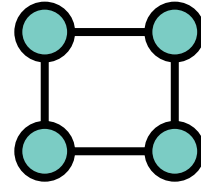
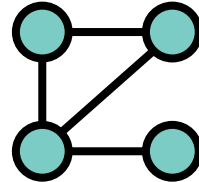
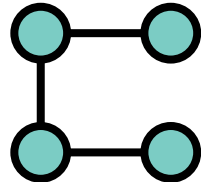
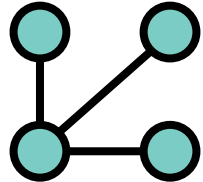
Four Cycle

## Sparse Shapes

## Mixed Shapes (Dense + Sparse)

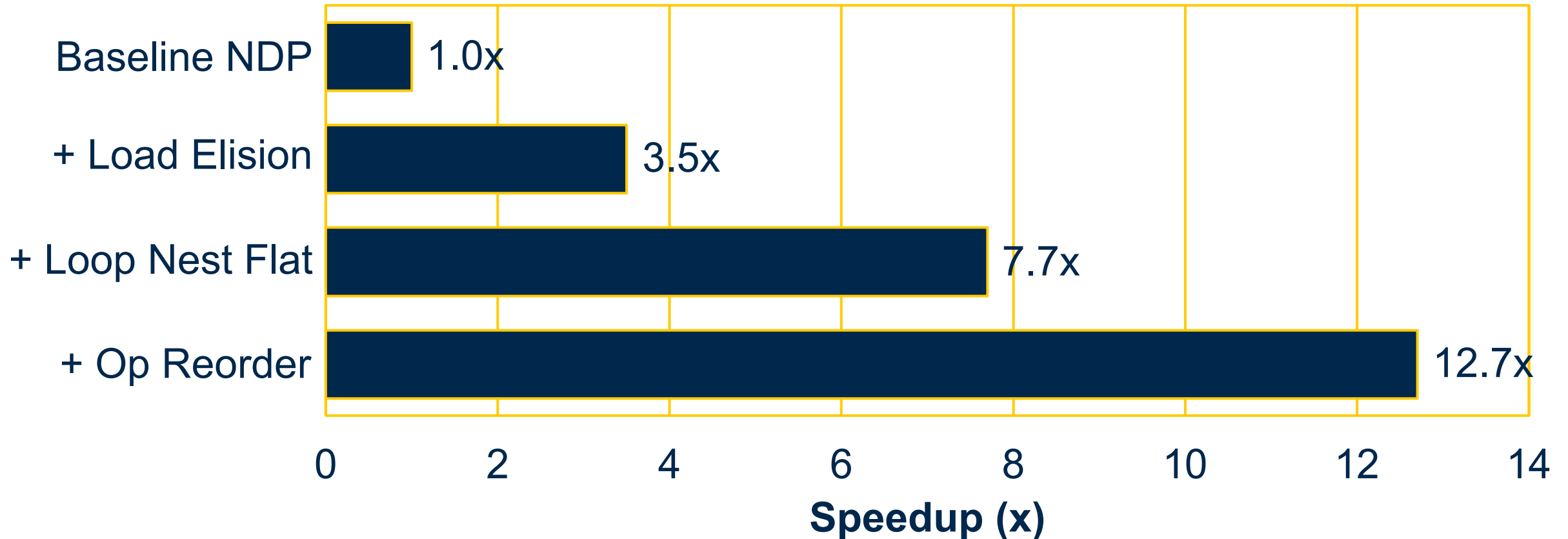


3-Motif Mining



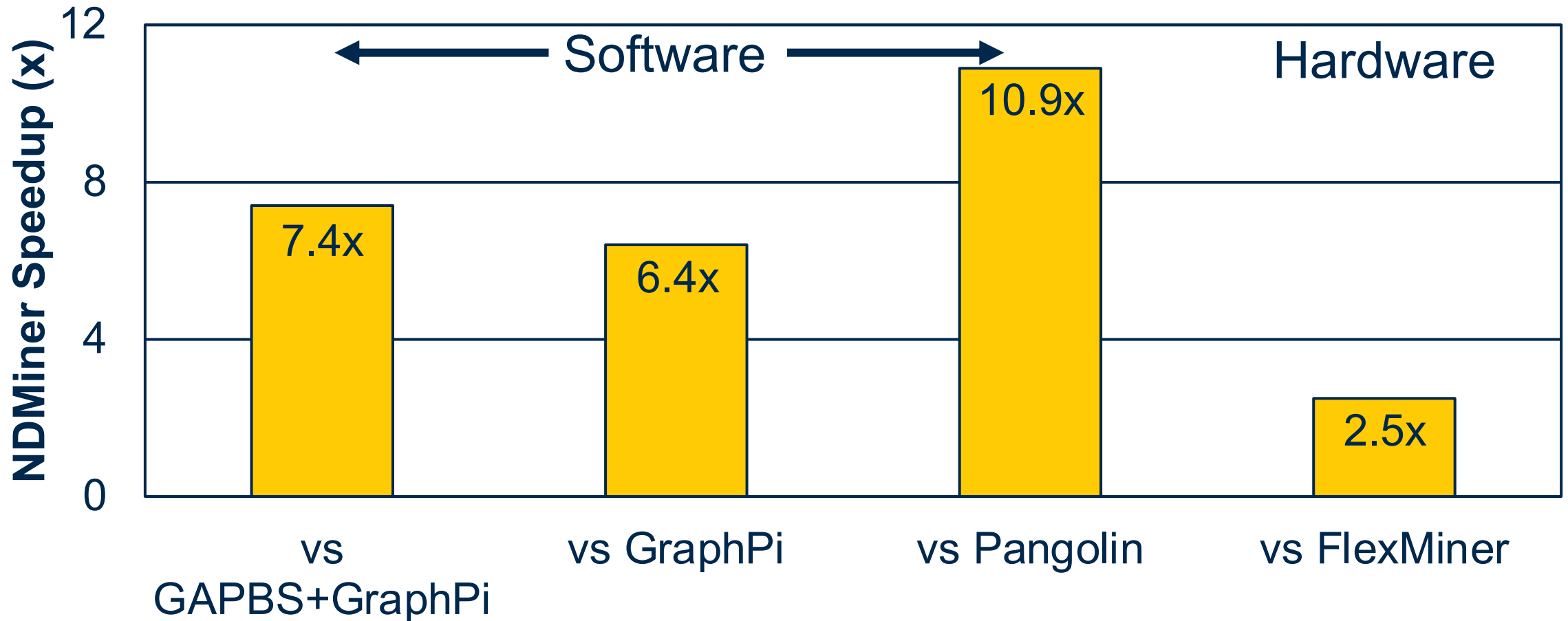
4-Motif Mining

# Effectiveness of Proposed Optimizations



Domain-specific optimizations improve the performance of NDMiner by 12.7x, on average, compared to a baseline NDP system

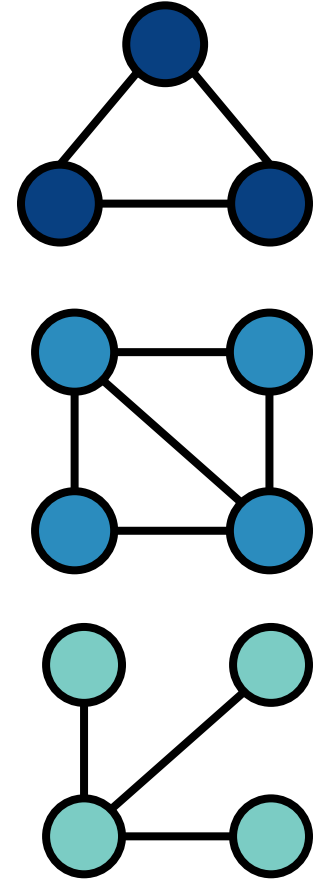
# Performance vs SOTA



NDMiner outperforms SOTA software baselines by 6.4-10.9x, on average, and SOTA hardware baselines by 2.5x, on average

# Conclusion

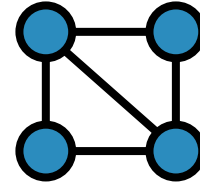
- GPM workloads bottlenecked by control flow and irregular memory accesses
- NDMiner: domain-specific near data processing (NDP) architecture
- Results in data movement and algorithmic complexity reduction, and improved parallelism to gain performance



**Thank You!**

# Backup Slides

# Redundant Loads in Sparse GPM



Mining Diamonds

Redundant fetching of  $N_{u_0u_1}$   
or  $u_2$  and  $u_3$

Widespread in sparse GPM

Not present in clique mining  
Fully connected nature leads to  
tighter constraints

```
1. for  $u_0$  in  $V$ :
2.    $N_{u_0} = G.out\_neigh(u_0)$ 
3.   for  $u_1$  in  $N_{u_0}$ :
4.     if  $u_1 \geq u_0$ : break
5.      $N_{u_1} = G.out\_neigh(u_1)$ 
6.      $N_{u_0u_1} = Intersection(N_{u_0}, N_{u_1})$ 
7.     for  $u_2$  in  $N_{u_0u_1}$ : Redundant
8.       for  $u_3$  in  $N_{u_0u_1}$ : set reads
9.         if  $u_3 \geq u_2$ : break
10.    num_diamonds++
```

Sparse GPM algorithms involve redundant reads

# Loop Nest Flattening

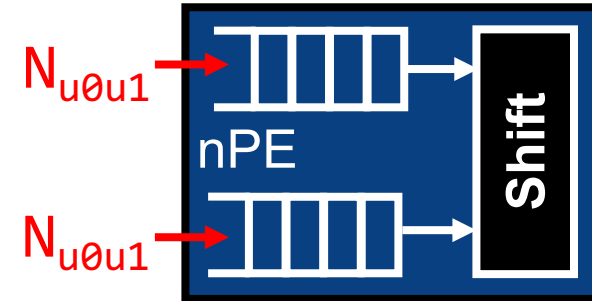
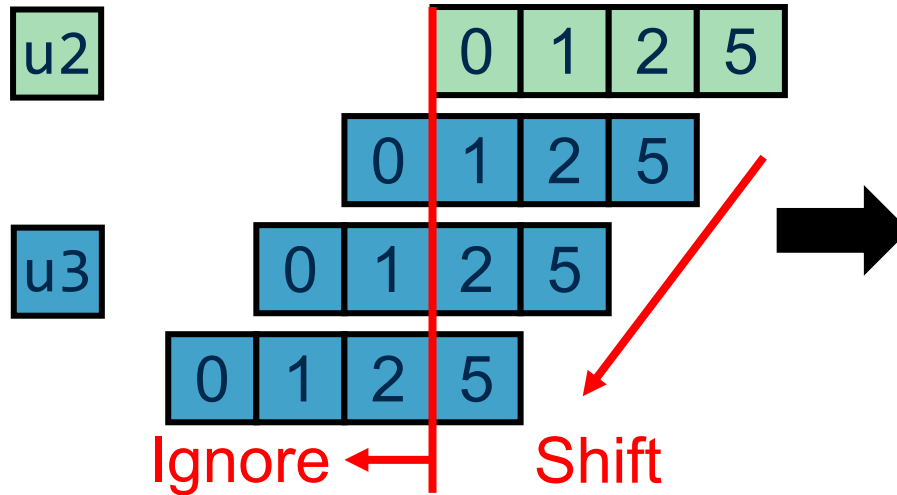
```

for u2 in Nu $\theta$ u1:
  for u3 in Nu $\theta$ u1:
    if u3 >= u2: break
  
```



$u2, u3 = \text{shift\_record}(N_{u\theta})$

$$u2, u3 \in N_{u\theta u1} = \{0, 1, 2, 5\}$$



$\{0, 1\} \{1, 2\} \{2, 5\}$   
 $\{0, 2\} \{1, 5\}$   
 $\{0, 5\}$

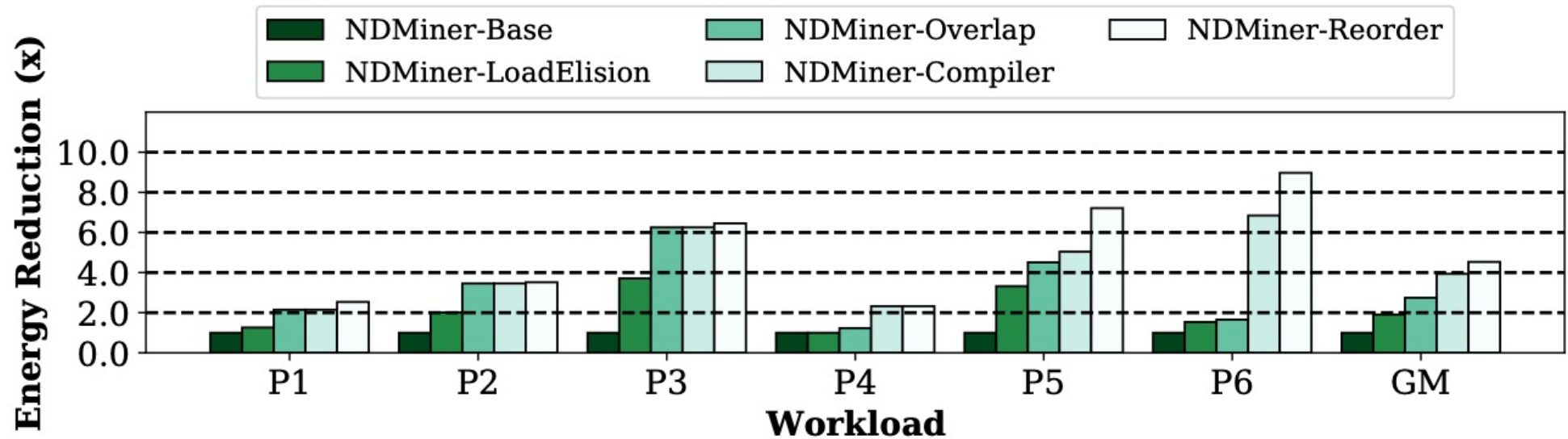
Reduction in algorithmic complexity of sparse GPM

# Overhead Analysis

	<b>Near-data PEs (nPEs)</b>	<b>Load Elision Unit (LEU)</b>	<b>Set Operation Reorder Unit</b>
<b>Location</b>	Buffer chip of DRAM	Buffer chip of DRAM	CPU memory controller front-end
<b>Area (mm<sup>2</sup>)</b>	0.01237	0.00096	0.4147
<b>Power (mW)</b>	18.45	0.36	32.76

16x nPEs and 32x LEUs in a DRAM DIMM

# Energy Consumption



Energy consumption of different NDMiner configurations on a representative **patents** dataset.

# Qualitative Comparison to Prior Works

	<b>Symm. Break.</b>	<b>NDP</b>	<b>Load Elision</b>	<b>Loop Nest Flattening</b>	<b>Op Reorder</b>
GraphZero [30]	✓	✗	✗	✗	✗
GraphPi [48]	✓	✗	✗	✗	✗
Gramer [60]	✗	✗	✗	✗	✗
FlexMiner [12]	✓	✗	✗	✗	✗
SISA [8]	✗	✓	✗	✗	✗
IntersectX [43]	✓	✗	✗	✗	✗
<b>NDMiner</b>	✓	✓	✓	✓	✓

# Input Graph Datasets

<b>Graph</b>	<b>#Vtx</b>	<b>#Edge</b>	<b>Size (MB)</b>	<b>Avg Degree</b>	<b>Description</b>
wiki-vote (wi)	7.1k	103.7k	0.5	14.6	Voting network
pokec (po)	1.6M	30.6M	129.3	19.1	Social network
patents (pa)	3.7M	16.5M	91.8	4.4	Citation network
livejournal (lj)	4.0M	34.7M	162.8	8.7	Social network
orkut (or)	3.1M	117.8M	470.5	38.1	Social network