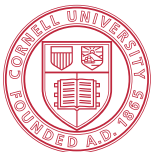


# A New Approach to Automatic Memory Banking using Trace-Based Address Mining

Yuan Zhou\*, Khalid Al-Hawaj\*, Zhiru Zhang

Computer Systems Lab  
Electrical and Computer Engineering  
Cornell University



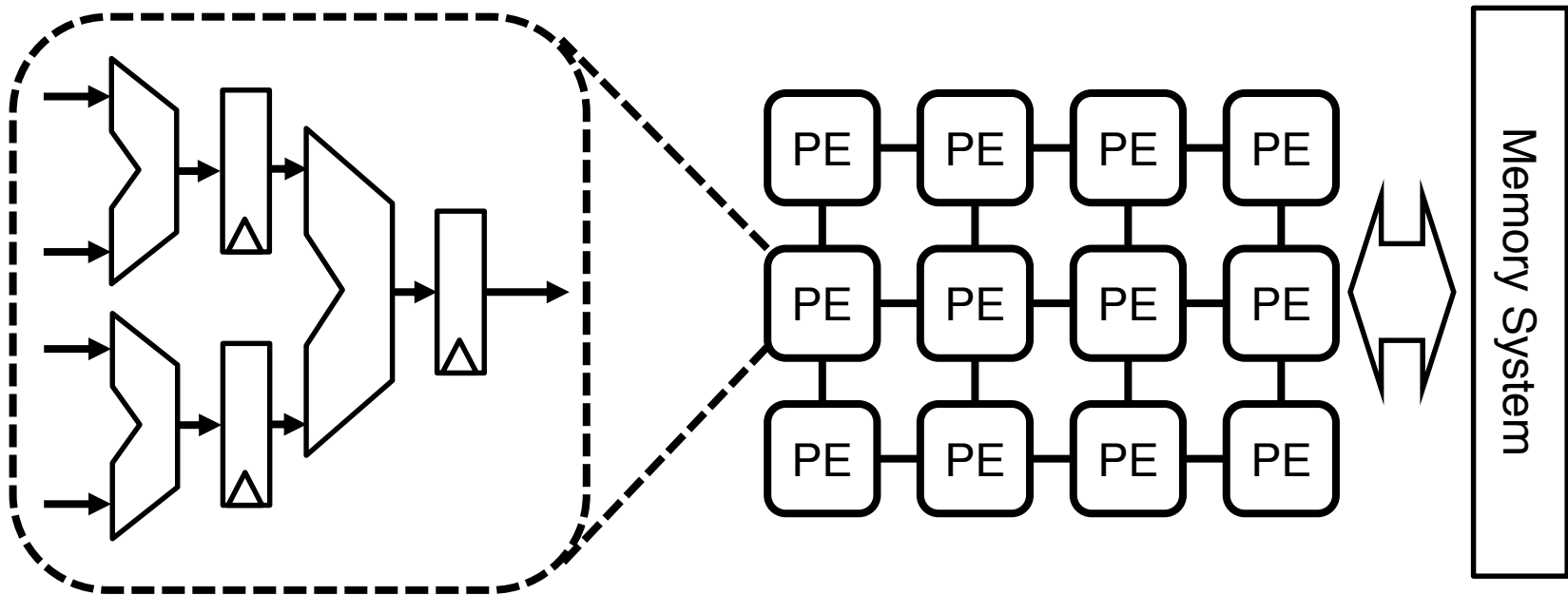
Cornell University

\* Equal contributions



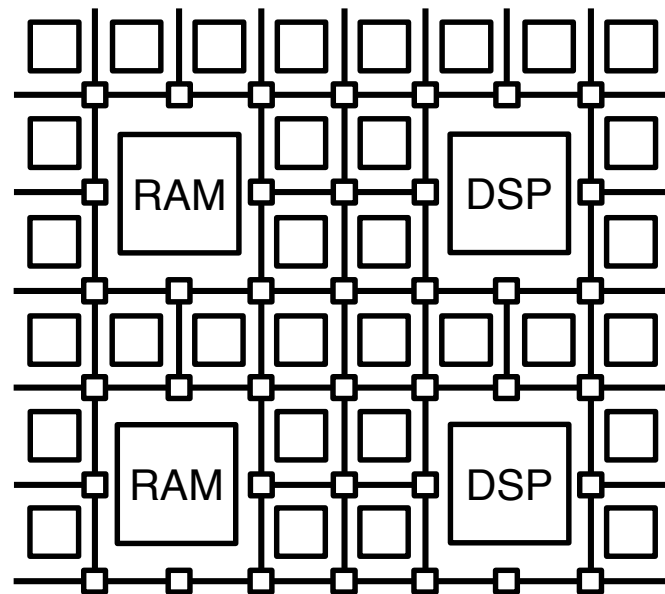
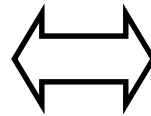
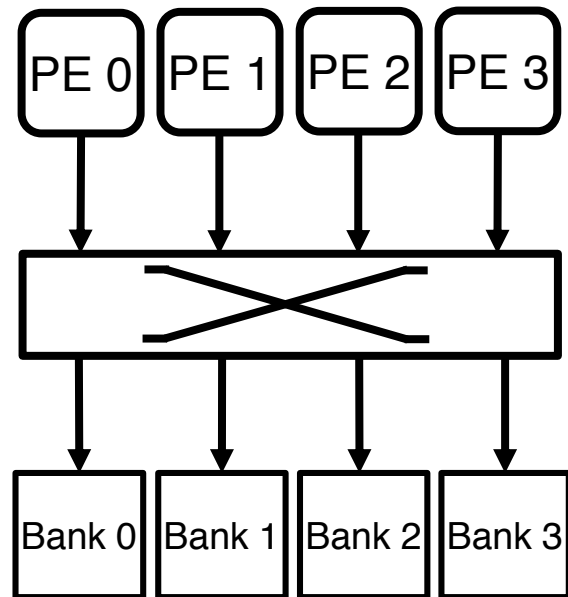
# FPGA Accelerators Demand High Bandwidth

- ▶ Modern FPGA accelerators maximize performance using highly parallel architecture
  - Many processing elements (PEs) with pipelined datapath
  - High bandwidth requirement



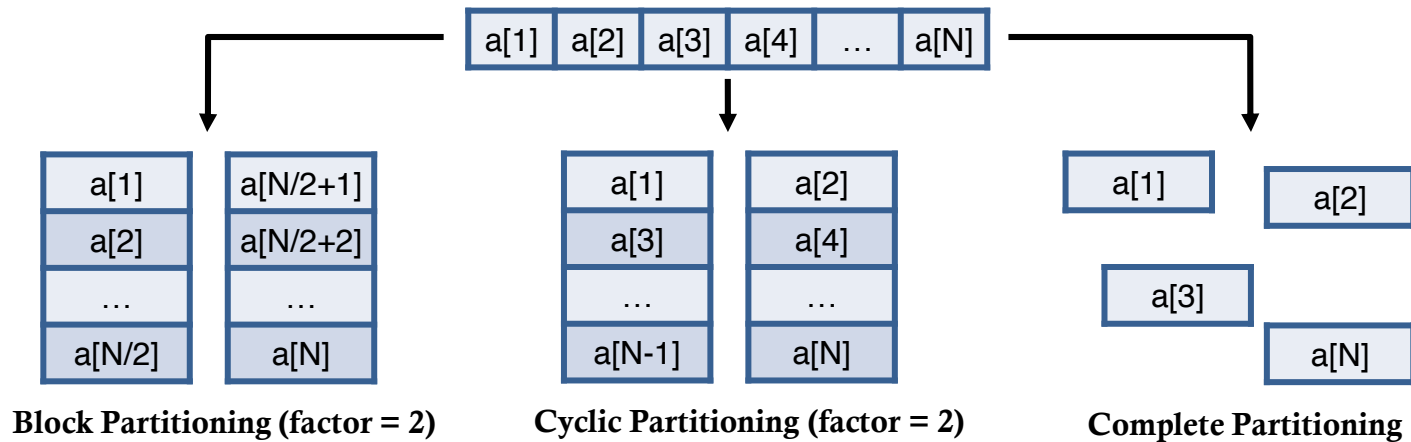
# Importance of Memory Banking

- ▶ Memory banking (partitioning) is commonly used in FPGA designs
  - Exploit abundant distributed memory resources in FPGA
  - Low storage overhead compared to duplication



# Memory Banking in HLS

- ▶ Commercial HLS tools support memory banking
  - Use cases: small/medium-size on-chip memories
  - Basic schemes: block, cyclic, and complete partitioning



- **Not automatic: user-specified pragmas required**

# Related Work

- ▶ Static compile-time techniques that automatically generate memory partitioning solutions for HLS
  
- ▶ Representative techniques
  - Linear-transformation-based memory partitioning [Wang et al., DAC'13] [Meng et al., DAC'15]
  - Generalized memory partitioning (GMP) [Wang et al., FPGA'14]
  - Lattice-based memory partitioning [Cilardo & Gallo, TACO'15]
  - Difference-based memory partitioning [Yin et al., ICCAD'16]
  - ...

# Pros & Cons of Compile-Time Memory Partitioning

- ▶ Efficient in runtime
- ▶ But sensitive to syntactic variances

```
for ( i = 0; i < N-1; i ++ )  
    sum += A[i] + A[i+1];
```

vs.

```
for ( i = 0; i < N-1; i ++ )  
    sum += A[i+(i%2)] + A[i+((i+1)%2)];
```

- ▶ Only effective for affine access patterns

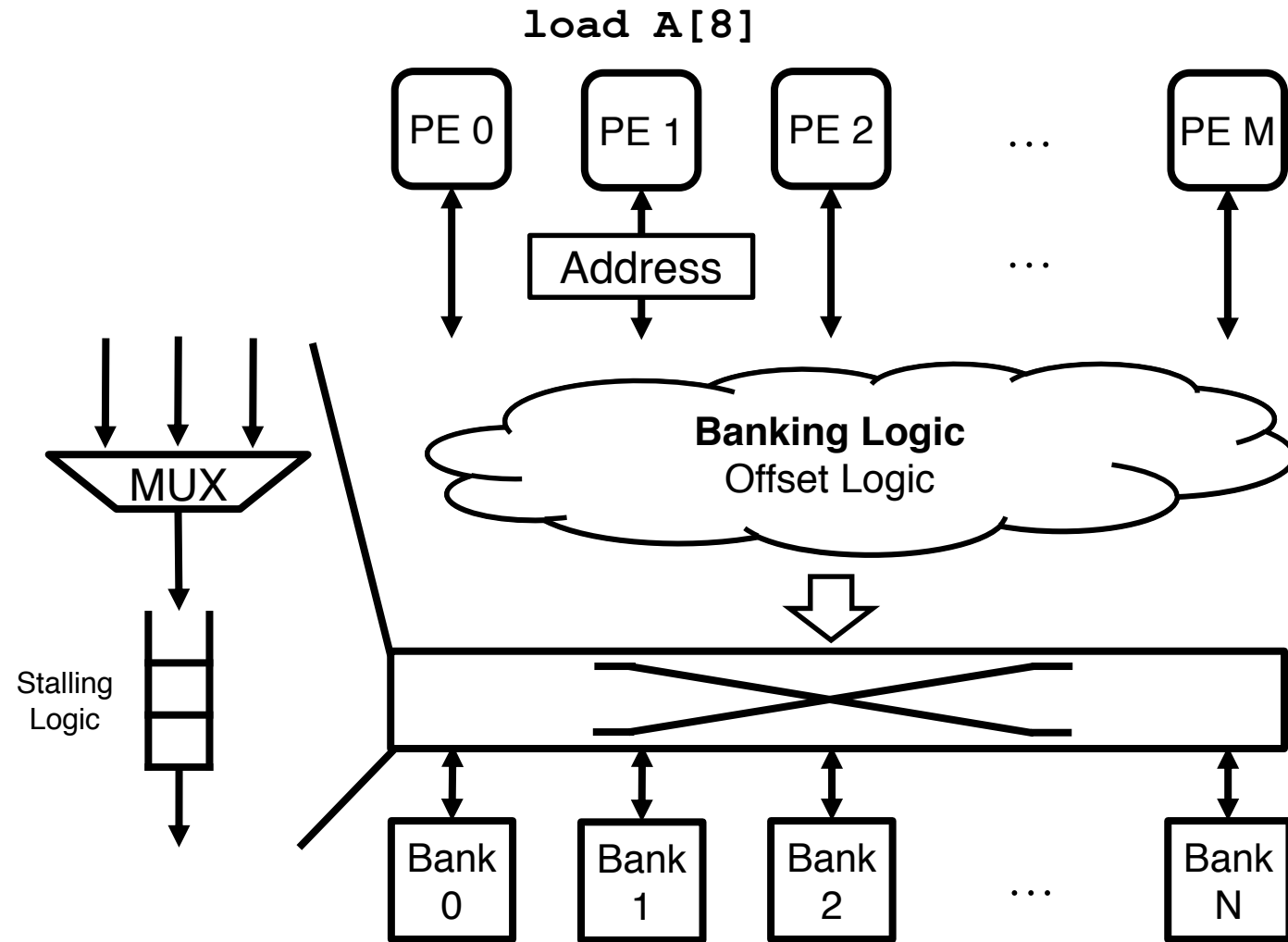
```
for ( i = 0; i < N-1; i ++ )  
    sum += A[i] + A[i+1];
```

vs.

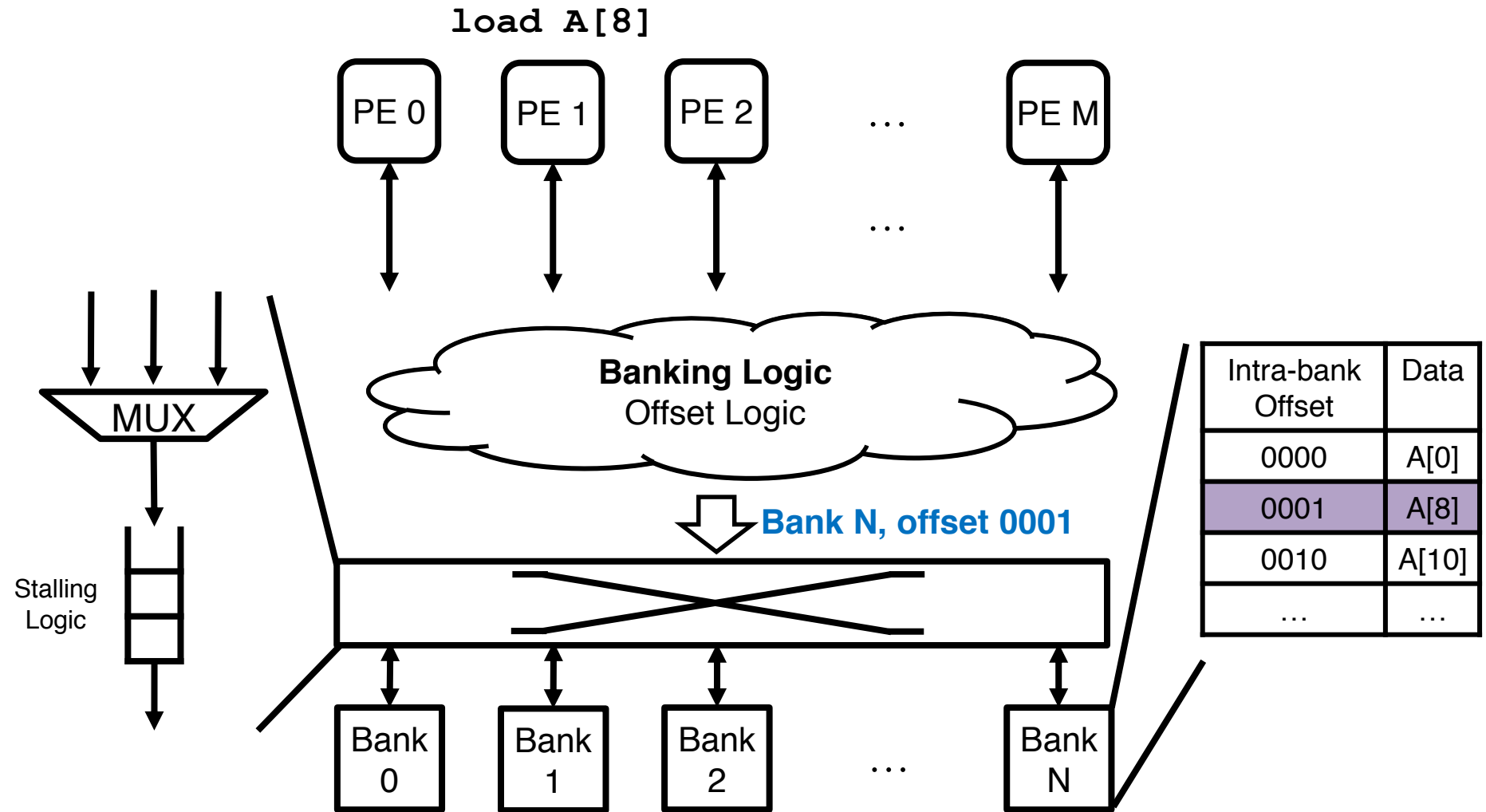
```
for ( i = 0; i < N-1; i ++ )  
    sum += A[B[i]] + A[B[i+1]];
```

- ▶ Only considers simple metrics of hardware complexity

# FPGA Accelerator with Banked Memory



# FPGA Accelerator with Banked Memory





# Our Proposal: Trace-based Memory Banking

- ▶ The memory trace of an application contains useful information about memory access pattern
  - Not sensitive to coding style
  - Not limited to affine memory accesses
  
- ▶ We explore the opportunity of finding memory banking solution with trace analysis
  - Focus on conflict-free banking schemes

# A Motivational Example

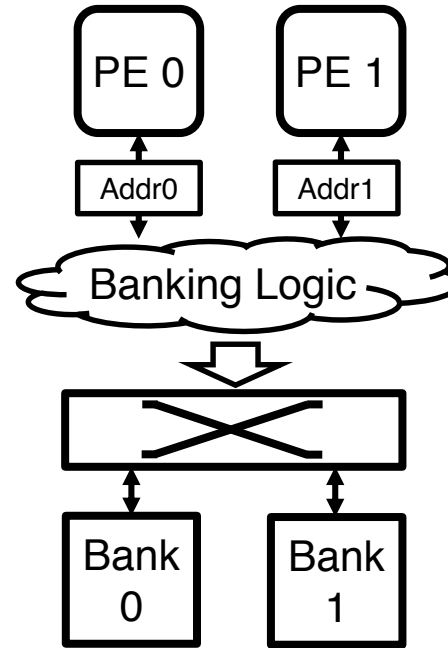
```
int A[N];
```

```
for ( int i = 0; i < N-1; i ++ )  
  compute(A[i], A[i+1]);
```



Iteration	Address0	Address1
0	000 <b>0</b>	000 <b>1</b>
1	000 <b>1</b>	001 <b>0</b>
...	...	...
10	101 <b>0</b>	101 <b>1</b>
...	...	...

- Mask: Least significant bit (LSB)
- Mask ID: value of LSB



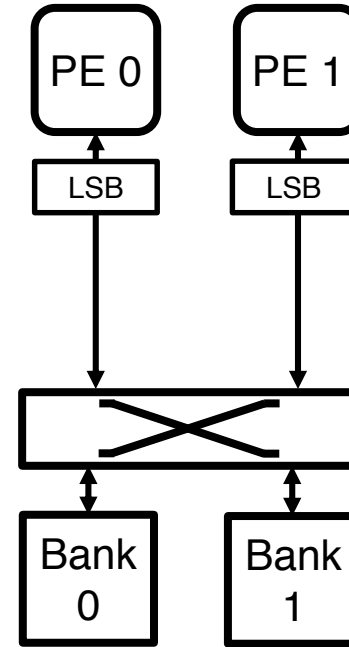
# A Motivational Example

```
int A[N];
```

```
for ( int i = 0; i < N-1; i ++ )  
  compute(A[i], A[i+1]);
```

Iteration	Address0	Address1
0	0000	0001
1	0001	0010
...	...	...
10	1010	1011
...	...	...

Bank0	Bank1
0000	0001
0010	0011
0100	0101
...	...



- Mask: Least significant bit (LSB)
- Mask ID: value of LSB

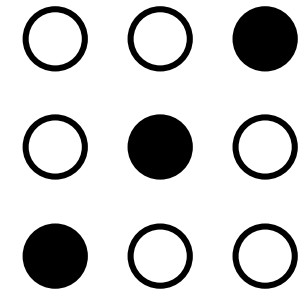
Cyclic partitioning, factor=2

**Important to identify the mask**

# Another Example

```
int A[Rows][Cols];  
  
for ( int i = 1; i < Rows - 1; i ++ )  
  for ( int j = 1; j < Cols - 1; j ++ )  
    compute(A[i-1][j+1], A[i][j], A[i+1][j-1]);
```

Memory  
access  
pattern



- ▶ Three parallel memory accesses per iteration
- ▶ Assumption: minimum number of memory banks

# Memory Trace and Initial Mask

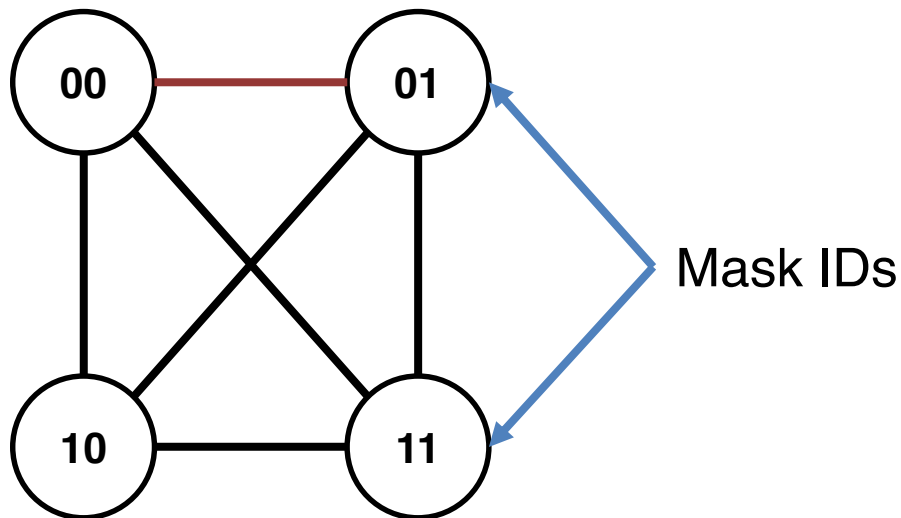
- ▶ Addresses are constructed by concatenating array indices
  - Assume both array indices are four bits

Iteration	Addr0 (i j)	Addr1 (i j)	Addr2 (i j)
0	0000 00 <b>10</b>	0001 00 <b>01</b>	0010 00 <b>00</b>
1	0000 00 <b>11</b>	0001 00 <b>10</b>	0010 00 <b>01</b>
2	0000 01 <b>00</b>	0001 00 <b>11</b>	0010 00 <b>10</b>
3	0000 01 <b>01</b>	0001 01 <b>00</b>	0010 00 <b>11</b>
...	...	...	...

- ▶ Start with the two LSBs as the mask, which distinguishes all three parallel memory accesses
- ▶ Banking function: Mask ID → Bank ID

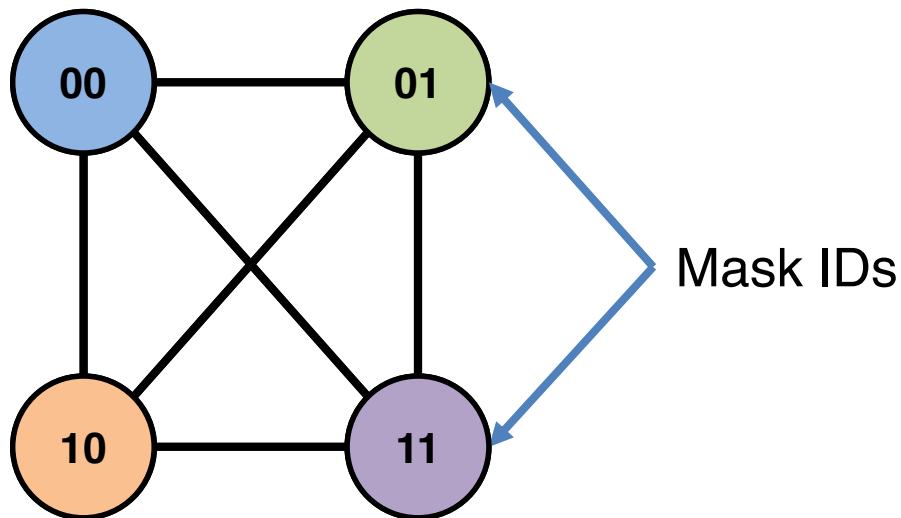
# Conflict Graph Construction

Iteration	Addr0	Addr1	Addr2
0	0000 00 <b>10</b>	0001 00 <b>01</b>	0010 00 <b>00</b>
1	0000 00 <b>11</b>	0001 00 <b>10</b>	0010 00 <b>01</b>
2	0000 01 <b>00</b>	0001 00 <b>11</b>	0010 00 <b>10</b>
3	0000 01 <b>01</b>	0001 01 <b>00</b>	0010 00 <b>11</b>
...	...	...	...



# Conflict Graph Construction

Iteration	Addr0	Addr1	Addr2
0	0000 00 <b>10</b>	0001 00 <b>01</b>	0010 00 <b>00</b>
1	0000 00 <b>11</b>	0001 00 <b>10</b>	0010 00 <b>01</b>
2	0000 01 <b>00</b>	0001 00 <b>11</b>	0010 00 <b>10</b>
3	0000 01 <b>01</b>	0001 01 <b>00</b>	0010 00 <b>11</b>
...	...	...	...



- Bank0
- Bank1
- Bank2
- Bank3

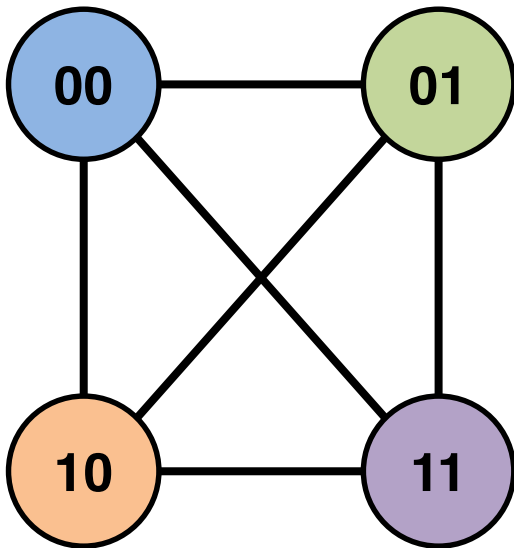
Not colorable with three colors



Impossible to find a three-bank solution

# Mask Selection

- ▶ Lower-bounding chromatic number of the conflict graph
  - Use max-clique as a heuristic to filter out mask candidates when max clique size  $>$  #banks



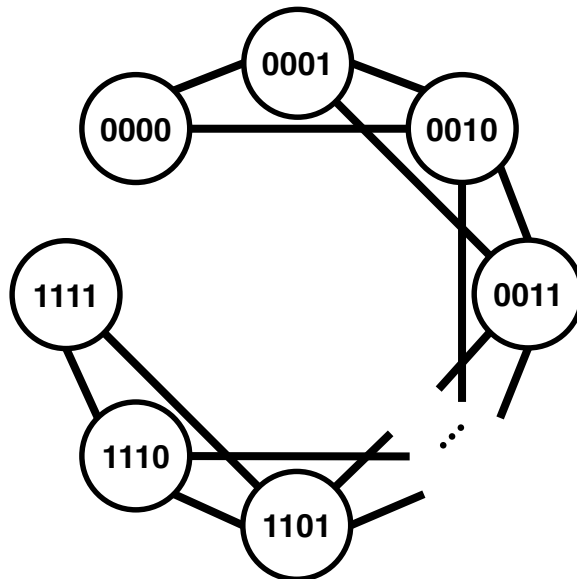
- Max clique size = 4  $>$  3, not colorable
- A different mask is needed



# Conflict Graph Coloring

- ▶ A good mask: all four bits in the 2<sup>nd</sup> array index ( j )

Iteration	Addr0	Addr1	Addr2
0	0000 0010	0001 0001	0010 0000
1	0000 0011	0001 0010	0010 0001
...	...	...	...

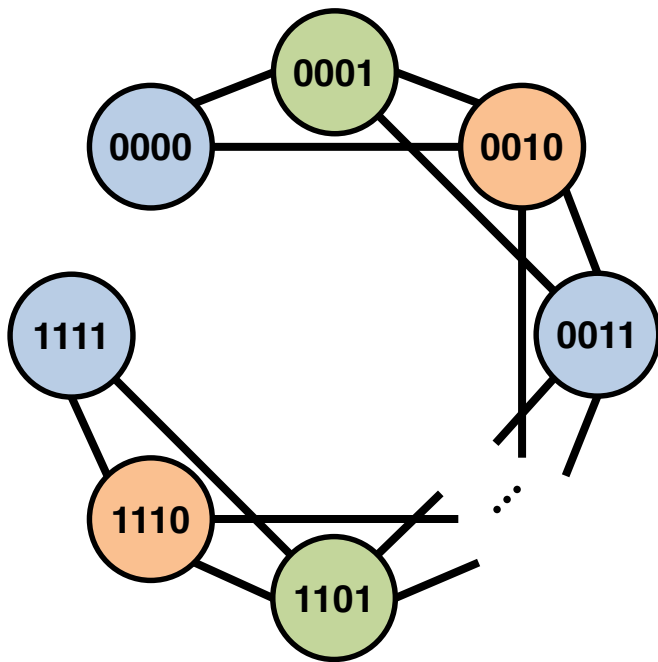


Order-based Heuristics →

← Evolutionary Algorithm

# Banking Function from Graph Coloring

- ▶ Conflict graph is colorable by three colors, solution found

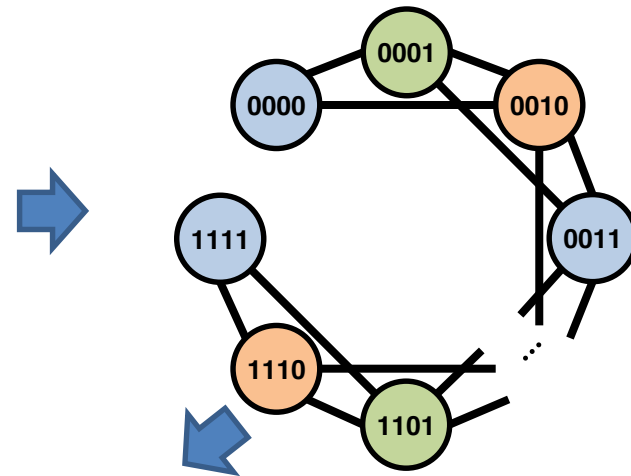


Mask ID	Bank
0000	0
0001	1
0010	2
...	...
1101	1
1110	2
1111	0

$$\begin{aligned} \text{Bank} &= \text{MaskID} \% 3 \\ &= j \% 3 \end{aligned}$$

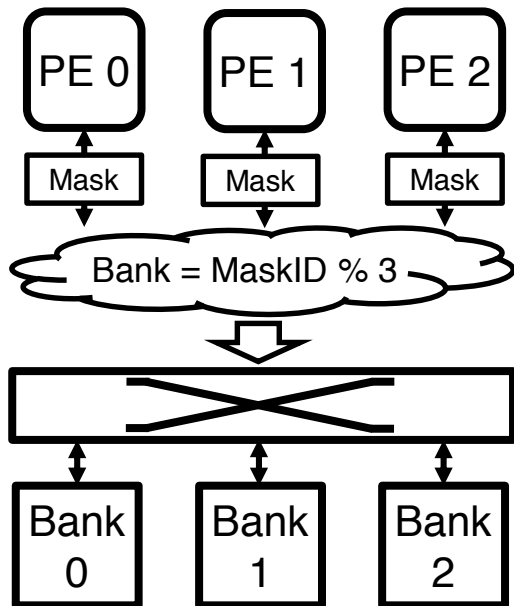
# Trace Analysis Uncovers Regularity in Access Pattern

Iteration	Addr0	Addr1	Addr2
0	0000 0010	0001 0001	0010 0000
1	0000 0011	0001 0010	0010 0001
...	...	...	...



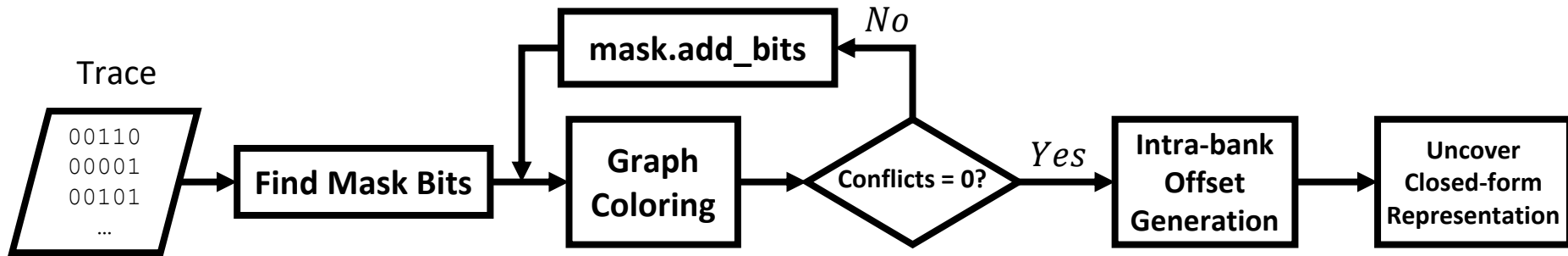
Mask ID	Bank ID
0000	0
0001	1
0010	2
...	...
1111	0

$$\text{Bank} = \text{MaskID} \% 3 = j \% 3$$



i/j	0	1	2	3	4	5	6	7
0	0	1	2	0	1	2	0	1
1	0	1	2	0	1	2	0	1
2	0	1	2	0	1	2	0	1
3	0	1	2	0	1	2	0	1
4	0	1	2	0	1	2	0	1
5	0	1	2	0	1	2	0	1
6	0	1	2	0	1	2	0	1
7	0	1	2	0	1	2	0	1

# The Complete Flow: TraceBanking



- ▶ A two-step approach to generate banking solution
  1. Identify the mask from representative memory trace
  2. Find an optimized banking solution using graph coloring
- ▶ Intra-bank offset generation
- ▶ Represent solution with closed-form equations if possible

# SMT-Based Verification

- ▶ Verify that a representative trace results in a zero-conflict banking solution
  - SMT problem formulation
  - Basic compiler support (or instrumentation) for obtaining loop bounds and array indices
- ▶ Also useful to verify compile-time banking techniques, which are supposed to be correct by construction

```
int A[Rows][Cols];  
  
for ( int i = 1; i < Rows - 1; i ++ )  
  for ( int j = 1; j < Cols - 1; j ++ )  
    compute(A[i-1][j+1], A[i][j], A[i+1][j-1]);
```

```
/* SMT problem formulation */  
// banking function  
int bank(int i, int j) {  
  int mask = j;  
  return j % 3;  
}  
  
// iteration domain  
assert((i > 0) && (i < Rows - 1));  
assert((j > 0) && (j < Cols - 1));  
// has conflicts?  
assert(  
  (bank(i-1, j+1) == bank(i, j)) ||  
  (bank(i-1, j+1) == bank(i+1, i-1)) ||  
  ...  
  || (bank(i, j) == bank(i+1, j-1))  
);  
// solution is free of banking conflicts if  
// the problem is unsatisfiable
```

# Case Study: Face Detection

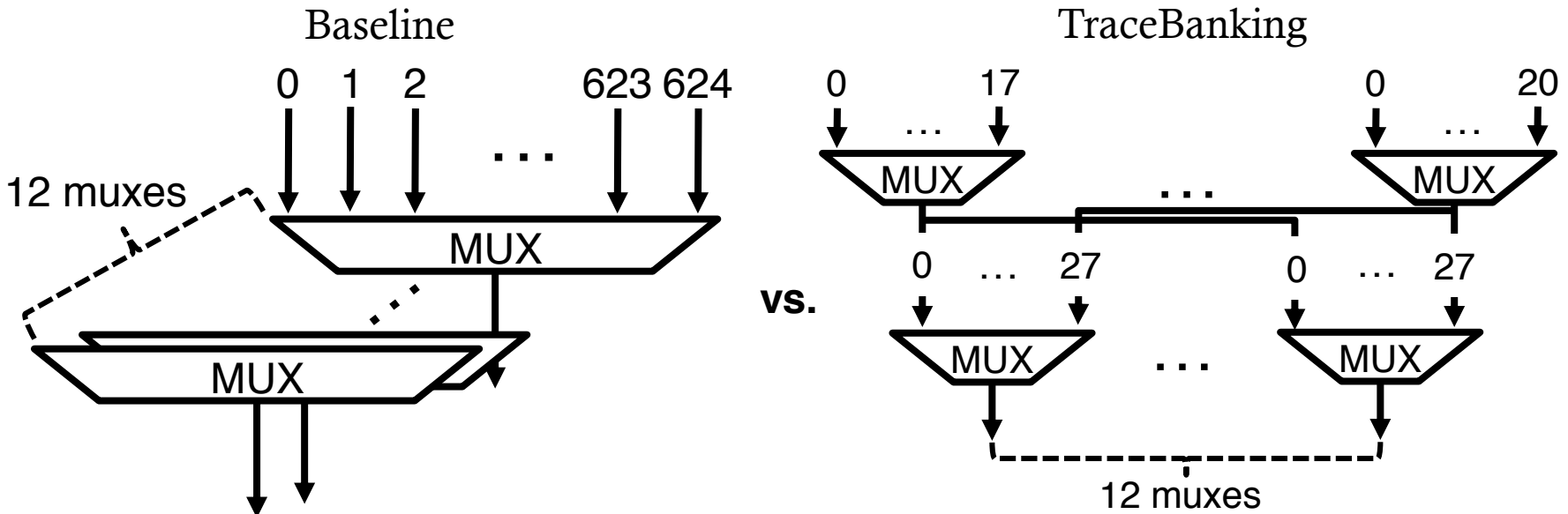
- ▶ Haar algorithm
  - Detects human faces with cascaded weak classifiers
- ▶ Window buffer fully partitioned for high throughput
- ▶ Difficult for existing techniques
  - Indirect array accesses
  - Variable number of accesses per iteration
- ▶ Baseline: Full Mux design
  - 12 instances of 625-to-1 multiplexers

```
pixel IntImg[25][25];
#pragma HLS array_partition
      variable=IntImg complete
pixel c[12];
int filter_no;

CLASSIFIER:
for (filter_no=0; filter_no<2913; filter_no++) {
  #pragma HLS pipeline II=1
  // read array indexes from look-up tables
  r0.x = rectangles_array0[filter_no];
  r0.y = rectangles_array1[filter_no];
  ...
  // access 8 data elements from array
  c[0] = IntImg[r0.y][r0.x];
  c[1] = IntImg[r0.y][r0.x+r0.w];
  ...
  // if condition met, access 4 more elements
  if ((r2.w != 0) && (r2.h != 0)) {
    c[8] = IntImg[r2.y][r2.x];
    ...
  }
  else {
    c[8] = 0;
    ...
  }
  // process data
  classify(c);
}
```

# Banking vs. Full Mux

- ▶ Certain addresses are never accessed at the same time, they can be grouped into the same bank
  - 28-bank solution, each bank has ~20 data elements



- ▶ Improve area with two stages of narrower multiplexers

# Comparison with Baseline

- ▶ Area, timing and latency result compared with baseline

Design	Slice	LUT	FF	DSP	BRAM	CP (ns)	Latency
Baseline	21275	53553	23785	3	22	9.22	2919
TraceBanking	4915	8266	12559	6	34	4.52	2923
	-76.9%	-84.6%	-47.2%			-51.0%	

- ▶ TraceBanking solution is incorporated into a complete face detection design

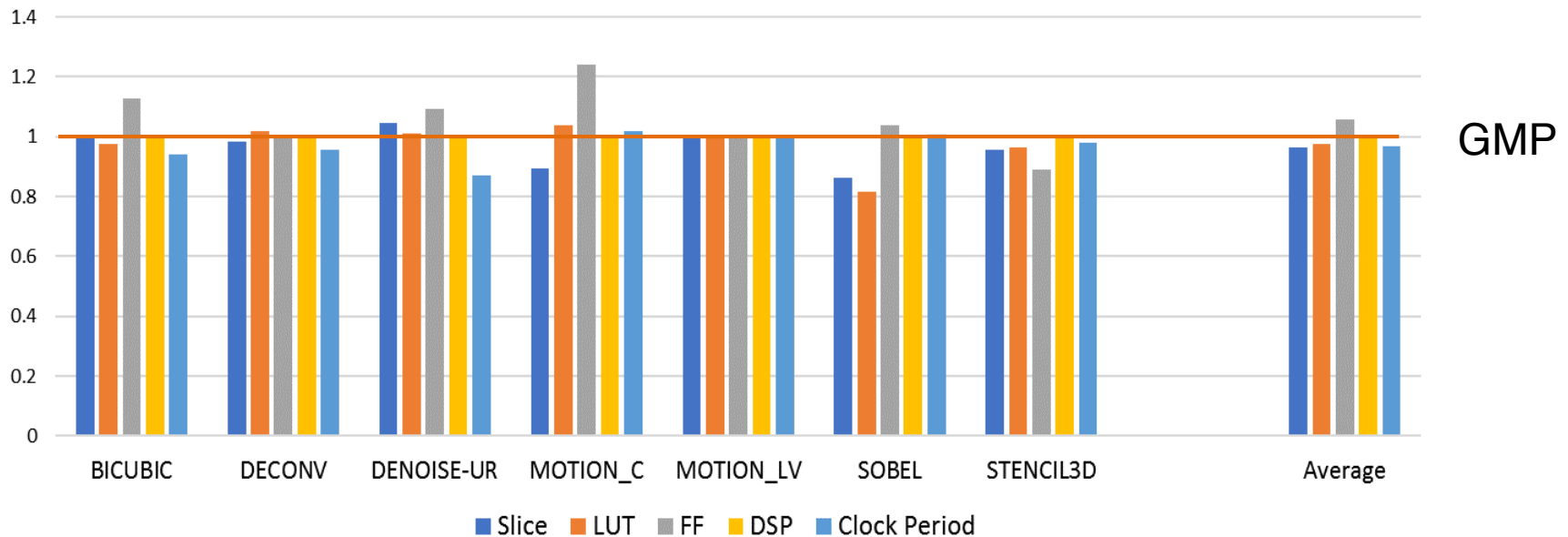


# Results on Benchmarks with Affine Accesses

## ▶ Baseline

- Hand-written synthesizable HLS C++ using linear coefficients generated by the GMP algorithm [Wang et al., FPGA'14]

## ▶ Area and timing result normalized to baseline



# Conclusion and Future Work

- ▶ Automatic memory banking is necessary for current and future HLS applications
- ▶ Trace-based memory banking is a promising approach
  - More flexible
  - Complementary to compile-time techniques
- ▶ Future work
  - Explore conflict-less banking solutions
  - Extend to generate other specialized memory systems
    - Data reuse buffers