iPIM: Programmable In-Memory Image Processing Accelerator Using Near-Bank Architecture

Peng Gu*[1], Xinfeng Xie*[1], Yufei Ding[1]

Guoyang Chen[2], Weifeng Zhang[2], Dimin Niu[2], Yuan Xie[1,2]

[1]University of California, Santa Barbara

[2]Alibaba Group

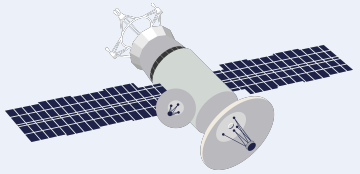
*co-primary first authors

Image Processing: Application Scenarios


Smartphone Camera

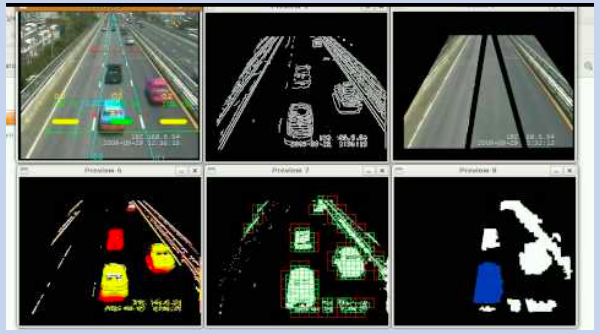

Traffic Camera

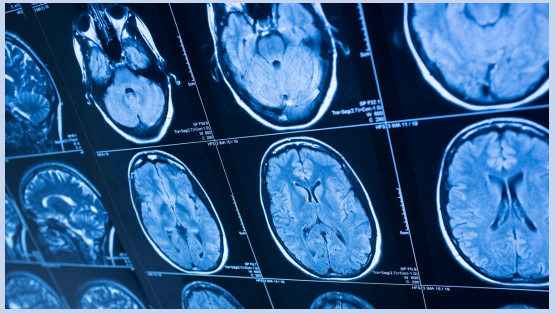

Medical Imaging Equipment

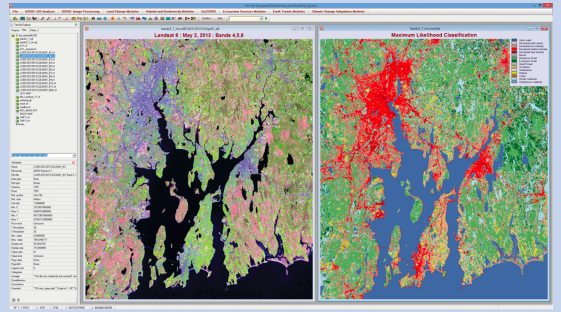

Satellite Imaging

Data Center, Work Station


Machine Learning


Traffic Analysis


Medical Image Processing


Geographic Information System

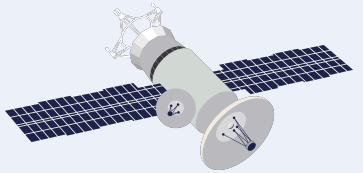
http://www.nectec.or.th/2008/r-d/img.html
https://www.surrey.ac.uk/centre-vision-speech-signal-processing/research/m-lab-biomedical-imaging-and-processing
https://clarklabs.org/terrset/idrisi-image-processing/

Image Processing: Application Scenarios


Smartphone Camera


Traffic Camera


Medical Imaging Equipment


Satellite Imaging

Data Center, Work Station

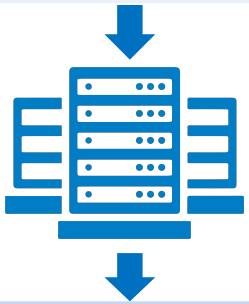
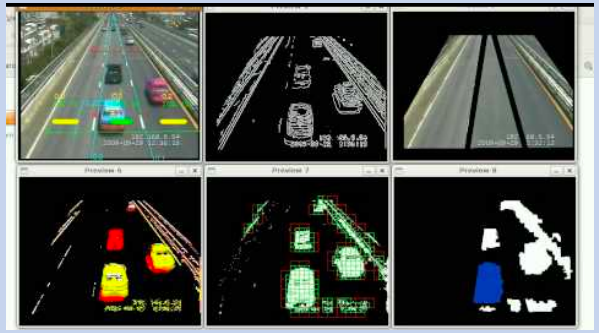
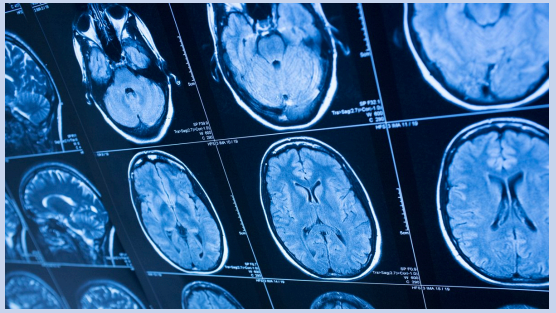


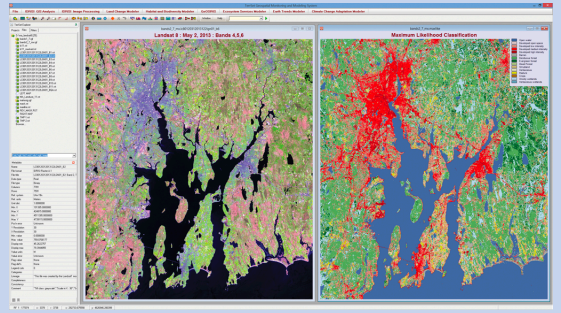
Image Processing Accelerator

- Domain Programmability
- High Performance
- High Energy Efficiency


Machine Learning


Traffic Analysis


Medical Image Processing


Geographic Information System

http://www.nectec.or.th/2008/r-d/img.html
https://www.surrey.ac.uk/centre-vision-speech-signal-processing/research/m-lab-biomedical-imaging-and-processing
https://clarklabs.org/terrset/idrisi-image-processing/

Image Processing: Characteristics

- Image Processing algorithms consist of pipeline stages that are both ***wide*** and ***heterogeneous***
  - Each stage is *wide* – Example: Image Blur

Image blur filter

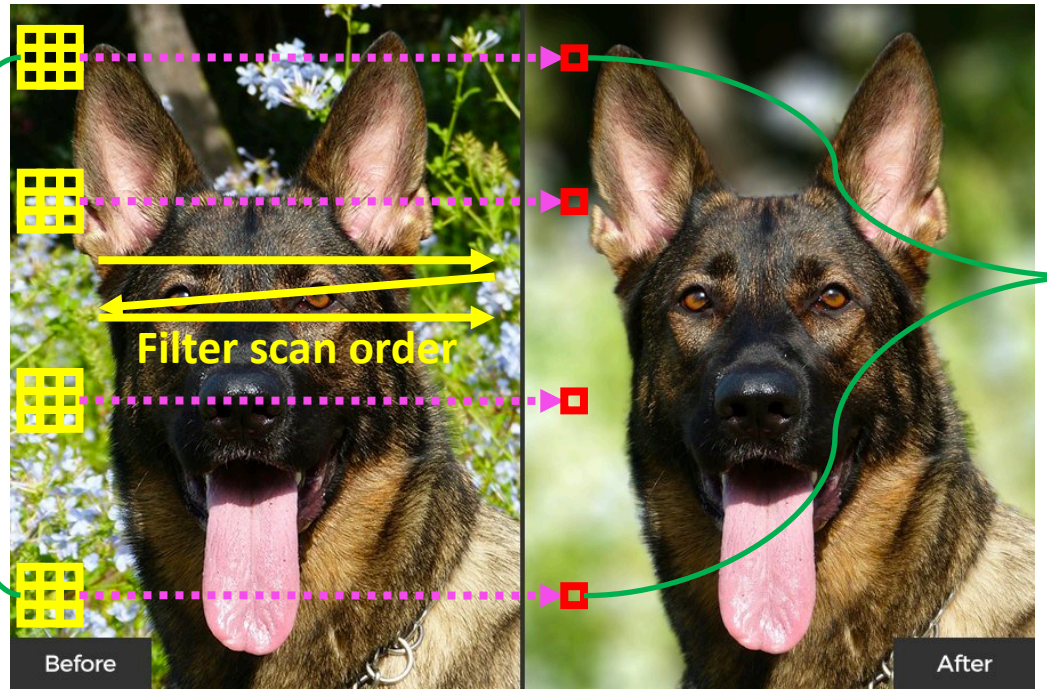
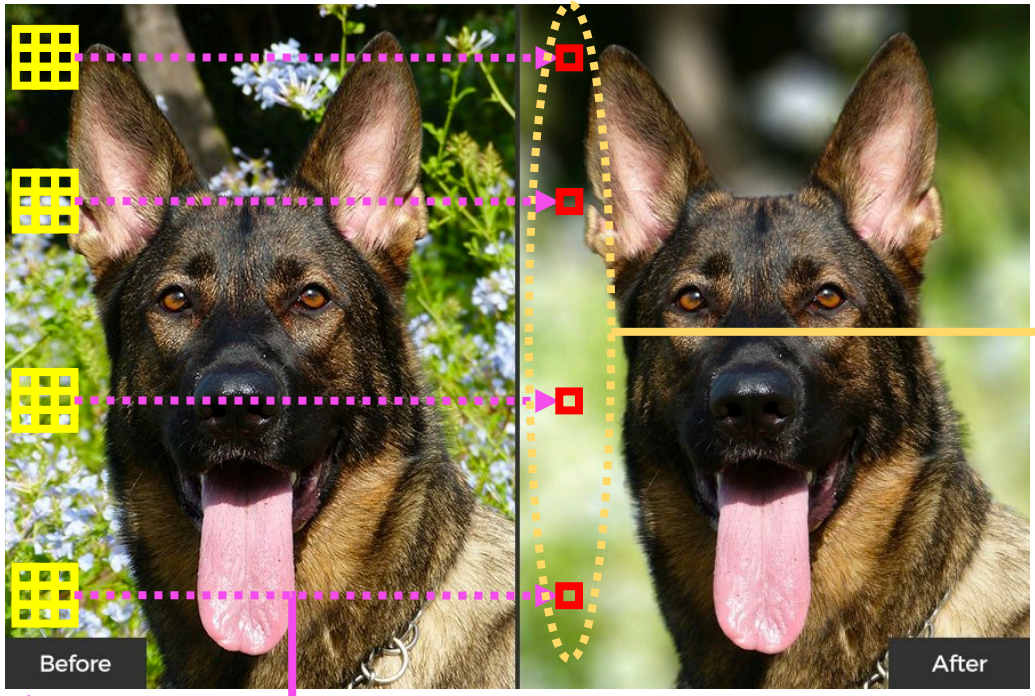


Image blur output

Image Processing: Characteristics

- Image Processing algorithms consist of pipeline stages that are both ***wide*** and ***heterogeneous***
  - Each stage is *wide* – Example: Image Blur

❶ Memory-level parallelism among pixels

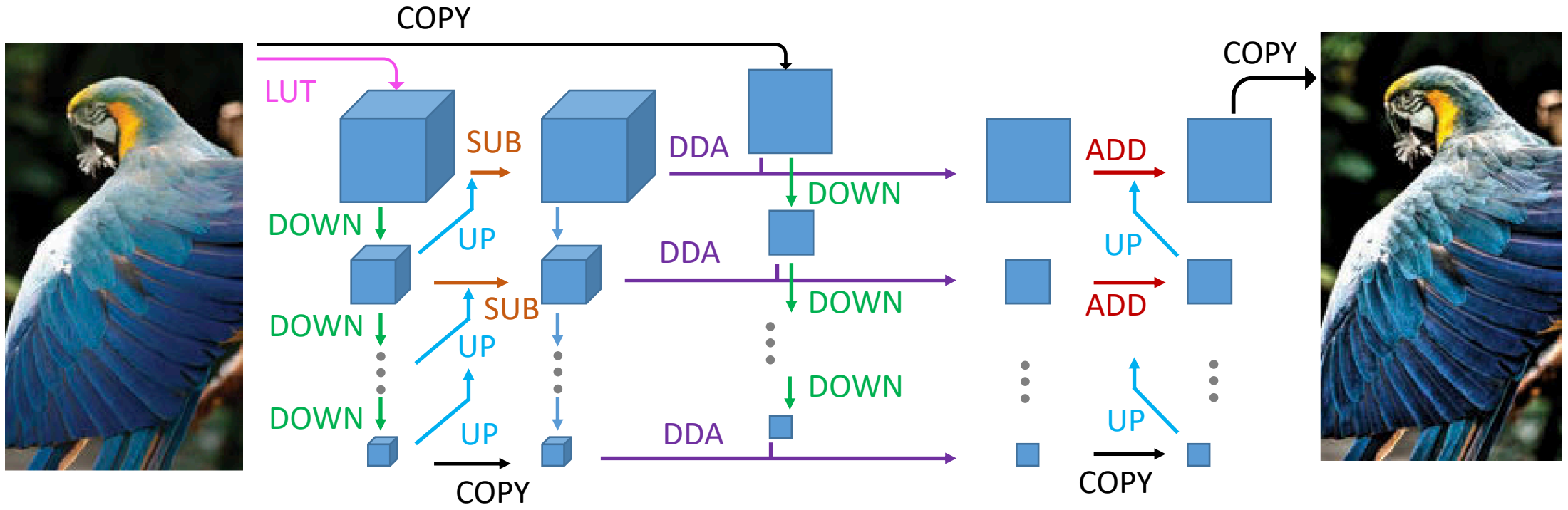


❷ Large size of intermediate data

❸ Low arithmetic density operations

Image Processing: Characteristics

- Image Processing algorithms consist of pipeline stages that are both ***wide*** and ***heterogeneous***
  - Overall pipelines are *heterogeneous* – Example: Local Laplacian Filter



https://dl.acm.org/doi/pdf/10.1145/2499370.2462176

Image Processing: Characteristics

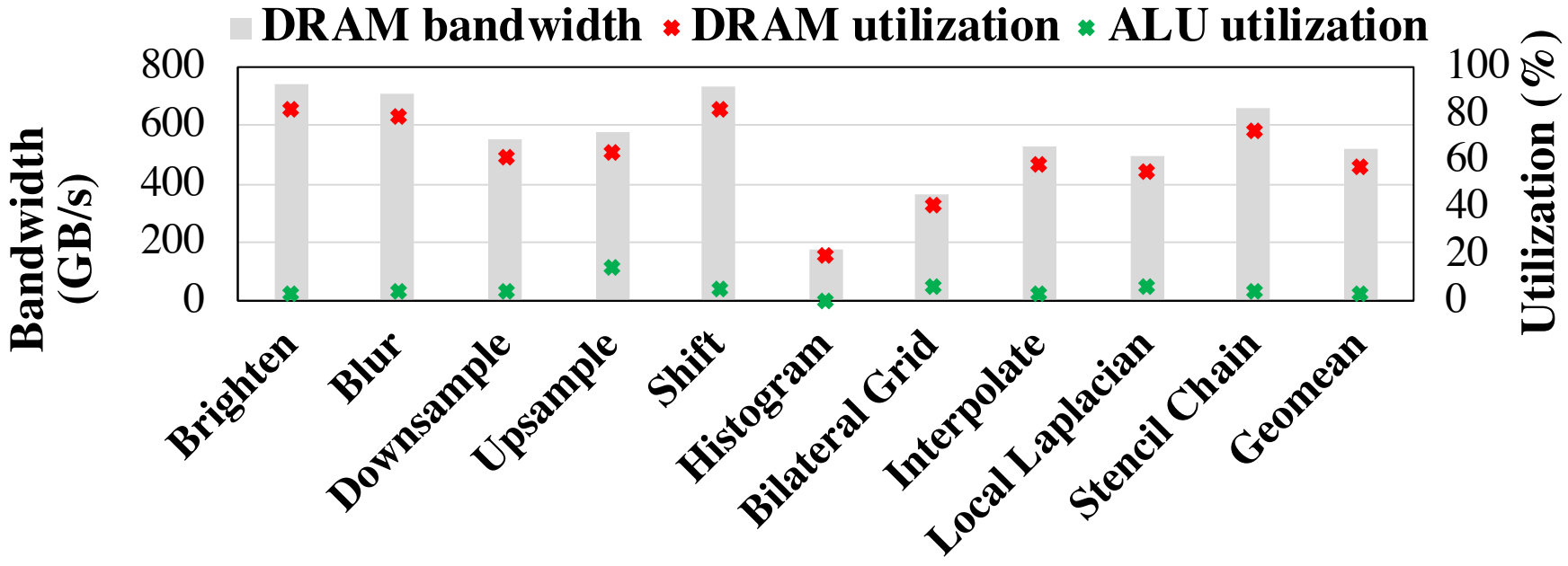
- Image Processing Algorithms consist of pipeline stages that are both ***wide*** and ***heterogeneous***

Image processing workloads have high memory bandwidth demand:

- Software: low temporal reuse due to
  - (1) low arithmetic density
  - (2) difficulty of pipeline fusion
- Hardware: on-chip cache cannot hold all intermediate data

Motivation Data: Memory Bandwidth Bottleneck

- On average: **57.55%** memory utilization v.s. **3.43%** ALU utilization
  - Benchmark: single-stage / multi-stage kernels
  - Configuration: Halide toolchain on a Tesla V100 GPU

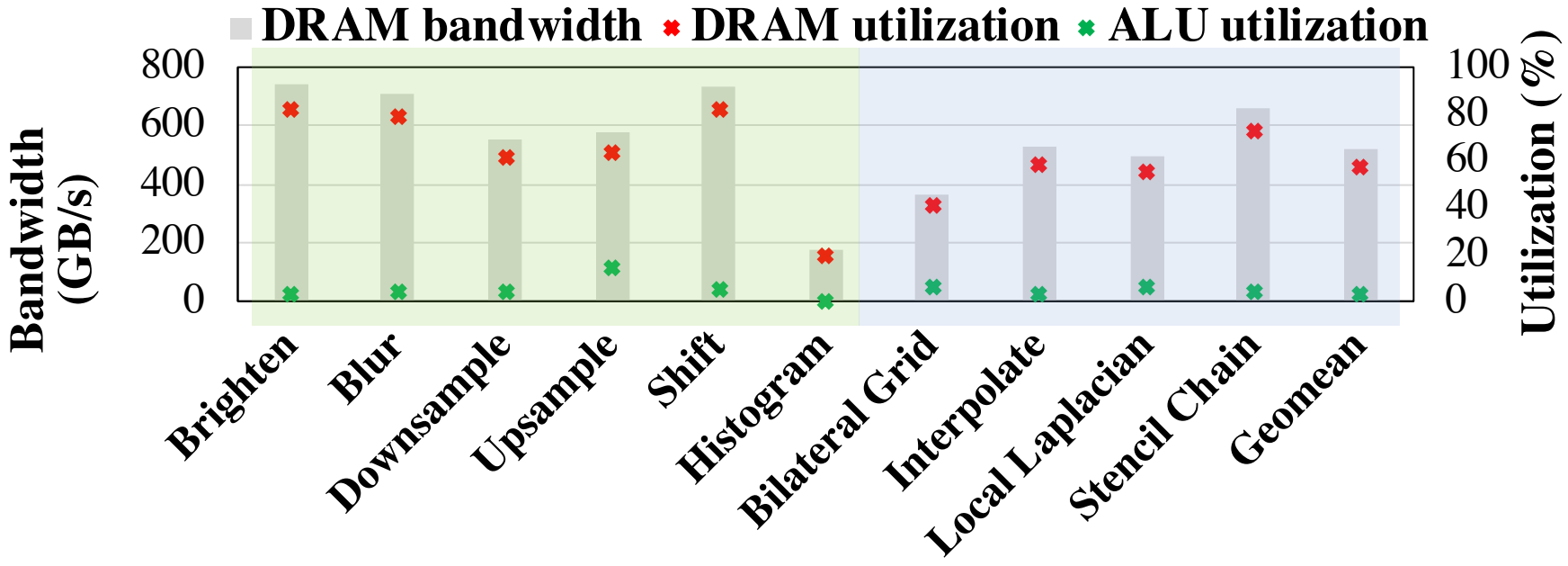


Motivation Data: Memory Bandwidth Bottleneck

Single-stage kernels
ALU utilization: **2.85%**
DRAM utilization: **58.80%**

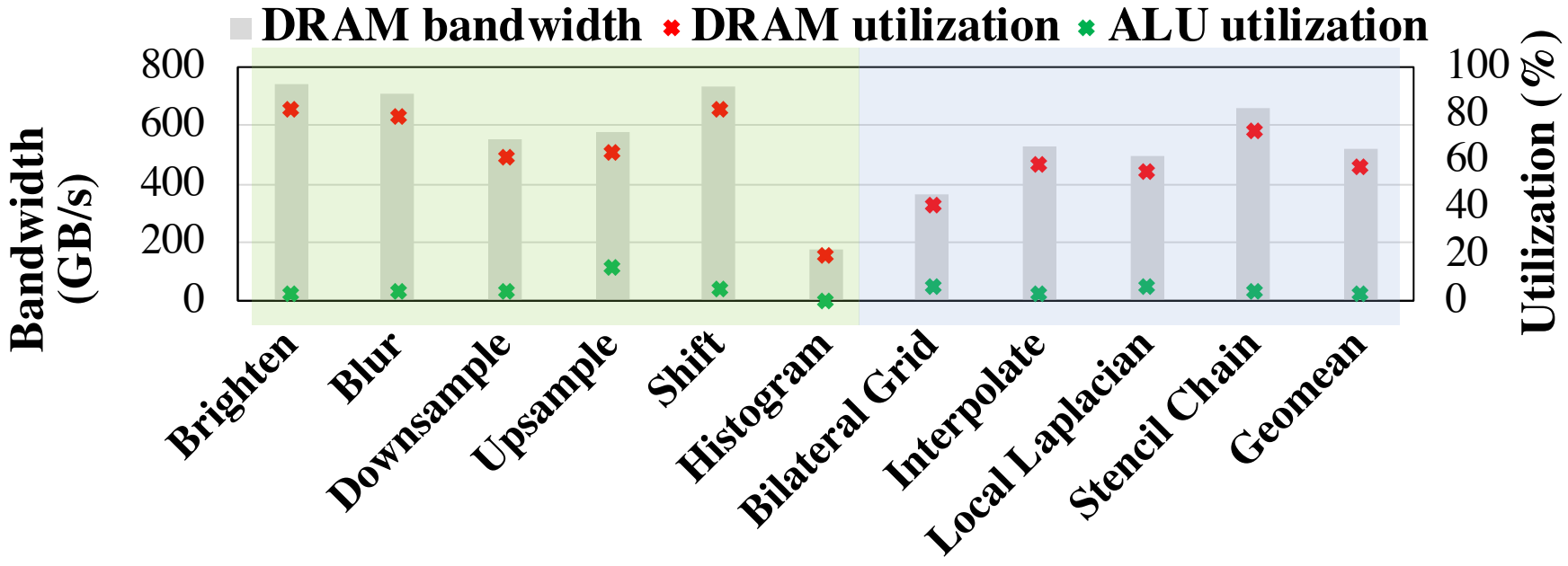
→

Multi-stage kernels
ALU utilization: **4.53%**
DRAM utilization: **55.73%**

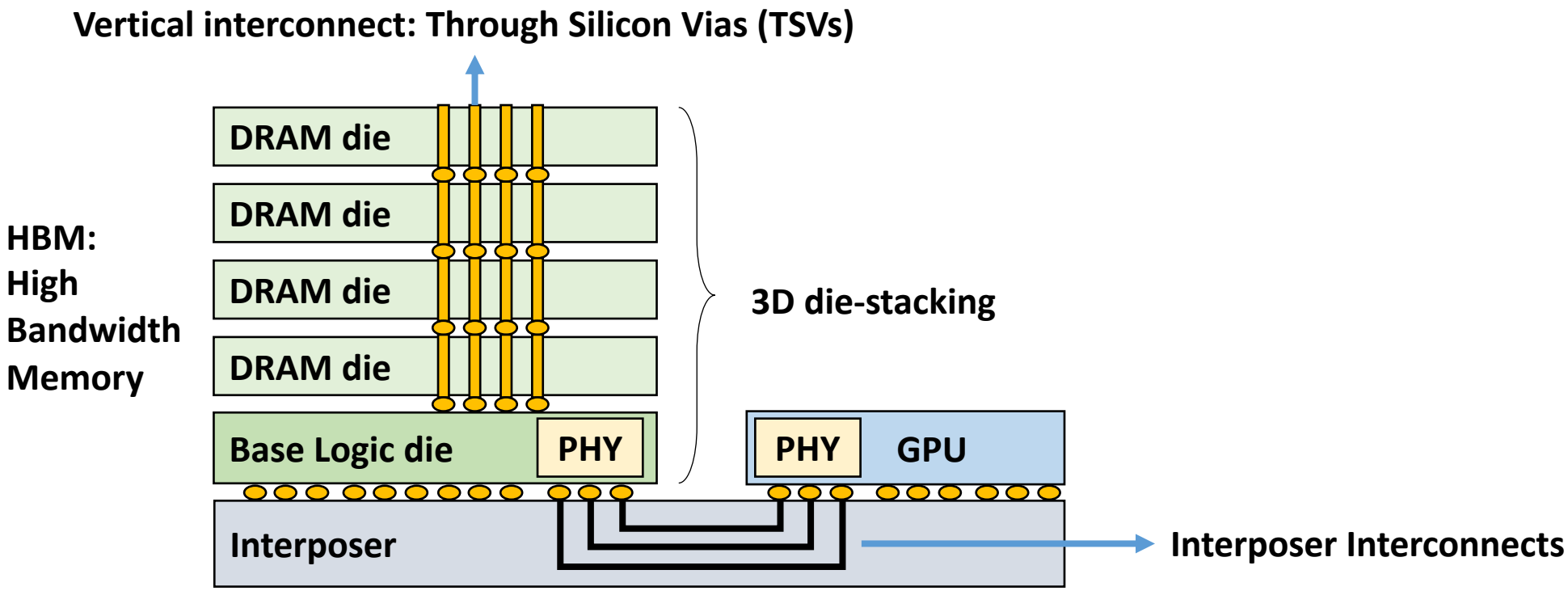


Motivation Data: Memory Bandwidth Bottleneck

Pipeline optimization does not change memory-bound behavior of image processing workloads on GPU

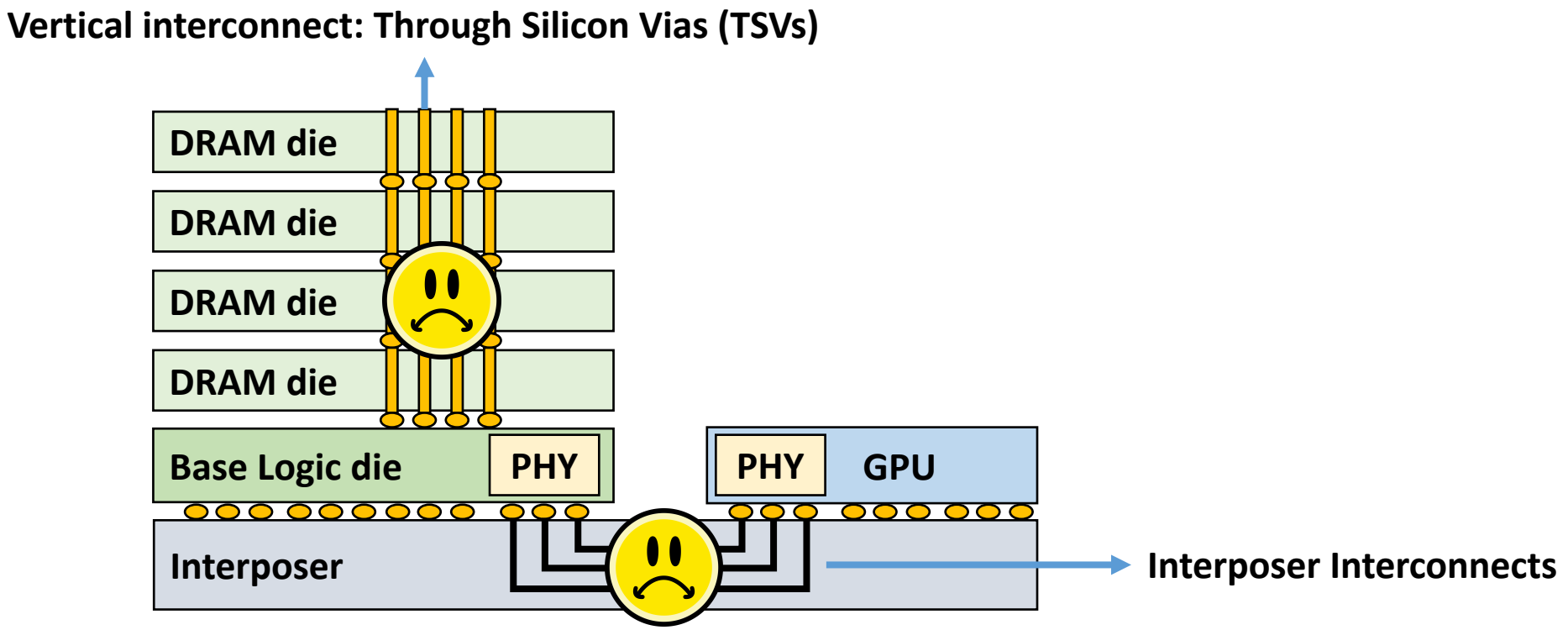


GPU: Bandwidth Scaling Challenge



GPU provides the highest memory bandwidth: High Bandwidth Memory (HBM) provides large bandwidth by 3D die-stacking technology

GPU: Bandwidth Scaling Challenge

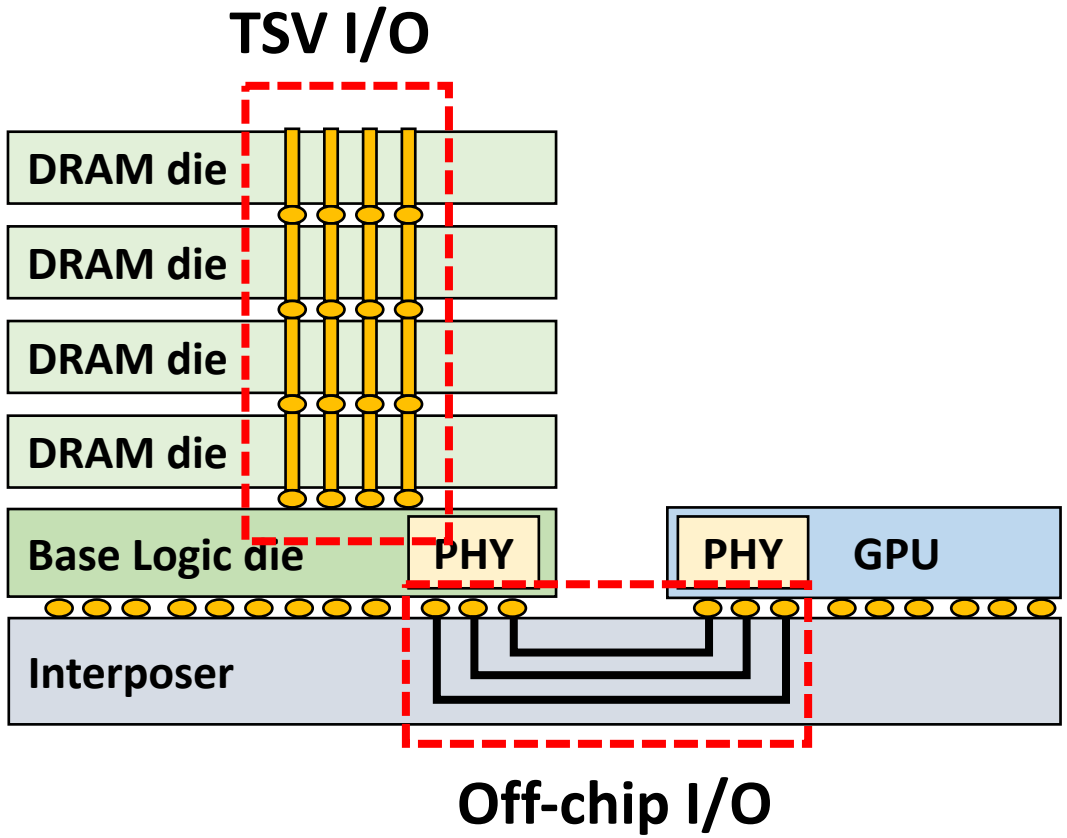


Memory bandwidth wall: memory bandwidth cannot scale with computation throughput

- Off-chip I/O (Interposer Interconnects)
- TSV I/O

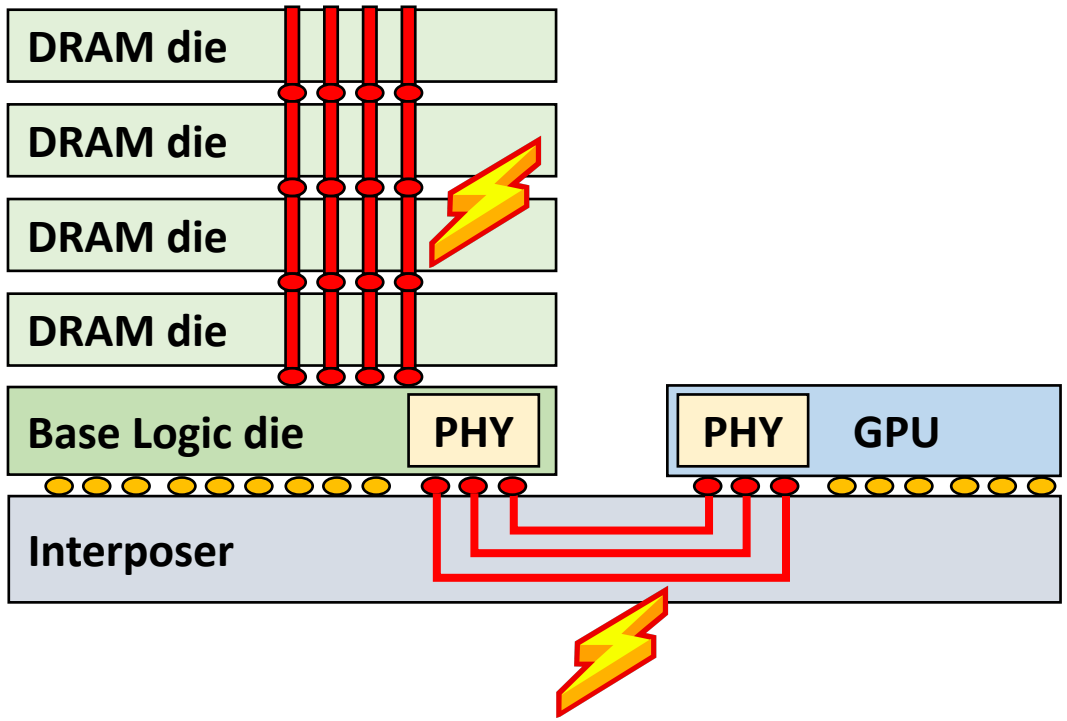
GPU: Bandwidth Scaling Challenge

- Raw memory bandwidth =
  (the number of I/Os) X (data rate)
  - Increasing the number of I/Os is difficult under tight area budget
    - Limited off-chip pins
    - TSVs already consumes ~18.8% DRAM die area for the current HBM2



GPU: Bandwidth Scaling Challenge

- Raw memory bandwidth =
  (the number of I/Os) X (data rate)
  - Increasing data rate will have signal integrity issues and increase power consumption as well



Summary

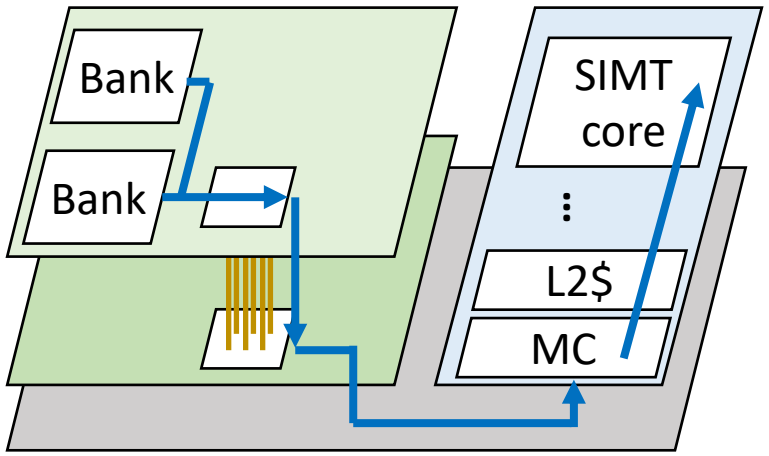
- Image processing is important in many application domains
- GPU suffers from memory bandwidth bottleneck:
  - Software: image processing pipelines are wide and heterogeneous
  - Hardware: GPU has bandwidth scaling challenges

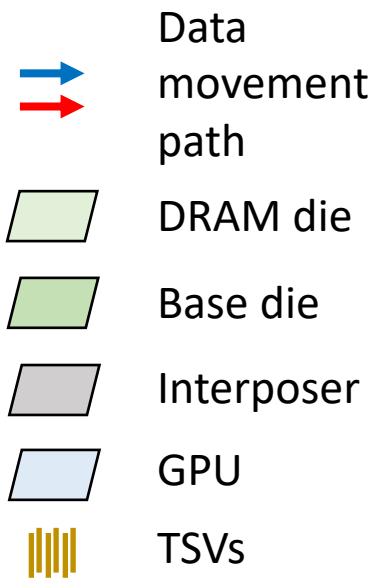
How to design a programmable image processing accelerator to provide more memory bandwidth?

3D-Stacking Processing-in-memory (PIM) Architecture

- Key idea:
  - Integrate computation logic **closer to** physical memory in order to increase memory bandwidth and reduce data movement energy

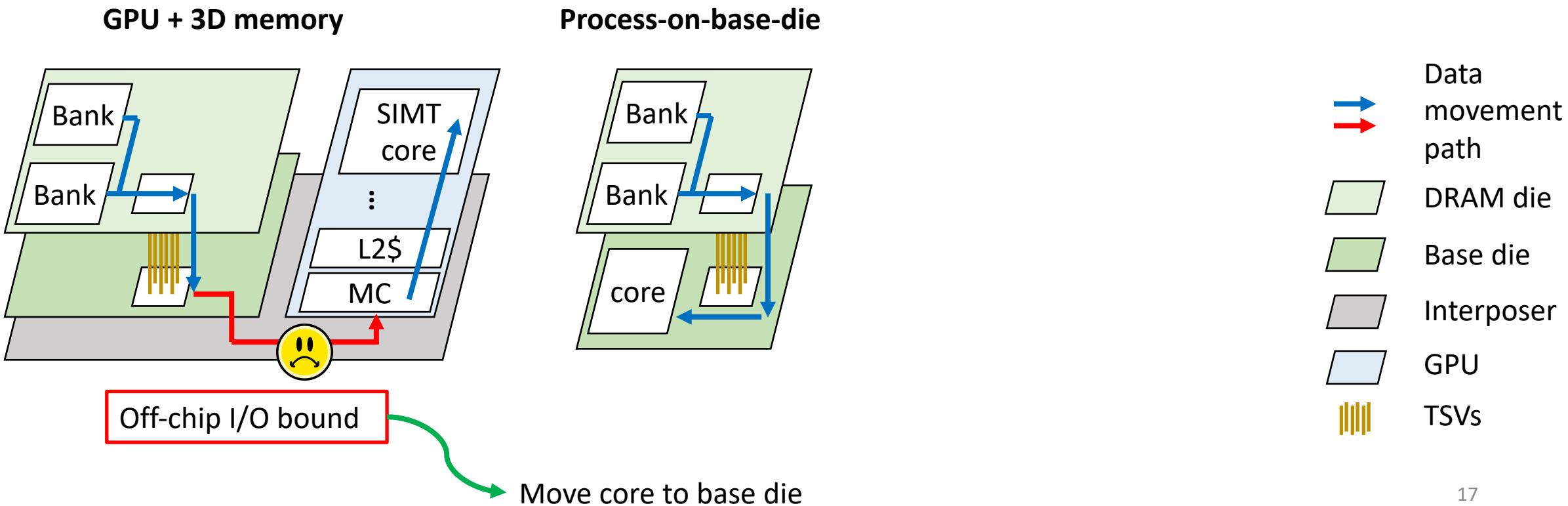
GPU + 3D memory





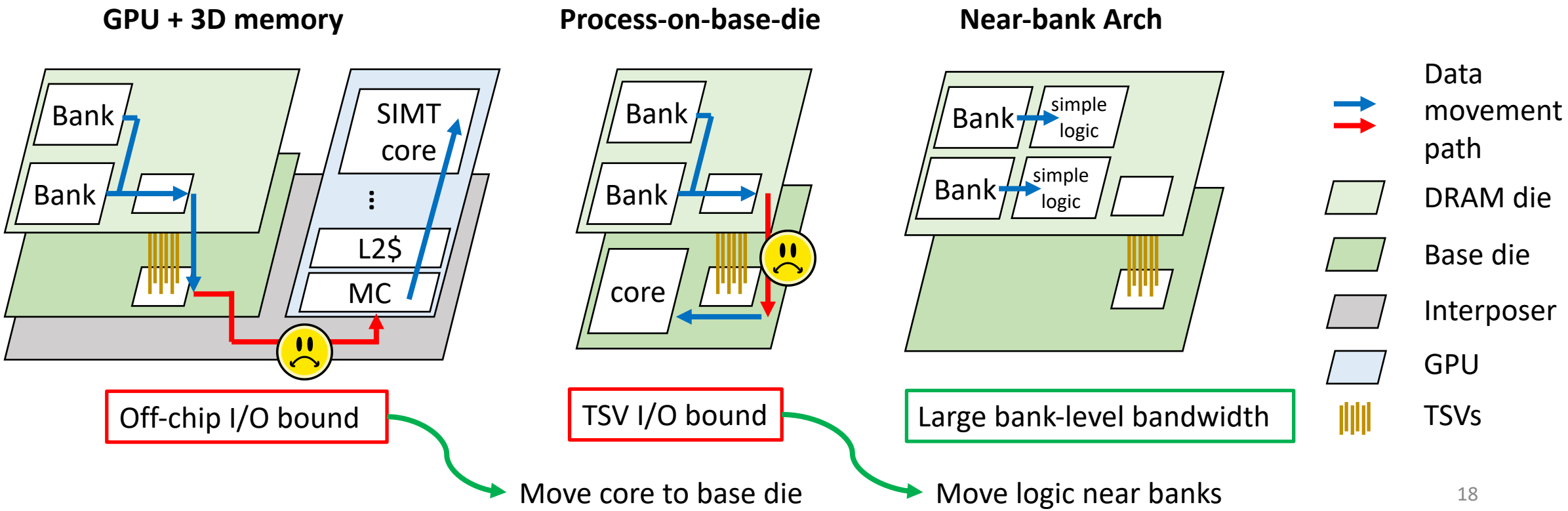
3D-Stacking Processing-in-memory (PIM) Architecture

- Key idea:
  - Integrate computation logic **closer to** physical memory in order to increase memory bandwidth and reduce data movement energy



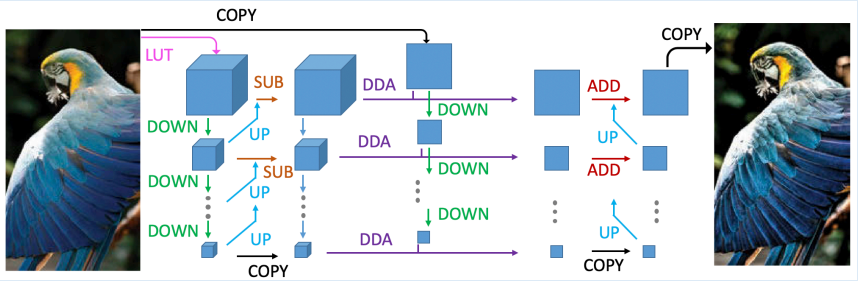
3D-Stacking Processing-in-memory (PIM) Architecture

- Key idea:
  - Integrate computation logic **closer to** physical memory in order to increase memory bandwidth and reduce data movement energy



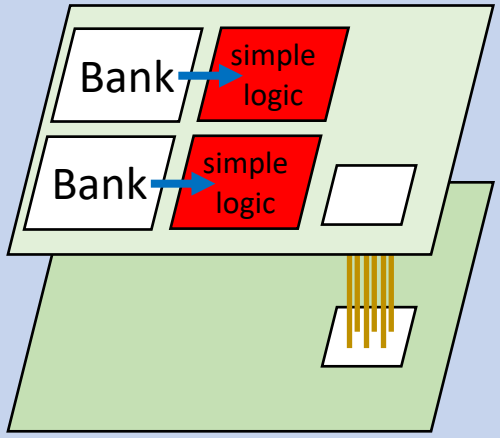
Challenges for near-bank architecture

Application



Complex computation and memory access patterns

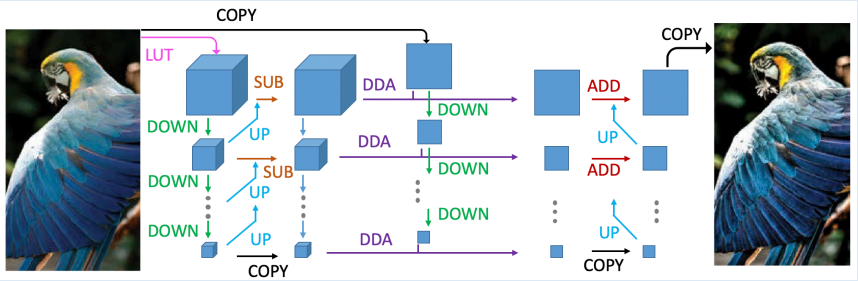
Hardware



Resource constraints

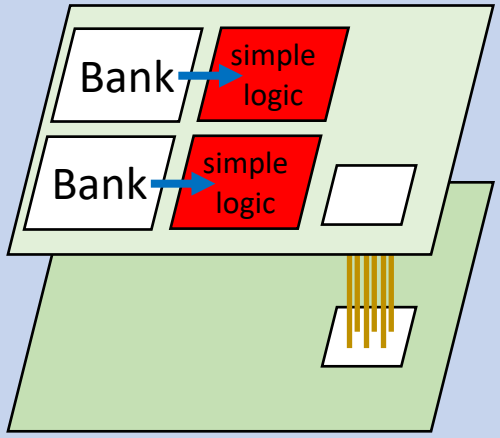
Challenges for near-bank architecture

Application



Complex computation and memory access patterns

Hardware

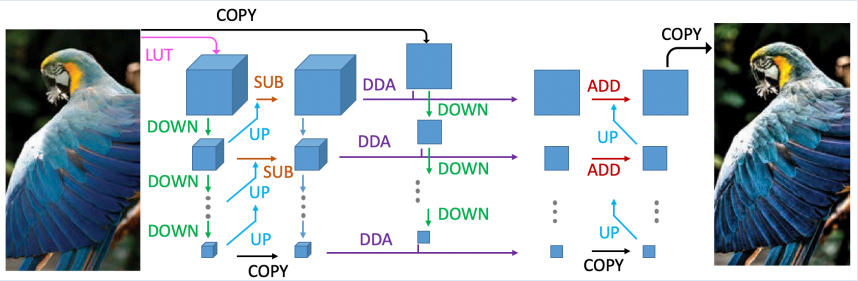


Resource constraints

Lightweight programmable architecture for image processing domain

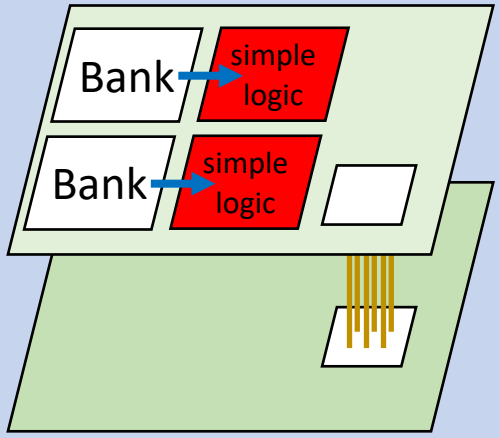
Challenges for near-bank architecture

Application



Complex computation and memory access patterns

Hardware



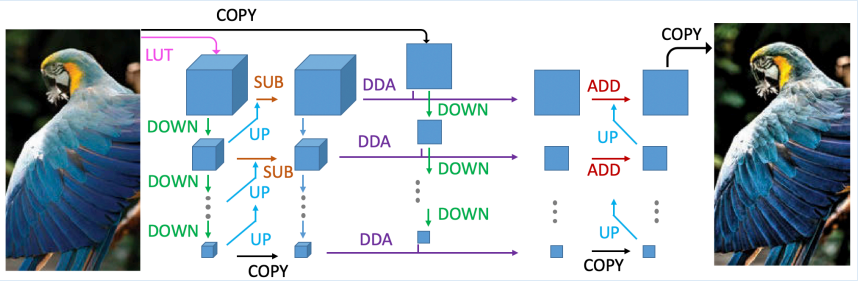
Resource constraints

Concise yet powerful Instruction Set Architecture (ISA)

Lightweight programmable architecture for image processing domain

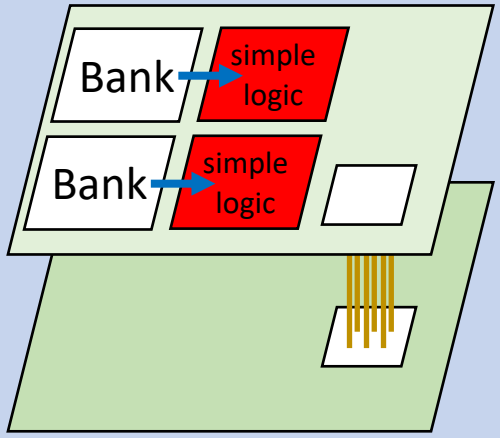
Challenges for near-bank architecture

Application



Complex computation and memory access patterns

Hardware



Resource constraints

End-to-end software support:

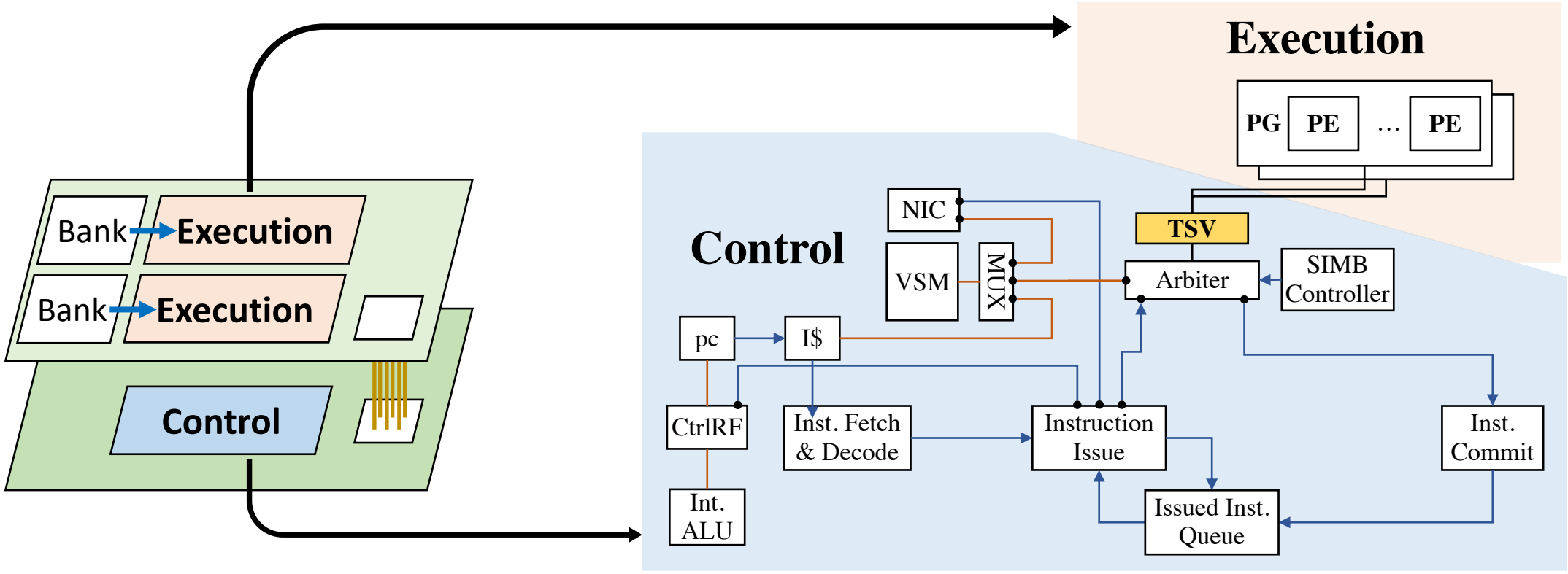
- Programming interface
- Compiler optimization

Concise yet powerful Instruction Set Architecture (ISA)

Lightweight programmable architecture for image processing domain

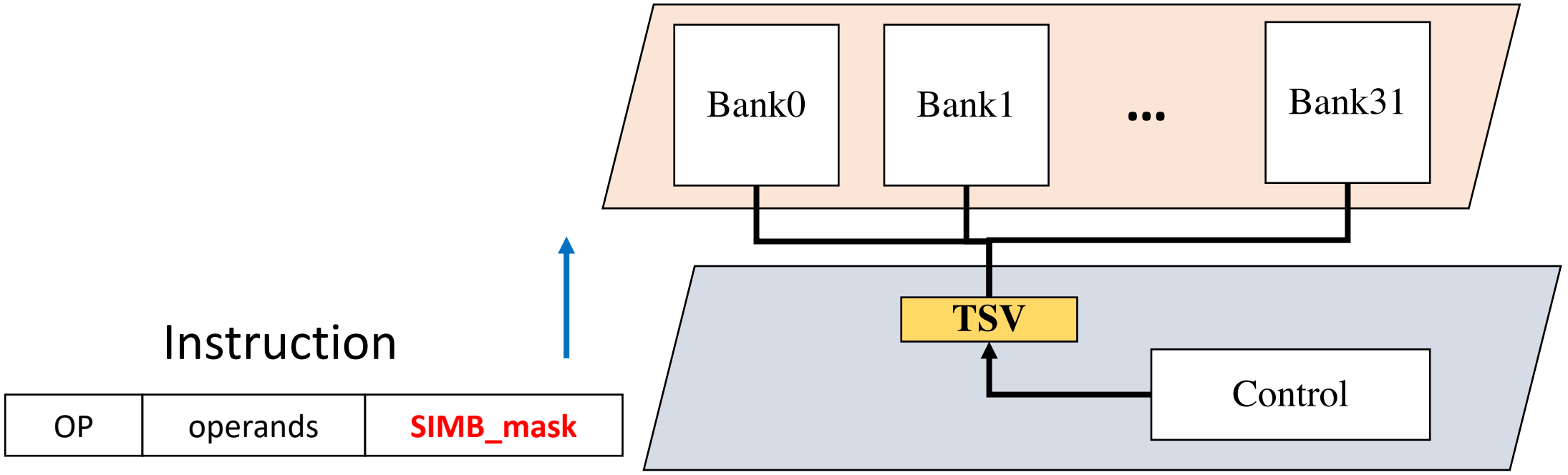
Key Contributions

- **iPIM**: A decoupled control-execution architecture



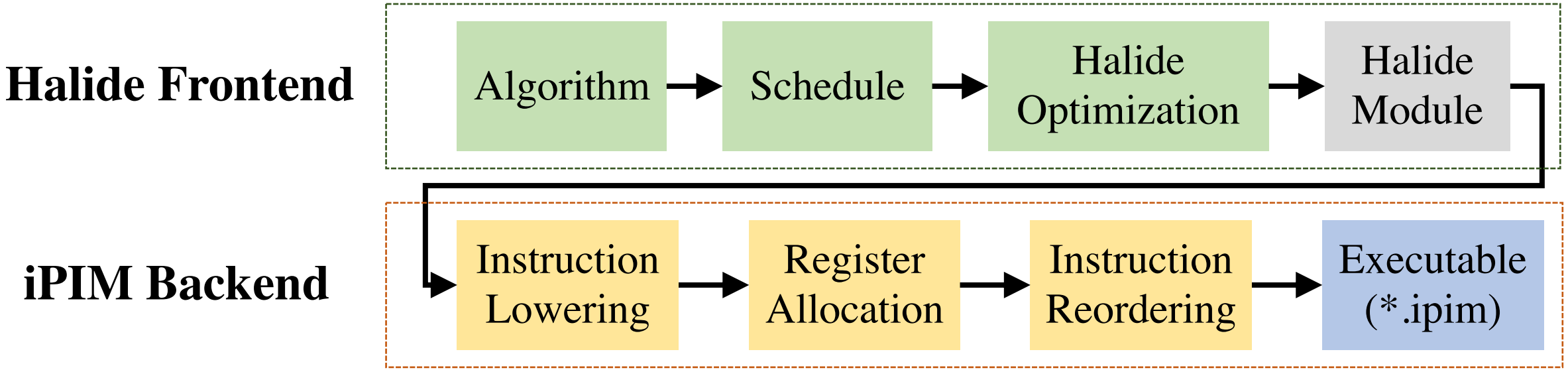
Key Contributions

- Lightweight programmable arch: **A decoupled control-execution architecture**
- Flexible ISA support: **Single-Instruction-Multiple-Bank (SIMB) ISA**



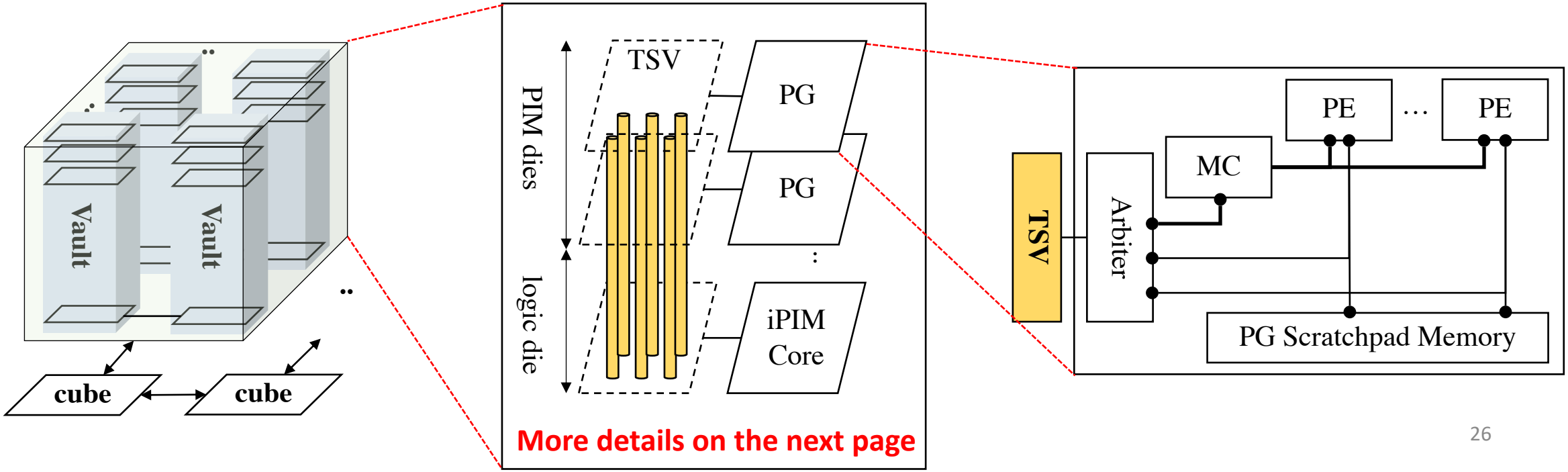
Key Contributions

- Lightweight programmable arch: **A decoupled control-execution architecture**
- Flexible ISA support: **Single-Instruction-Multiple-Bank (SIMB) ISA**
- End-to-end compilation flow: **Halide-iPIM**



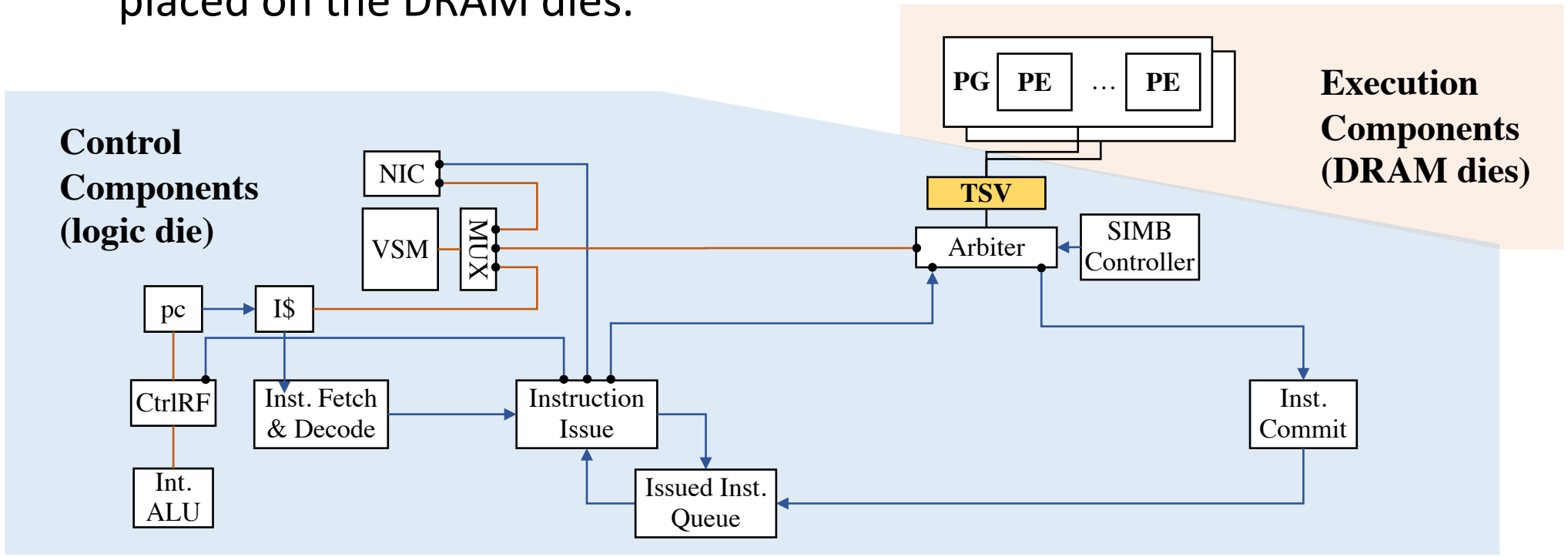
iPIM: High-level Arch Design Overview

- 3D-stacking, near-bank processing-in-memory architecture
- Hierarchical design with good scalability
  - **Cube – Vault – Process Group (PG) – Process Engine (PE)**
  - A Process Engine (PE) contains a DRAM bank and simple logic components



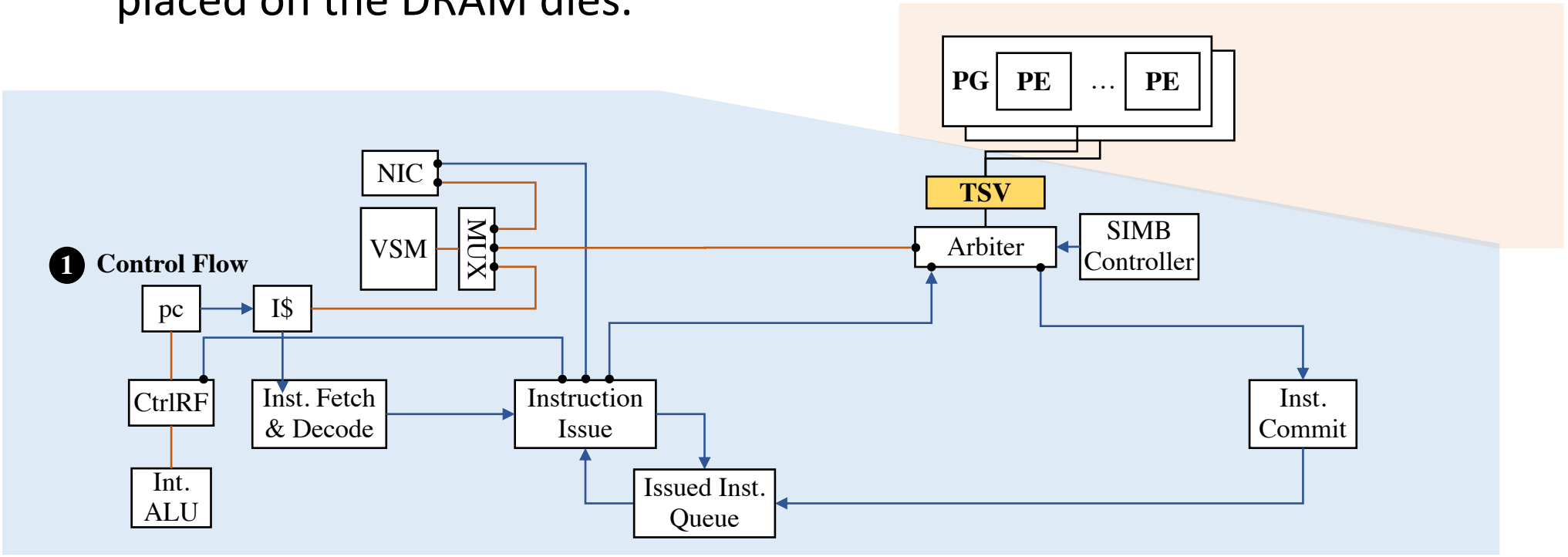
Vault Architecture

- Key idea: **Decoupled Control-Execution Architecture**
  - Front-end (complex logic) **control components** of the core are placed on the base logic die;
  - Back-end (simple logic and memory-intensive) **execution components** are placed on the DRAM dies.



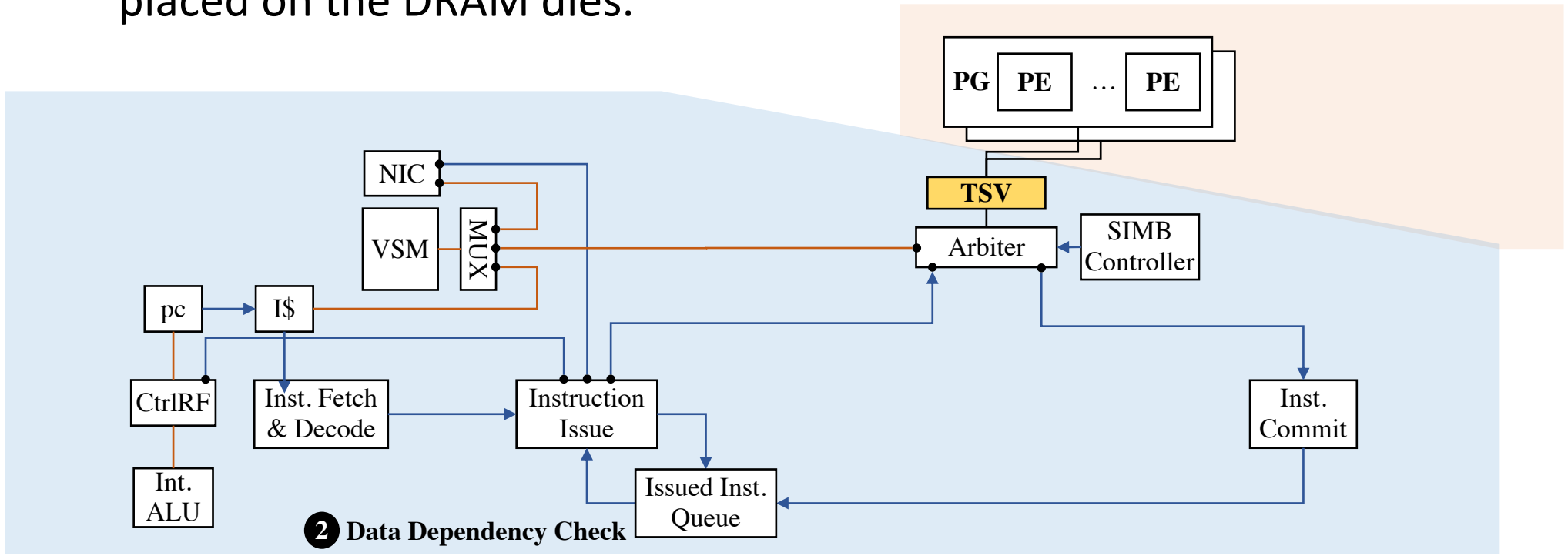
Vault Architecture

- Key idea: **Decoupled Control-Execution Architecture**
  - Front-end (complex logic) control components of the core are placed on the base logic die;
  - Back-end (simple logic and memory-intensive) execution components are placed on the DRAM dies.



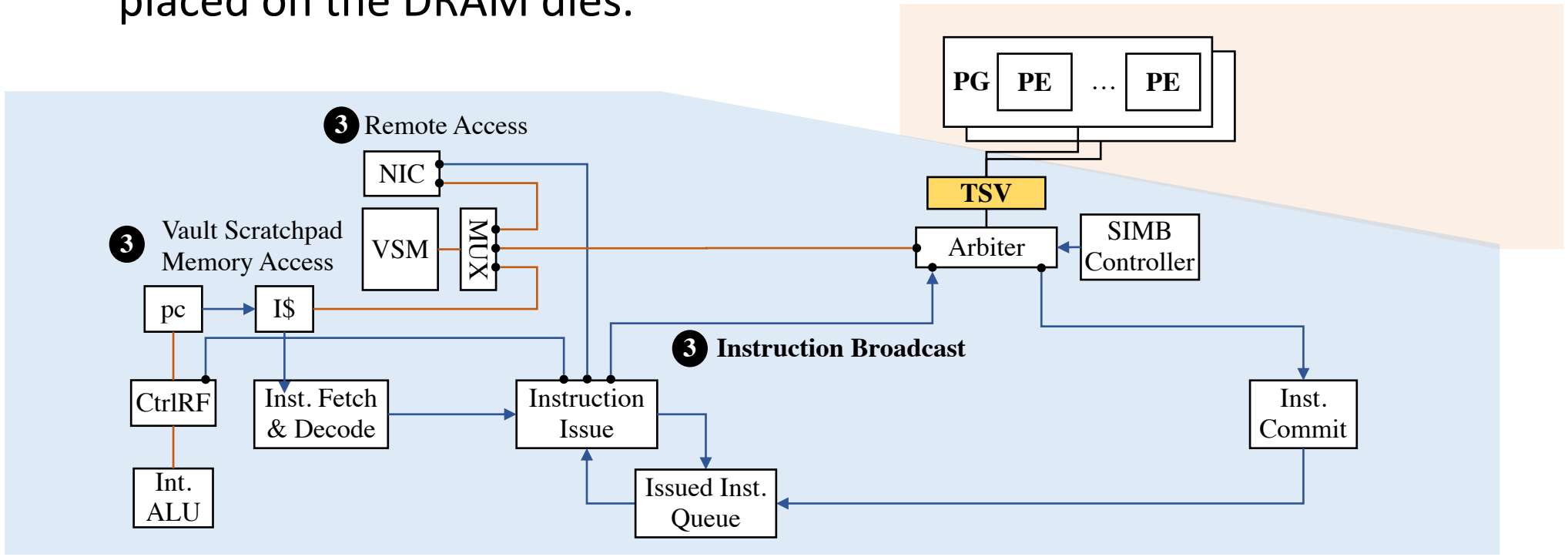
Vault Architecture

- Key idea: **Decoupled Control-Execution Architecture**
  - Front-end (complex logic) control components of the core are placed on the base logic die;
  - Back-end (simple logic and memory-intensive) execution components are placed on the DRAM dies.



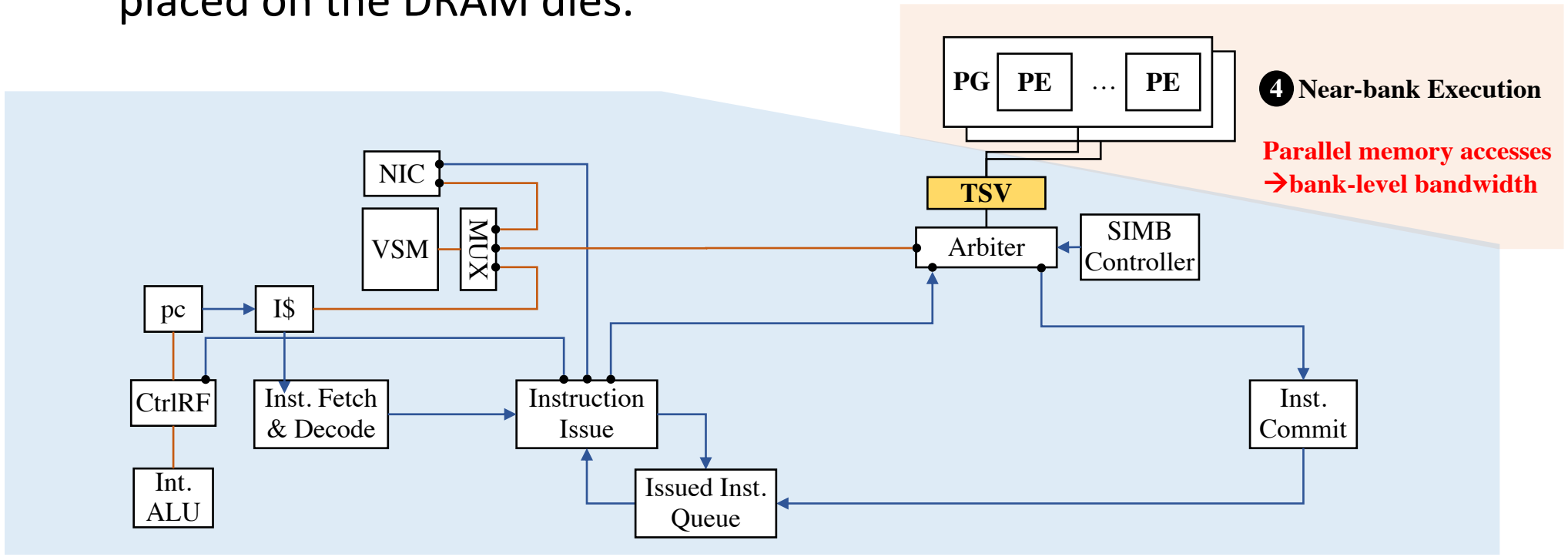
Vault Architecture

- Key idea: **Decoupled Control-Execution Architecture**
  - Front-end (complex logic) control components of the core are placed on the base logic die;
  - Back-end (simple logic and memory-intensive) execution components are placed on the DRAM dies.



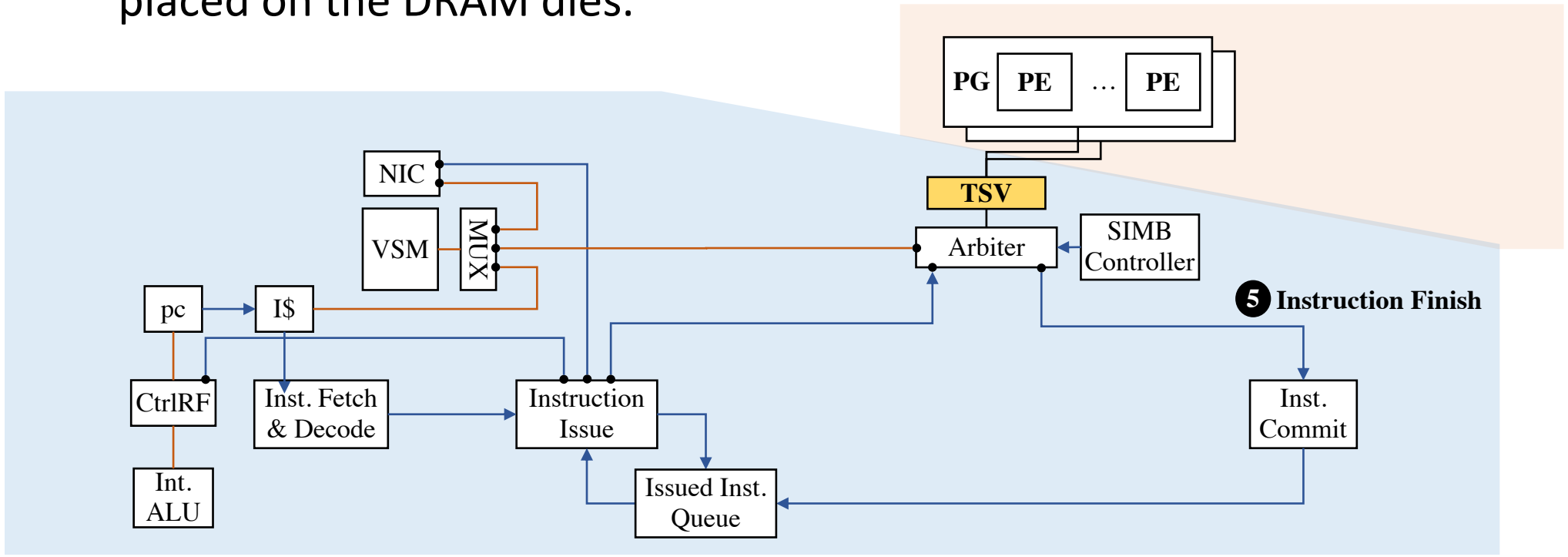
Vault Architecture

- Key idea: **Decoupled Control-Execution Architecture**
  - Front-end (complex logic) control components of the core are placed on the base logic die;
  - Back-end (simple logic and memory-intensive) execution components are placed on the DRAM dies.



Vault Architecture

- Key idea: **Decoupled Control-Execution Architecture**
  - Front-end (complex logic) control components of the core are placed on the base logic die;
  - Back-end (simple logic and memory-intensive) execution components are placed on the DRAM dies.

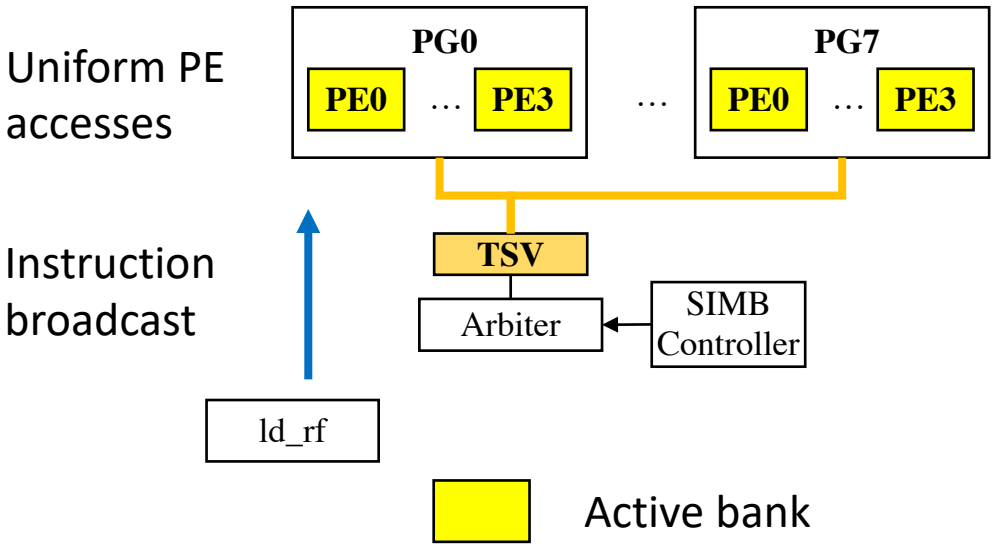


Single-Instruction-Multiple-Bank (SIMB) ISA

- Enable massive bank-level concurrent execution to exploit data parallelism

Example: load data from the DRAM bank to the local data register file (DataRF)

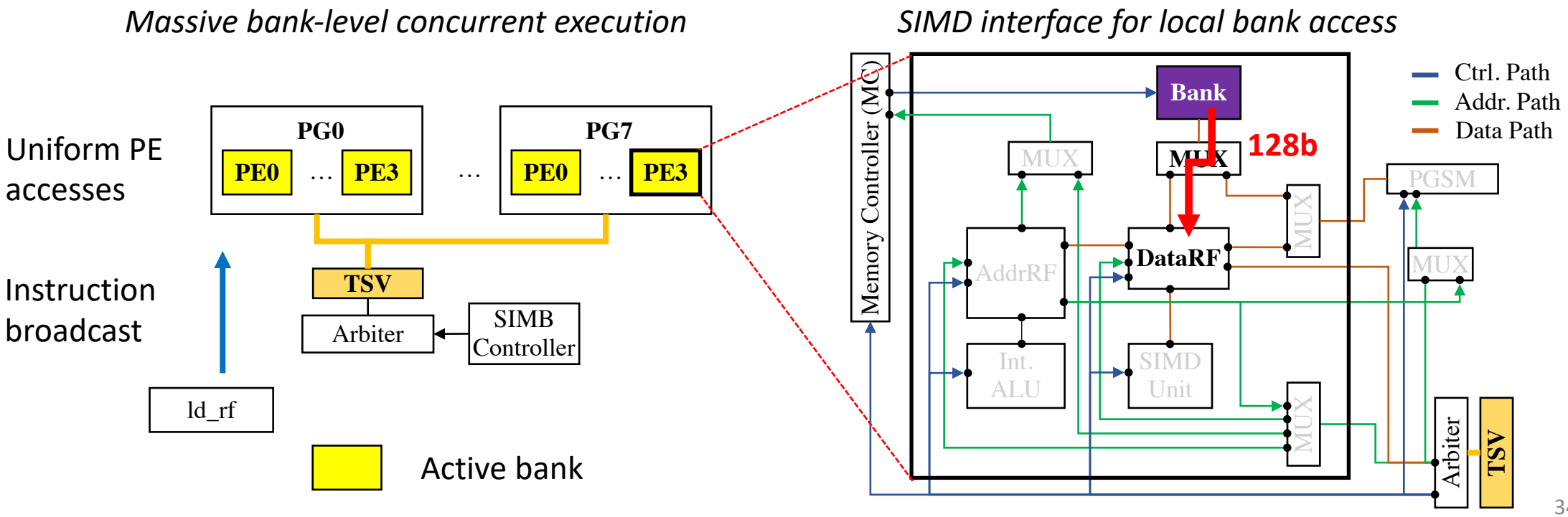
Massive bank-level concurrent execution



Single-Instruction-Multiple-Bank (SIMB) ISA

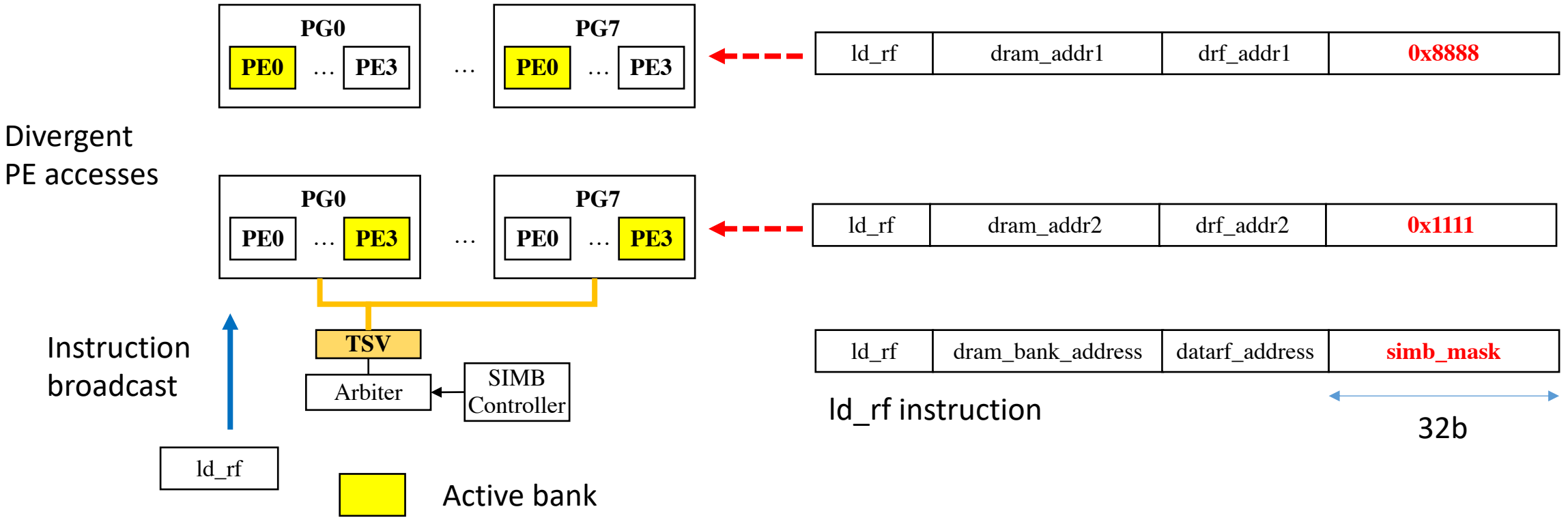
- Enable massive bank-level concurrent execution to exploit data parallelism
- SIMD interface to exploit abundant bank-level bandwidth

Example: load data from the DRAM bank to the local data register file (DataRF)



Single-Instruction-Multiple-Bank (SIMB) ISA

- Predicate execution (*simb_mask*) to allow divergent bank-level accesses / computations

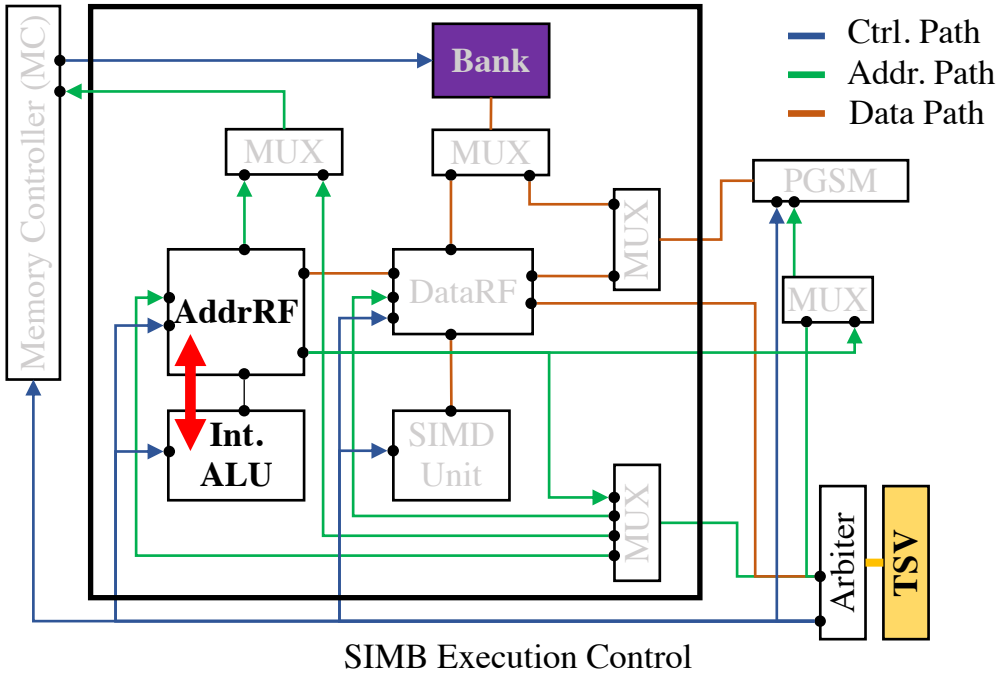


Single-Instruction-Multiple-Bank (SIMB) ISA

- Support indirect addressing in image processing domain

Example: access pixel[xi, offset+yi]

cal_arf	op	dst_arf	src_arf1	src_arf2	simb_mask



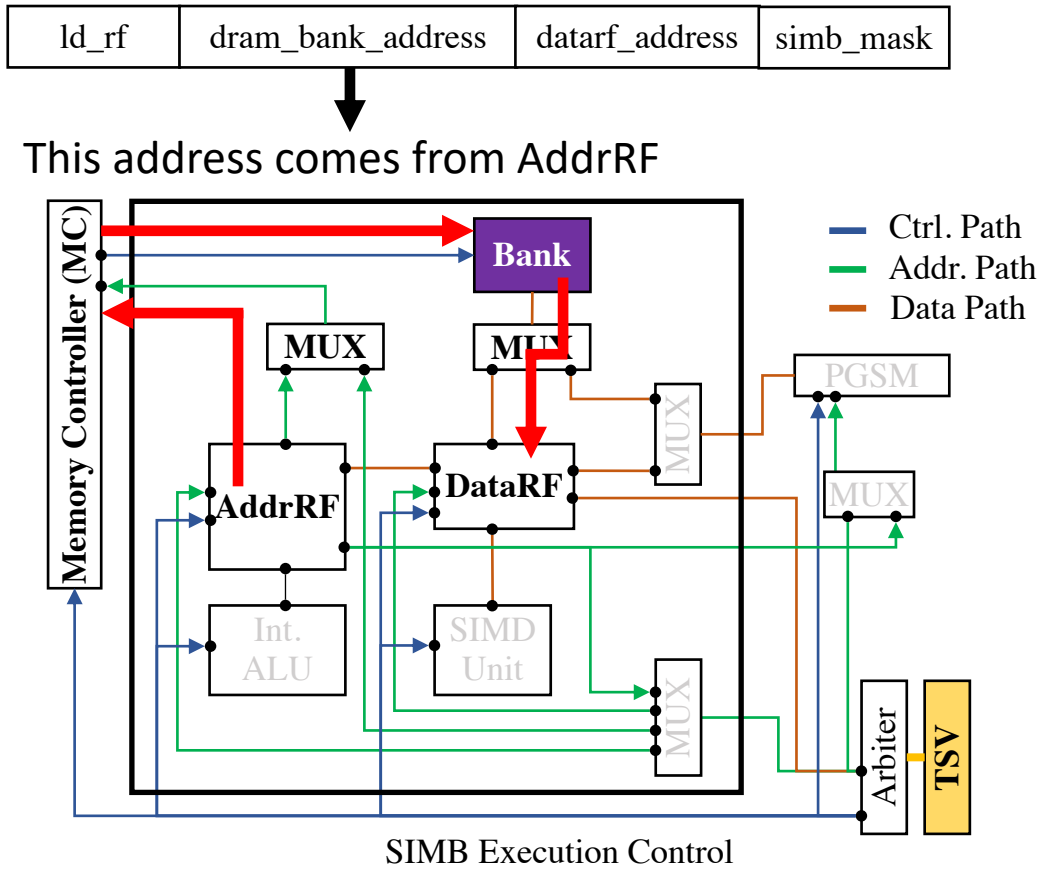
SIMB Execution Control

Perform:
dram_address = xi + (offset+yi)*img_width

Single-Instruction-Multiple-Bank (SIMB) ISA

- Support indirect addressing in image processing domain

Example: access pixel[xi, offset+yi]



Access local DRAM bank using dram_address stored in AddrRF

Single-Instruction-Multiple-Bank (SIMB) ISA

- SIMB ISA also supports:
  - Data-dependent calculation
    - mov_drf / mov_arf
  - Remote data access from different vaults / cubes
    - rd_vsm / wr_vsm / req
  - Control flow instructions
    - jump / cjump / calc_crf / seti_crf
  - Synchronization mechanism
    - Sync
- Please refer to the paper for more details

End-to-end compilation support: **Halide-iPIM**

- Halide
  - A domain specific programming language and toolchain for image processing
  - It decouples the **algorithm descriptions** and the **schedules to hardware mapping**

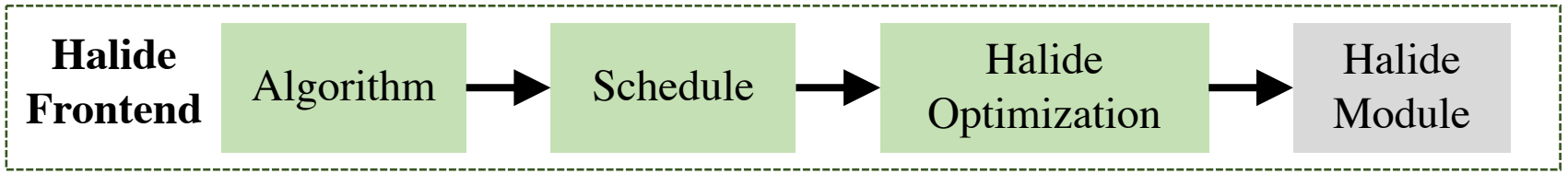
Example: image blur

```
// Algorithm
Func blurx(x, y) = (in(x - 1, y) + in(x, y)
                + in(x + 1, y)) / 3.0f;
Func out(x, y) = (blurx(x, y - 1) + blurx(x, y)
                + blurx(x, y + 1)) / 3.0f;

// Schedule for iPIM
out.compute_root()
   .ipim_tile(x, y, xi, yi, 8, 8)
   .load_pgsm(xi, yi)
   .vectorize(xi, 4);
```

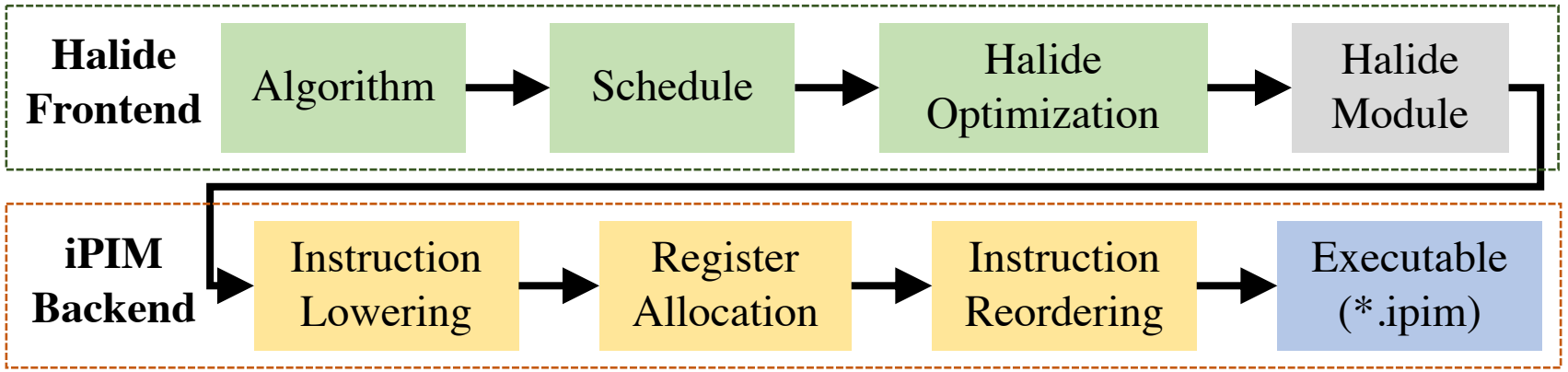
End-to-end compilation support: **Halide-iPIM**

- Halide
  - An domain specific programming language and toolchain for image processing
  - It decouples the algorithm descriptions and the schedules to hardware mapping
- Halide-iPIM
  - We extend Halide frontend to support customized schedules for iPIM
  - We leverage existing Halide schedules for pipeline fusion and vectorization on iPIM



End-to-end compilation support: **Halide-iPIM**

- Halide
  - An domain specific programming language and toolchain for image processing
  - It decouples the algorithm descriptions and the schedules to hardware mapping
- Halide-iPIM
  - We extend Halide frontend to support customized schedules for iPIM
  - We leverage existing Halide schedules for pipeline fusion and vectorization on iPIM
  - We develop three backend optimizations for iPIM



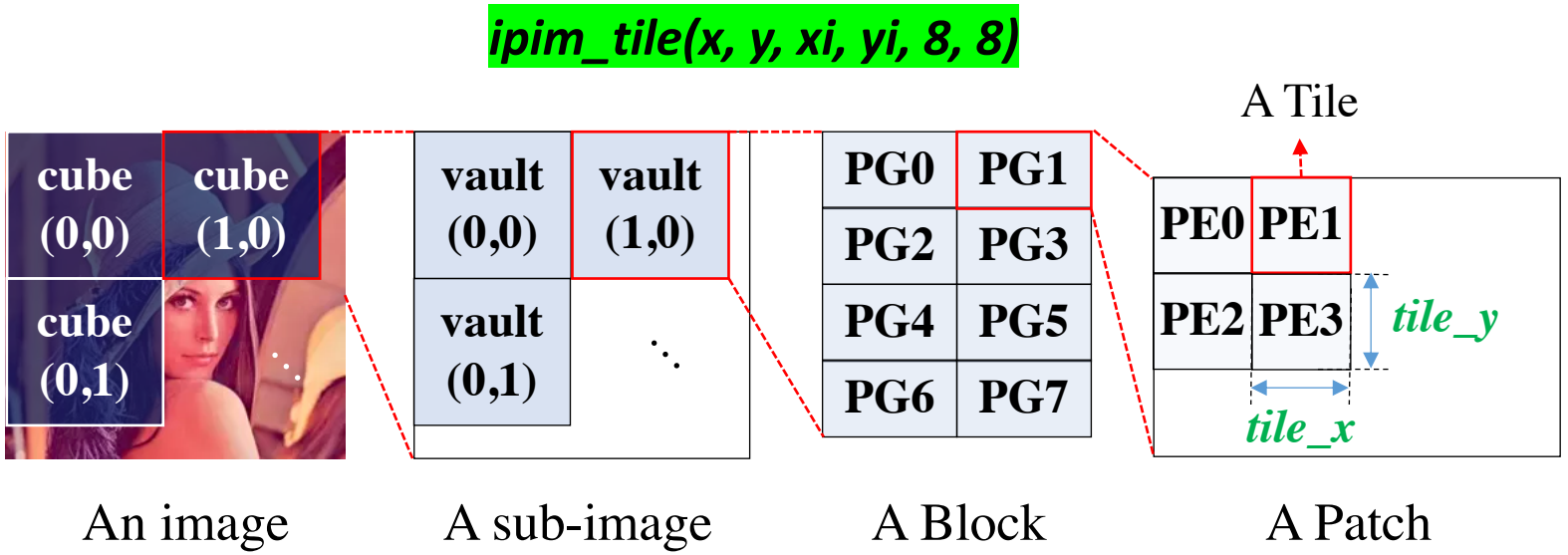
Halide-iPIM: Frontend Schedules

- Two new schedule primitives:
  - *ipim_tile()*
    - distribute data into different banks

Example: image blur

```
// Algorithm
Func blurx(x, y) = (in(x - 1, y) + in(x, y)
                 + in(x + 1, y)) / 3.0f;
Func out(x, y) = (blurx(x, y - 1) + blurx(x, y)
                 + blurx(x, y + 1)) / 3.0f;

// Schedule for iPIM
out.compute_root()
   .ipim_tile(x, y, xi, yi, 8, 8)
   .load_pgsm(xi, yi)
   .vectorize(xi, 4);
```



Halide-iPIM: Frontend Schedules

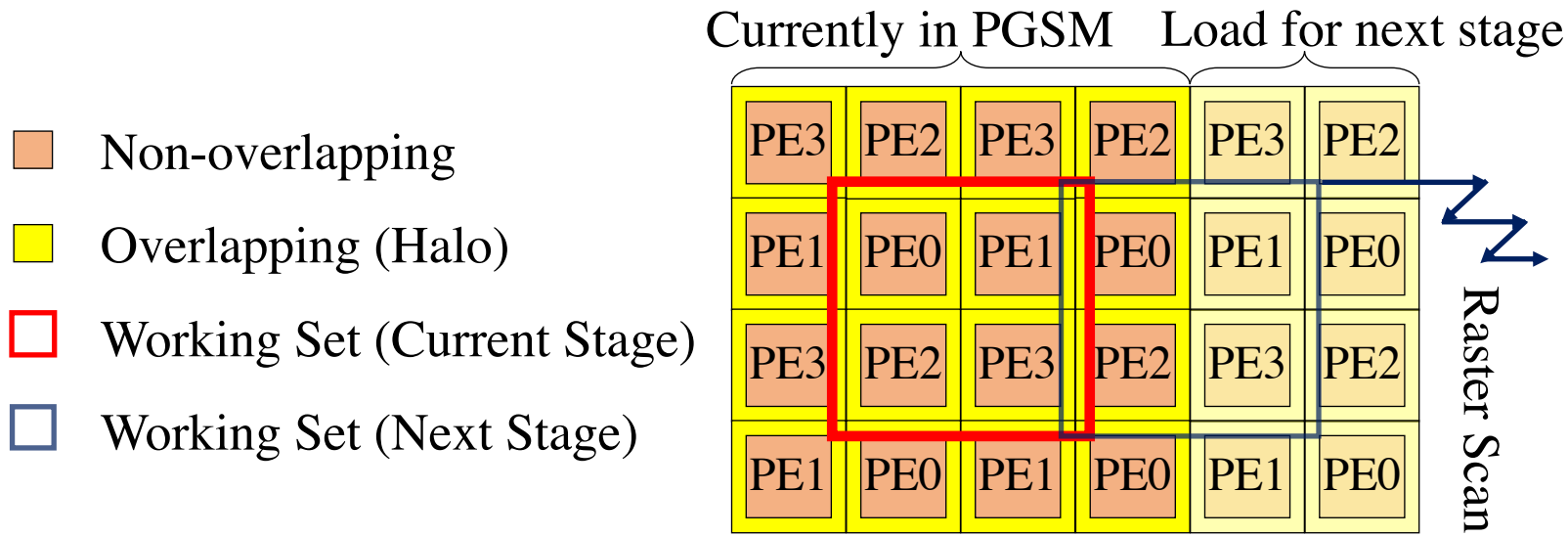
- Two new schedule primitives:
  - *load_pgsm()*
    - Utilize the scratchpad of a processing group (PG)

Example: image blur

```
// Algorithm
Func blurx(x, y) = (in(x - 1, y) + in(x, y)
            + in(x + 1, y)) / 3.0f;
Func out(x, y) = (blurx(x, y - 1) + blurx(x, y)
            + blurx(x, y + 1)) / 3.0f;

// Schedule for iPIM
out.compute_root()
   .ipim_tile(x, y, xi, yi, 8, 8)
   .load_pgsm(xi, yi)
   .vectorize(xi, 4);
```

load_pgsm(xi, yi)



Halide-iPIM: Frontend Schedules

- Leverage existing schedule primitives:
  - *compute_root()*
    - Specify pipeline fusing
  - *vectorize()*
    - Align data to improve utilization of SIMD units

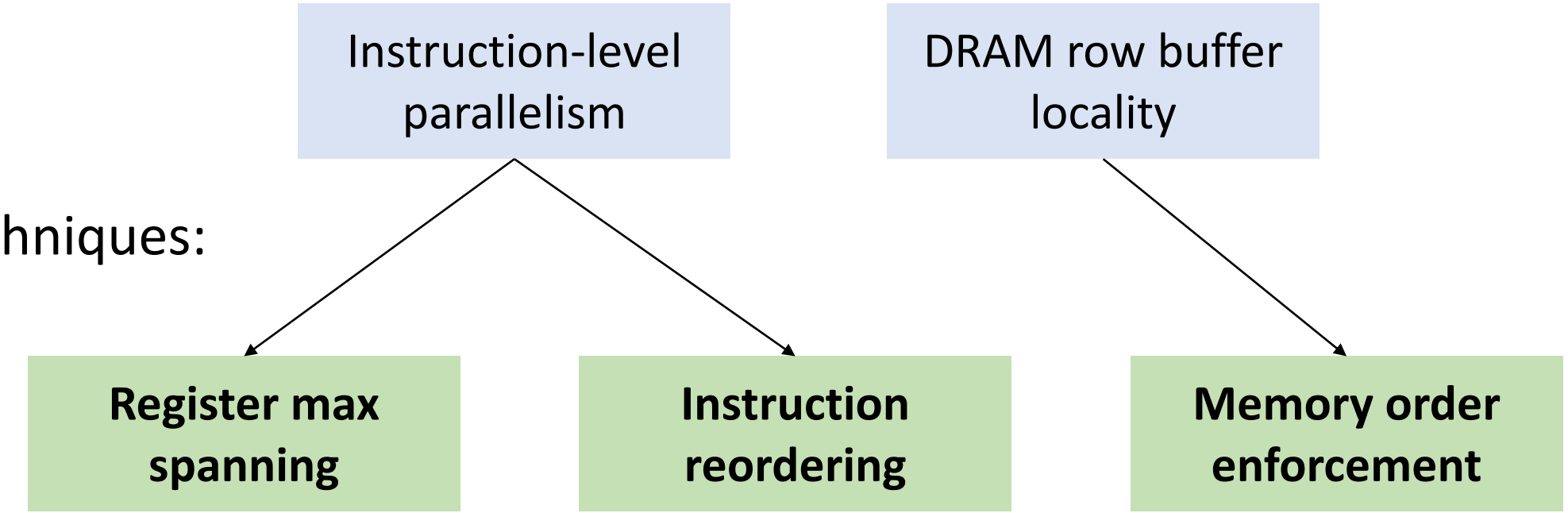
Example: image blur

```
// Algorithm
Func blurx(x, y) = (in(x - 1, y) + in(x, y)
                + in(x + 1, y)) / 3.0f;
Func out(x, y) = (blurx(x, y - 1) + blurx(x, y)
                + blurx(x, y + 1)) / 3.0f;

// Schedule for iPIM
out.compute_root()
   .ipim_tile(x, y, xi, yi, 8, 8)
   .load_pgsm(xi, yi)
   .vectorize(xi, 4);
```

Halide-iPIM: Backend Optimizations

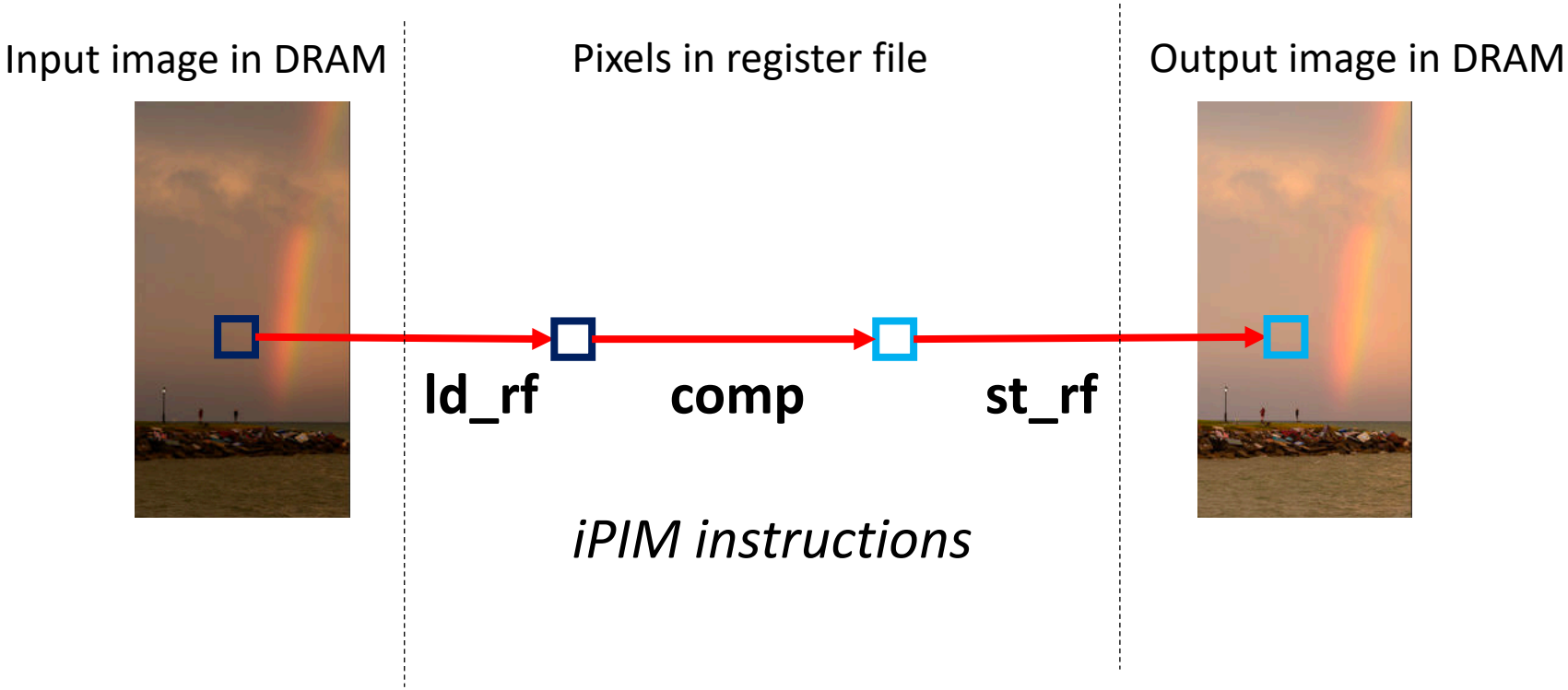
- Optimization objectives:



- Our techniques:

Halide-iPIM: Backend Optimizations

Example: **Image Brightening**

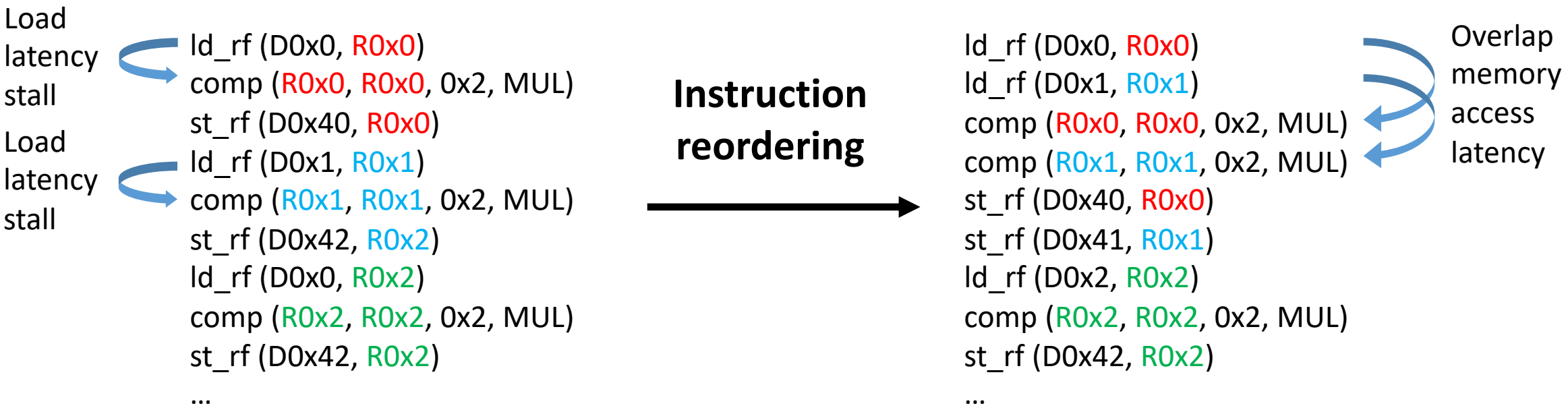


Halide-iPIM: Backend Optimizations

Example: **Image Brightening**

Halide-iPIM: Backend Optimizations

Example: **Image Brightening**



Halide-iPIM: Backend Optimizations

Example: **Image Brightening**

Disrupt row buffer locality

```
ld_rf (D0x0, R0x0)
ld_rf (D0x1, R0x1)
comp (R0x0, R0x0, 0x2, MUL)
comp (R0x1, R0x1, 0x2, MUL)
st_rf (D0x40, R0x0)
st_rf (D0x41, R0x1)
ld_rf (D0x2, R0x2)
comp (R0x2, R0x2, 0x2, MUL)
st_rf (D0x42, R0x2)
…
```

Memory order enforcement

```
ld_rf (D0x0, R0x0)
ld_rf (D0x1, R0x1)
ld_rf (D0x2, R0x2)
…
comp (R0x0, R0x0, 0x2, MUL)
comp (R0x1, R0x1, 0x2, MUL)
comp (R0x2, R0x2, 0x2, MUL)
…
st_rf (D0x40, R0x0)
st_rf (D0x41, R0x1)
st_rf (D0x42, R0x2)
…
```

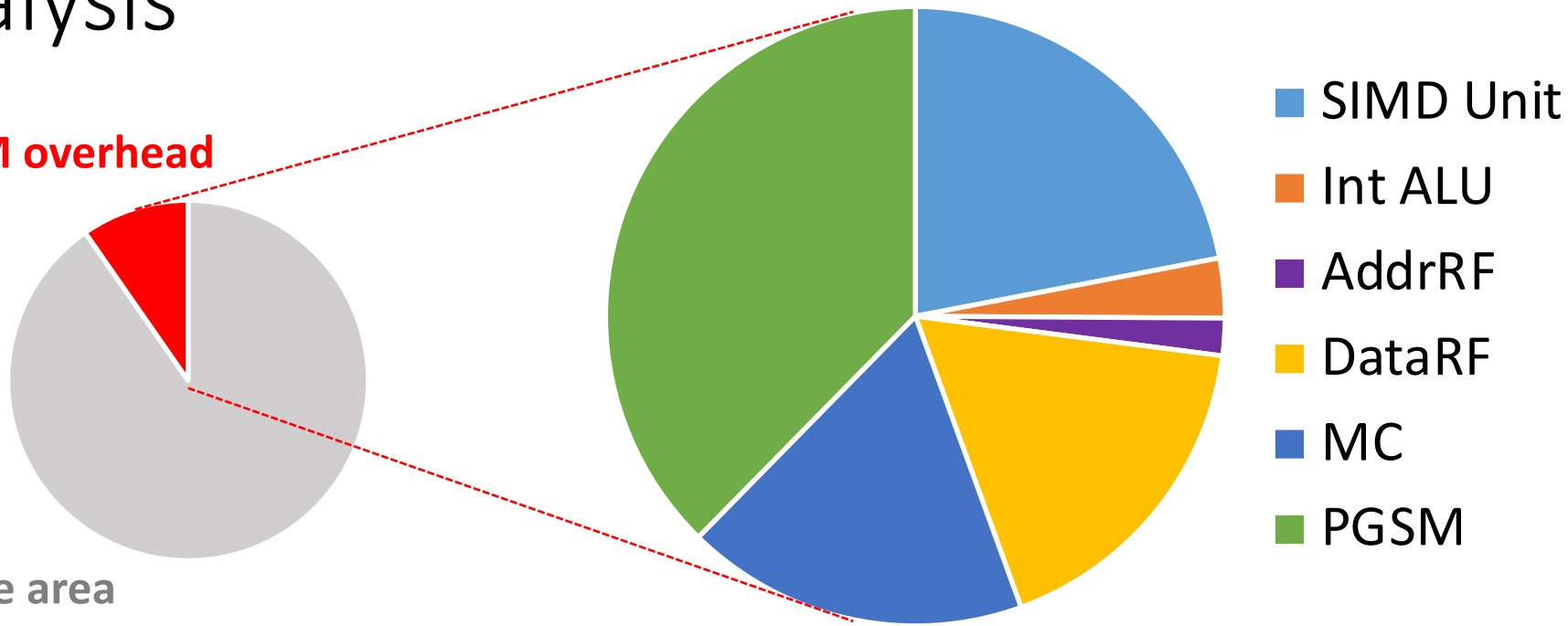
All loads to the same row buffer

Enforce memory order

All stores to the same row buffer

Evaluations

Area Analysis

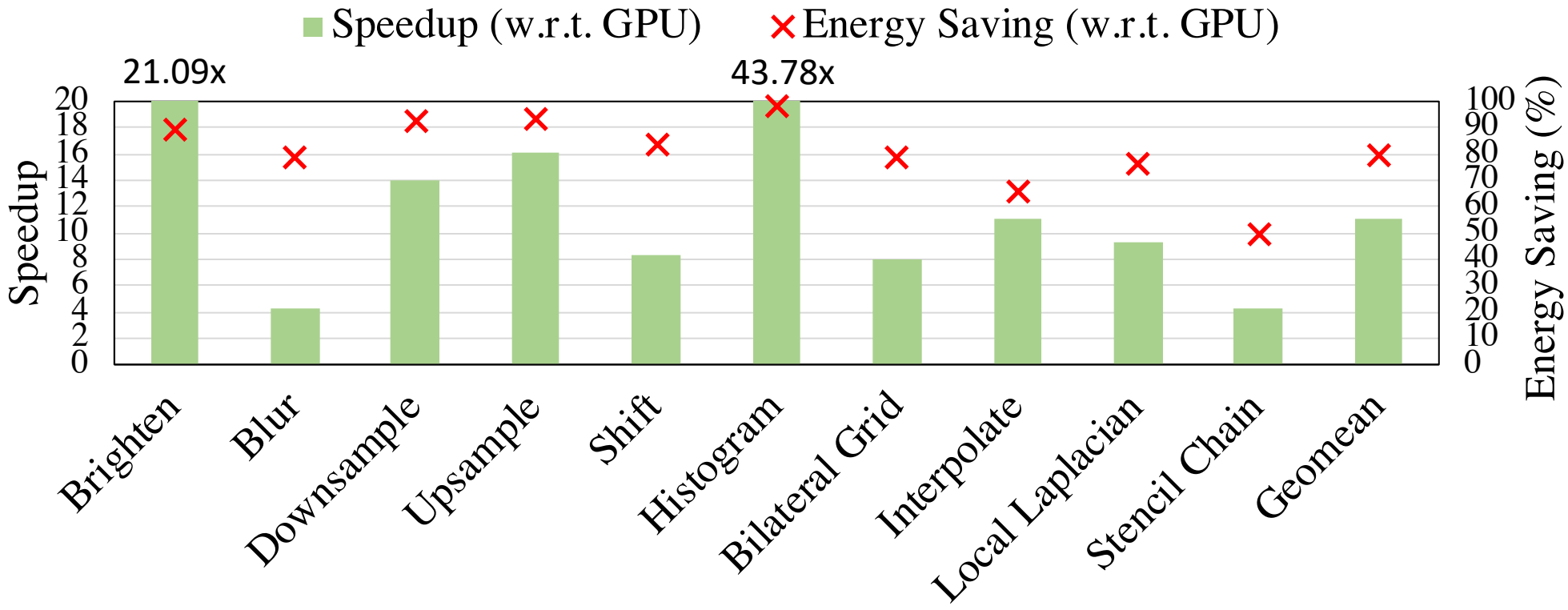


Area overhead of added components per DRAM die: 10.71%

- Conservatively assume 2x area overhead in DRAM process

Area of control logic on base die: 0.92mm^2 (fits in 3.5mm^2 extra area per vault)

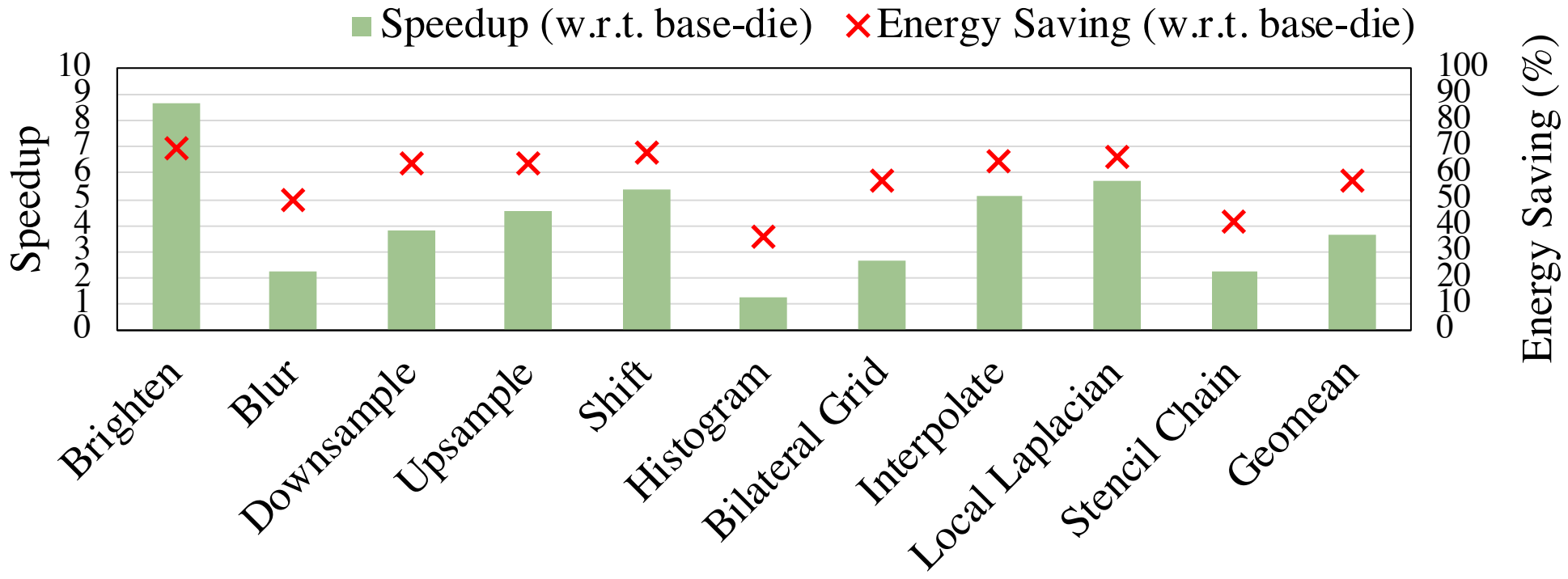
iPIM (Near-bank Arch) v.s. GPU



Compared to GPU baseline, iPIM achieves:

- 11.02x average speedup
- 79.49% average energy saving

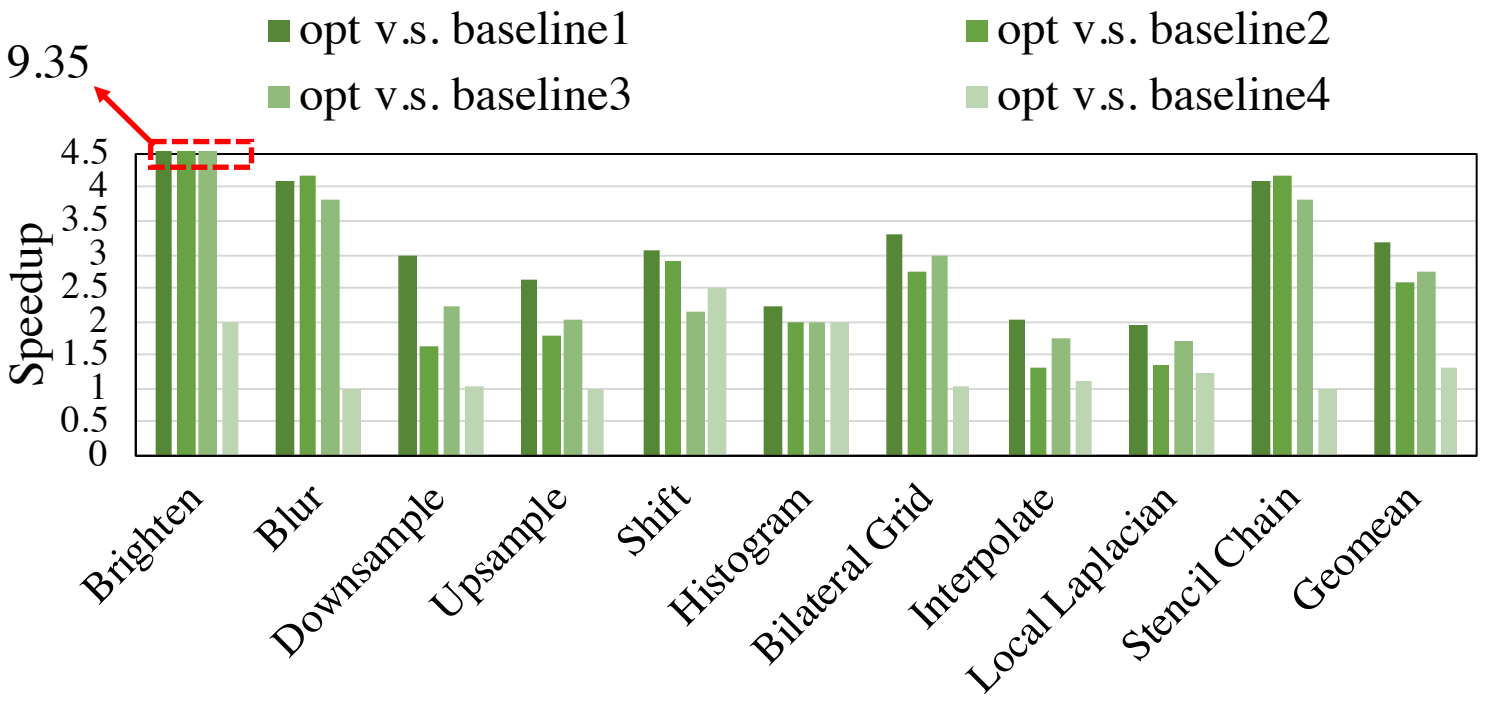
iPIM (Near-bank Arch) v.s. Process-on-base-die



Compared to process-on-base-die solution, iPIM achives:

- 3.61x average speedup
- 56.71% average energy saving

Effectiveness of iPIM Compiler Optimizations



opt: Apply all 3 optimizations
baseline1: No optimizations
baseline2: No register allocation optimization
baseline3: No instruction reordering
baseline4: No memory ordering enforcement

All three compiler backend optimizations together provide 3.19x speedup compared to unoptimized program

Instruction reordering is most effective: maximize instruction level parallelism

iPIM Key Takeaways:

- Lightweight programmable arch: **A decoupled control-execution architecture**
- Flexible ISA support: **Single-Instruction-Multiple-Bank (SIMB) ISA**
- End-to-end compilation flow: **Halide-iPIM**

- Evaluation results:
  - 11.02x speedup and 79.49% energy savings over state-of-the-art GPU accelerator
  - 3.61x speedup and 56.71% energy savings over the process-on-base-die solution
  - Overall compiler optimizations provide 3.19x speedup over unoptimized baseline

iPIM: Programmable In-Memory Image Processing Accelerator Using Near-Bank Architecture

Thank you!
Q&A

UC Santa Barbara
Scalable Energy-efficient
Architecture Lab