Exploiting Computation Reuse for Stencil Accelerators

Yuze Chi, Jason Cong

University of California, Los Angeles

{chiyuze,cong}@cs.ucla.edu

Presenter: Yuze Chi

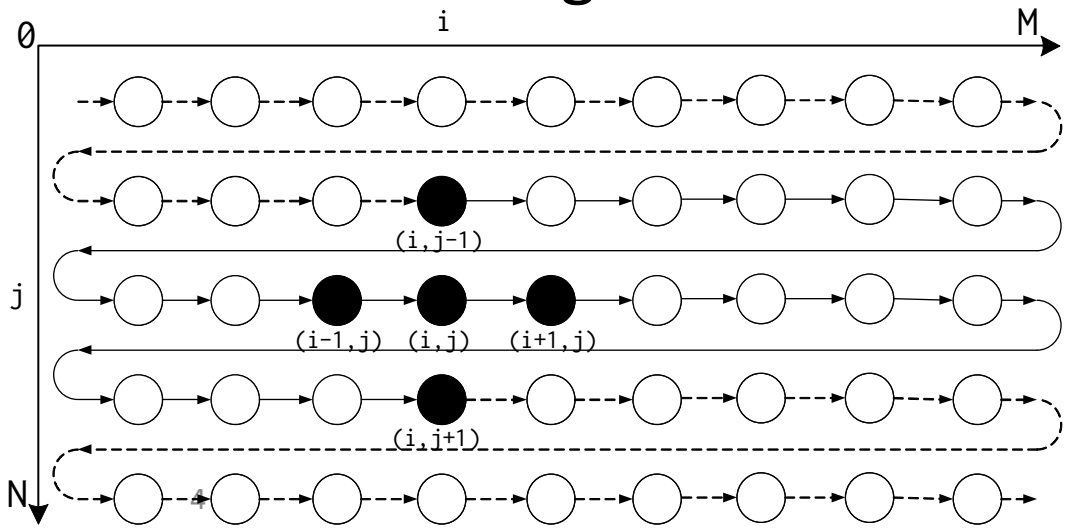
- PhD student in Computer Science Department, UCLA
- B.E. from Tsinghua Univ., Beijing
- Worked on software/hardware optimizations for graph processing, image processing, and genomics
- Currently building programming infrastructures to simplify heterogenous accelerator design
- https://vast.cs.ucla.edu/~chiyuze/

What is stencil computation?

What is Stencil Computation?

- A *sliding window* applied on an array
  - Compute output according to some fixed pattern using the stencil window
- Extensively used in many areas
  - Image processing, solving PDEs, cellular automata, etc.
- Example: a 5-point blur filter with uniform weights

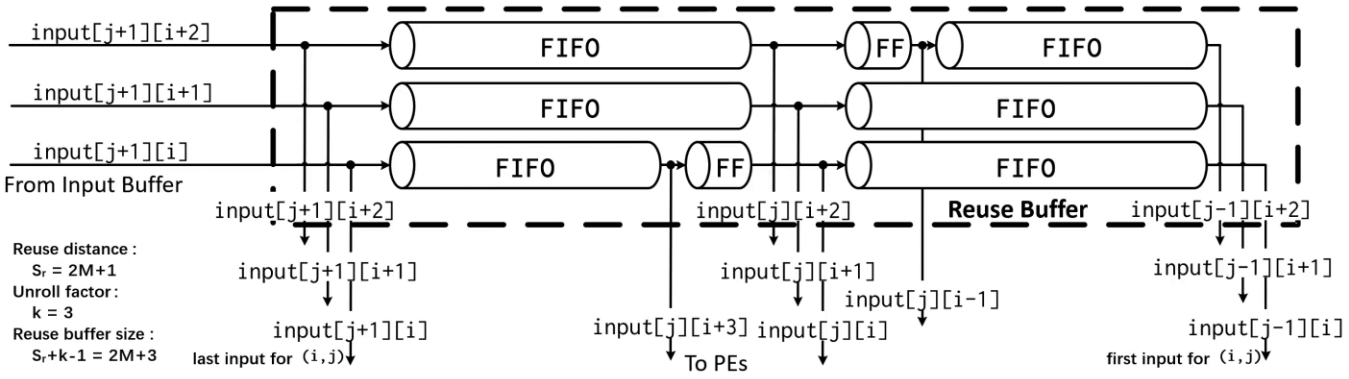
```
void blur(float X[N][M], float Y[N][M]) {
  for(int j = 1; j < N-1; ++j)
    for(int i = 1; i < M-1; ++i)
      Y[j][i] = (
        X[j-1][i  ] +
        X[j  ][i-1] +
        X[j  ][i  ] +
        X[j  ][i+1] +
        X[j+1][i  ]) * 0.2f;
}
```



How do people do stencil computation?

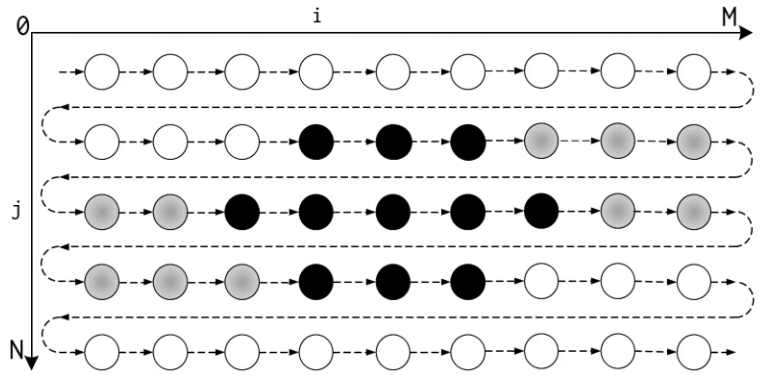
Three Aspects of Stencil Optimization

- Parallelization
  - Increase throughput
    - ICCAD'16, DAC'17, FPGA'18, ICCAD'18, …
- Communication Reuse
  - Avoid redundant memory access
    - DAC'14, ICCAD'18, …
- Computation Reuse
  - Avoid redundant computation
    - IPDPS'01, ICS'01, PACT'08, ICA3PP'16, OOPSLA'17, FPGA'19, TACO'19, …



Solved by SODA (ICCAD'18)

- ***Full*** data reuse
- ***Optimal*** buffer size
- ***Scalable*** parallelism



How can computation be *redundant*?

Computation Reuse

- Textbook Computation Reuse
  - Common-Subexpression Elimination (CSE)
    - `x = a + b + c; y = a + b + d;` `// 4 ops`
    - `tmp = a + b; x = tmp + c; y = tmp + d;` `// 3 ops`
- Tradeoff: Storage vs Computation
  - Additional registers for operation reduction
- Limitation
  - Based on Control-Data Flow Graph (CDFG) analysis/value numbering
  - Cannot eliminate all redundancy in stencil computation

Computation Reuse for Stencil Computation

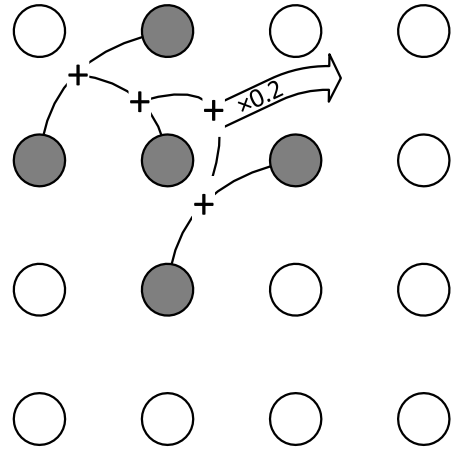
- Redundancy exists beyond a single loop iteration
  - Going back to the 5-point blur kernel

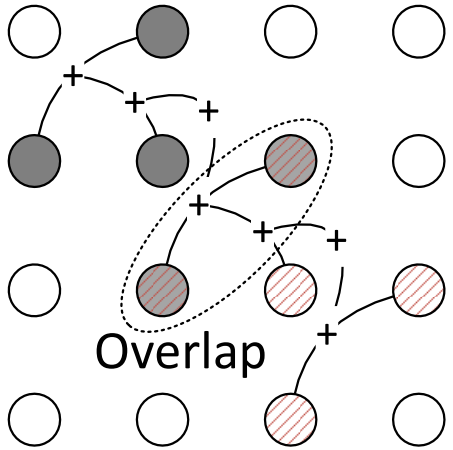
```
Y[j][i] = (X[j-1][i] + X[j][i-1] + X[j][i]
         + X[j][i+1] + X[j+1][i]) * 0.2f;
```

  - For different `(i, j)`, the stencil windows can overlap

```
Y[j+1][i+1] = (X[j][i+1] + X[j+1][i] + X[j+1][i+1]
             + X[j+1][i+2] + X[j+2][i+1]) * 0.2f;
```

  - Often called "*temporal*" since it crosses multiple loop iterations
- How to eliminate such redundancy?





Computation Reuse for Stencil Computation

- Computation reuse via an intermediate array
  - Instead of

```
Y[j][i] = (X[j-1][i] + X[j][i-1] + X[j][i]
         + X[j][i+1] + X[j+1][i]) * 0.2f;          // 4 ops per output
```

  - We do

```
T[j][i] = X[j-1][i] + X[j][i-1];
Y[j][i] = (T[j][i] + X[j][i] + T[j+1][i+1]) * 0.2f;  // 3 ops per output
```

- It looks very simple…?

What are the challenges?

Challenges of Computation Reuse for Stencil Computation

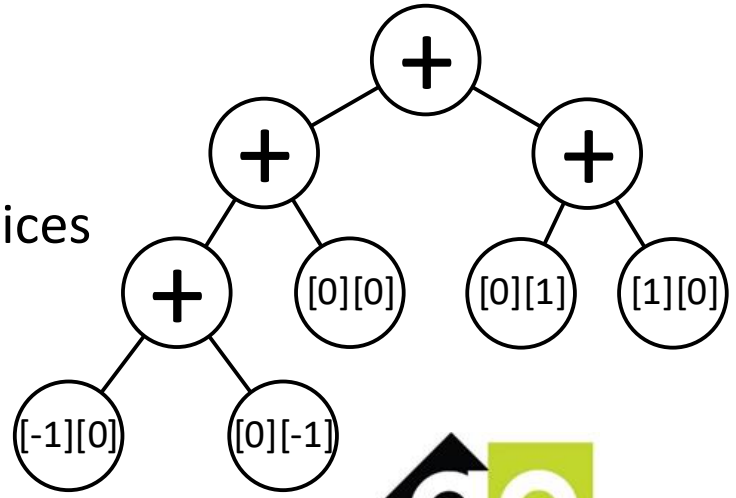
- Vast design space
  - Hard to determine the computation order of reduction operations
    - **(X[j-1][i] + X[j][i-1])** + X[j][i] + **(X[j][i+1] + X[j+1][i])** ✔
    - **(X[j-1][i] + X[j][i-1])** + **(X[j][i] + X[j][i+1])** + X[j+1][i] ❌
- Non-trivial trade-off
  - Hard to characterize the storage overhead of computation reuse
    - **T[j][i]** + X[j][i] + **T[j+1][i+1]**
    - For software: ~~register pressure~~ cache analysis / profiling / NN model
    - For hardware: concrete microarchitecture resource model

Computation reuse discovery

Find reuse opportunities from the vast design space

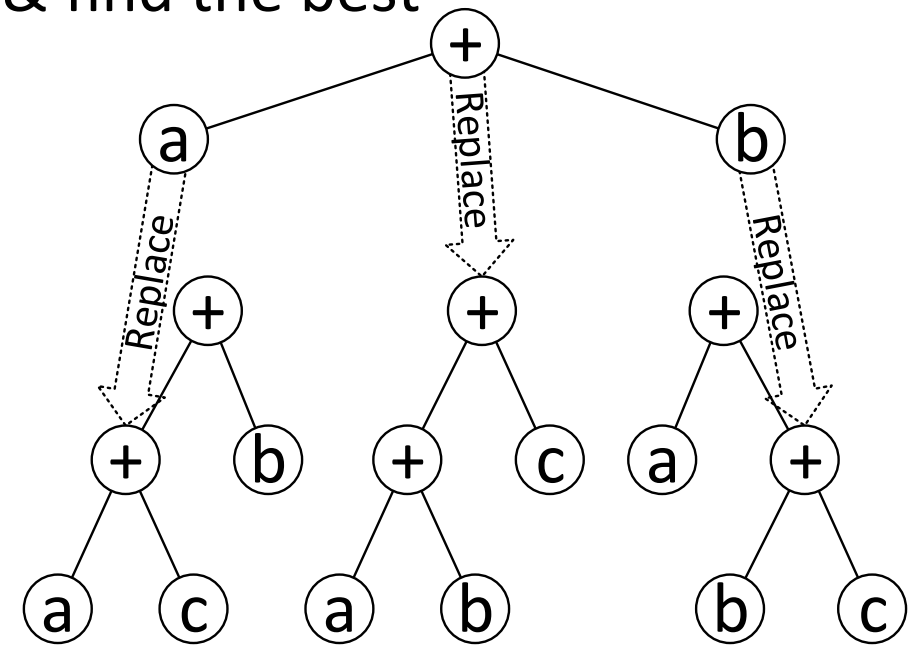
Find Computation Reuse by Normalization

- E.g. `((X[-1][0] + X[0][-1]) + X[0][0]) + (X[0][1] + X[1][0])`
  - Subexpressions (corresponding to the non-leaf nodes)
    - `X[-1][0] + X[0][-1] + X[0][0] +X[0][1] + X[1][0]`
    - `X[-1][0] + X[0][-1] + X[0][0]`
    - `X[0][1] + X[1][0]`
    - `X[-1][0] + X[0][-1]`
  - Normalization: subtract lexicographically least index from indices
    - `X[0][0] + X[1][-1] + X[1][0] + X[1][1] + X[2][0]`
    - `X[0][0] + X[1][-1] + X[1][0]`
    - **`X[0][0] + X[1][-1]`**
    - **`X[0][0] + X[1][-1]`**



Optimal Reuse by Dynamic Programming (ORDP)

- Idea: enumerate all possible computation order & find the best
  - Computation order ⇔ reduction tree
  - Enumeration via dynamic programming



- Computation reuse identified via *normalization*

Heuristic Search-Based Reuse (HSBR)

- ORDP is optimal but it only scales up to 10-point stencil windows
  - Need heuristic search!
- 3-step HSBR algorithm
  1. Reuse discovery
     - Enumerate all pairs of operands as common subexpressions
  2. Candidate generation
     - Reuse common subexpressions and generate new expressions as candidates
  3. Iterative invocation
     - Select candidates and iteratively invoke HSBR

HSBR Example

- E.g. `X[-1][0] + X[0][-1] + X[0][0] + X[0][1] + X[1][0]`
  - Reuse discovery
    - `X[-1][0] + X[0][-1]` can be reused for `X[0][1] + X[1][0]`
    - (other reusable operand pairs...)
  - Candidate generation
    - Replace `X[-1][0] + X[0][-1]` with `T[0][0]` to get `T[0][0] + X[0][0] + T[1][1]`
    - (generate other candidates...)
  - Iterative invocation
    - Invoke HSBR for `T[0][0] + X[0][0] + T[1][1]`
    - (invoke HSBR for other candidates…)

Computation Reuse Heuristics Summary

Paper	Temporal Exploration	Spatial Exploration	
	Inter-Iteration Reuse	Commutativity & Associativity	Operands Selection
Ernst [’94]	Via unrolling only	Yes	N/A
TCSE [IPDPS’01]	Yes	No	Innermost Loop
SoP [ICS’01]	Yes	Yes	Each Loop
ESR [PACT’08]	Yes	Yes	Innermost Loop
ExaStencil [ICA3PP’16]	Via unrolling only	No	N/A
GLORE [OOPSLA’17]	Yes	Yes	Each Loop + Diagonal
Folding [FPGA’19]	Pointwise operation only	No	N/A
DCMI [TACO’19]	Pointwise operation only	Yes	N/A
Zhao et al. [SC’19]	Pointwise operation only	Yes	N/A
HSBR [This work]	Yes	Yes	Arbitrary

Architecture-aware cost metric

Quantitively characterize the storage overhead

SODA μarchitecture + Computation Reuse

- SODA microarchitecture generates optimal communication reuse buffers
  - Minimum buffer size = reuse distance
- But for multi-stage stencil, total reuse distance can vary, e.g.
  - A two-input, two-stage stencil
    - $T[2] = X_1[0] + X_1[1] + X_2[0] + X_2[1]$
    - $Y[0] = X_1[3] + X_2[3] + T[0] + T[2]$
    - Total reuse distance: 3 + 3 + 2 = 8
      - $X_1[-1] \cdots X_1[2]$: 3
      - $X_2[-1] \cdots X_2[2]$: 3
      - $T[0] \cdots T[2]$: 2
  - Delay the first stage by 2 elements
    - $T[4] = X_1[2] + X_1[3] + X_2[2] + X_2[3]$
    - $Y[0] = X_1[3] + X_2[3] + T[0] + T[2]$
    - Total reuse distance: 1 + 1 + 4 = 6
      - $X_1[1] \cdots X_1[2]$: 1
      - $X_2[1] \cdots X_2[2]$: 1
      - $T[0] \cdots T[4]$: 4

SODA μarchitecture + Computation Reuse

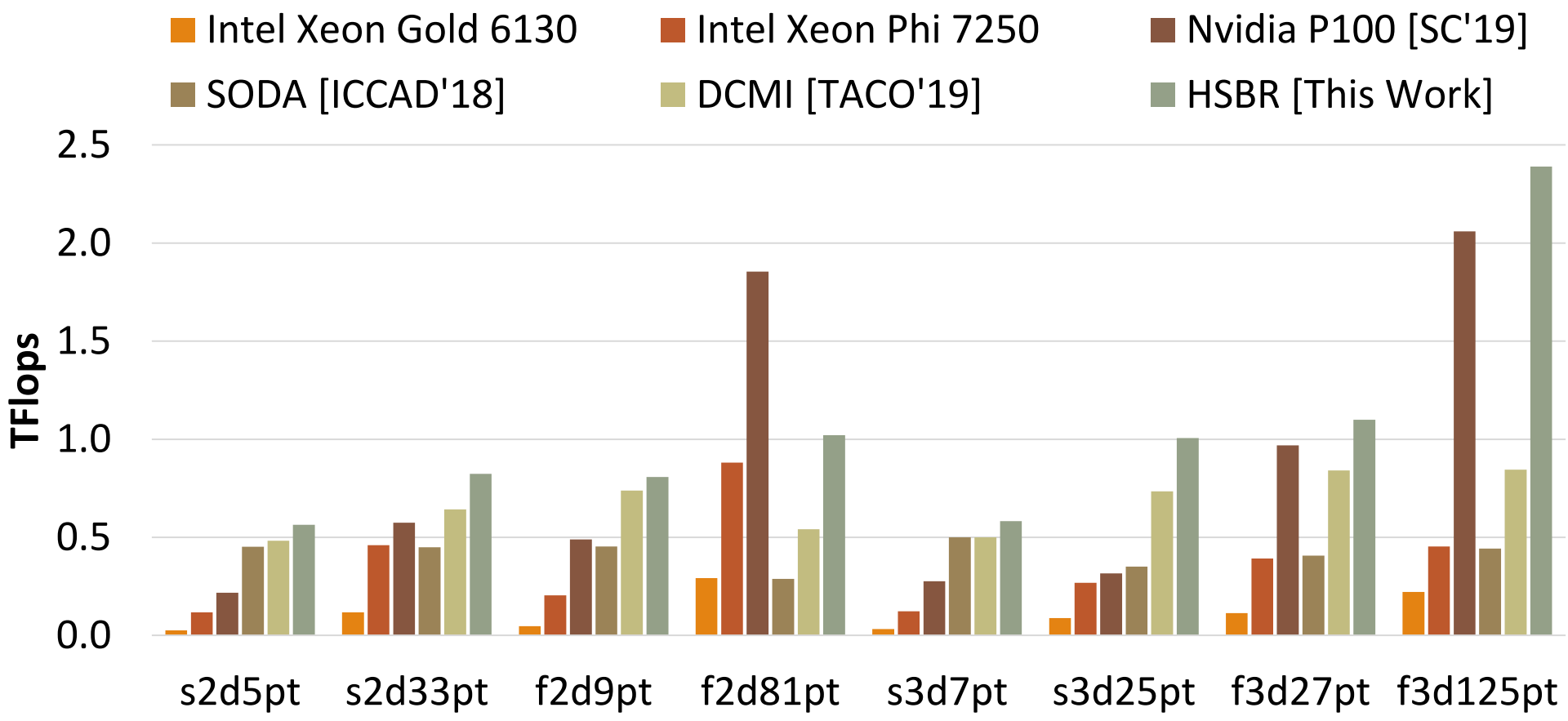
- Variables: different stages can produce outputs at different relative indices
  - E.g. $Y[0]$ and $T[2]$ vs $T[4]$ are produced at the same time
    - $T[2] = X_1[0] + X_1[1] + X_2[0] + X_2[1]$ vs $T[4] = X_1[2] + X_1[3] + X_2[2] + X_2[3]$
    - $Y[0] = X_1[3] + X_2[3] + T[0] + T[2]$
- Constraints: inputs needed by all stages must be available
  - E.g. $Y[0]$ and $T[1]$ cannot be produced at the same time because $T[2]$ is not available for $Y[0]$
- Goal: minimize total reuse distance & use as storage overhead metric
  - System of difference constraints (SDC) problem if all array elements have the same size
  - Solvable in polynomial time

Stencil Microarchitecture Summary

Paper	Intra-Stage		Inter-Stage	
	Parallelism	Buffer Allocation	Parallelism	Buffer Allocation
Cong et al. [DAC'14]	N/A	N/A	No	N/A
Darkroom [TOG'14]	N/A	N/A	Yes	Linearize
PolyMage [PACT'16]	Coarse-grained	Replicate	Yes	Greedy
SST [ICCAD'16]	N/A	N/A	Yes	Linear-Only
Wang and Liang [DAC'17]	Coarse-grained	Replicate	Yes	Linear-Only
HIPAcc [ICCAD'17]	Fine-grained	Coarsen	Yes	Replicate for each child
Zohouri et al. [FPGA'18]	Fine-grained	Replicate	Yes	Linear-Only
SODA [ICCAD'18]	Fine-grained	Partition	Yes	Greedy
HSBR [This work]	Fine-grained	Partition	Yes	Optimal

Experimental Results?

Performance Boost for Iterative Kernels



Operation/Resource Reduction (Geo. Mean)

Paper	Operation		Resource		
	Pointwise Operation	Reduction Operation	LUT	DSP	BRAM
SODA [ICCAD'18]	100%	100%	100%	100%	100%
DCMI [TACO'19]	19%	100%	85%	63%	100%
HSBR [This Work]	19%	42%	41%	45%	124%

- More details in the paper
  - Reduction of each benchmark
  - Impact of heuristics
  - Design-space exploration cost
  - Optimality gap

Conclusion

- We present
  - Two computation reuse discovery algorithms
    - Optimal reuse by dynamic programming for small kernels
    - Heuristic search–based reuse for large kernels
  - Architecture-aware cost metric
    - Minimize total buffer size for each computation reuse possibility
    - Optimize total buffer size over all computation reuse possibilities
- SODA-CR is open-source
  - https://github.com/UCLA-VAST/soda
  - https://github.com/UCLA-VAST/soda-cr

References

’94: Serializing Parallel Programs by Removing Redundant Computation, Ernst
IPDPS’01: Loop Fusion and Temporal Common Subexpression Elimination in Window-based Loops, Hammes et al.
ICS’01: Redundancies in Sum-of-Product Array Computations, Deitz et al.
PACT’08: Redundancy Elimination Revisited, Cooper et al.
DAC’14: An Optimal Microarchitecture for Stencil Computation Acceleration Based on Non-Uniform Partitioning of Data Reuse Buffers, Cong et al.
TOG’14: Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines, Hegarty et al.
PACT’16: A DSL Compiler for Accelerating Image Processing Pipelines on FPGAs, Chugh et al.
ICA3PP’16: Redundancy Elimination in the ExaStencils Code Generator, Kronawitter at al.
ICCAD’16: A Polyhedral Model-Based Framework for Dataflow Implementation on FPGA Devices of Iterative Stencil Loops, Natale et al.
OOPSLA’17: GLORE: Generalized Loop Redundancy Elimination upon LER-Notation, Ding et al.
ICCAD’17: Generating FPGA-based Image Processing Accelerators with Hipacc, Reiche et al.
DAC’17: A Comprehensive Framework for Synthesizing Stencil Algorithms on FPGAs using OpenCL Model, Wang and Liang
FPGA’18: Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL, Zohouri et al.
ICCAD’18: SODA: Stencil with Optimized Dataflow Architecture, Chi et al.
FPGA’19: LANMC: LSTM-Assisted Non-Rigid Motion Correction on FPGA for Calcium Image Stabilization, Chen et al.
TACO’19: DCMI: A Scalable Strategy for Accelerating Iterative Stencil Loops on FPGAs, Koraei et al.
SC’19: Exploiting Reuse and Vectorization in Blocked Stencil Computations on CPUs and GPUs, Zhao et al.

Questions

Acknowledgments

This work is partially funded by the NSF/Intel CAPA program (CCF-1723773) and NIH Brain Initiative (U01MH117079), and the contributions from Fujitsu Labs, Huawei, and Samsung under the CDSC industrial partnership program. We thank Amazon for providing AWS F1 credits.