Argonne NATIONAL LABORATORY

IMPLEMENTATION OF DATAFLOW SOFTWARE PIPELINING FOR CODELET MODEL

SIDDHISANKET RASKAR[1] ,JOSE MONSALVE DIAZ[1] , THOMAS APPLENCOURT[1] ,
KALYAN KUMARAN[1] , GUANG GAO[2]
[1]ARGONNE NATIONAL LABORATORY, LEMONT, IL, USA.
[2]UNIVERSITY OF DELWARE, NEWARK, DE, USA.

ICPE 2023, April 2023, Coimbra, Portugal

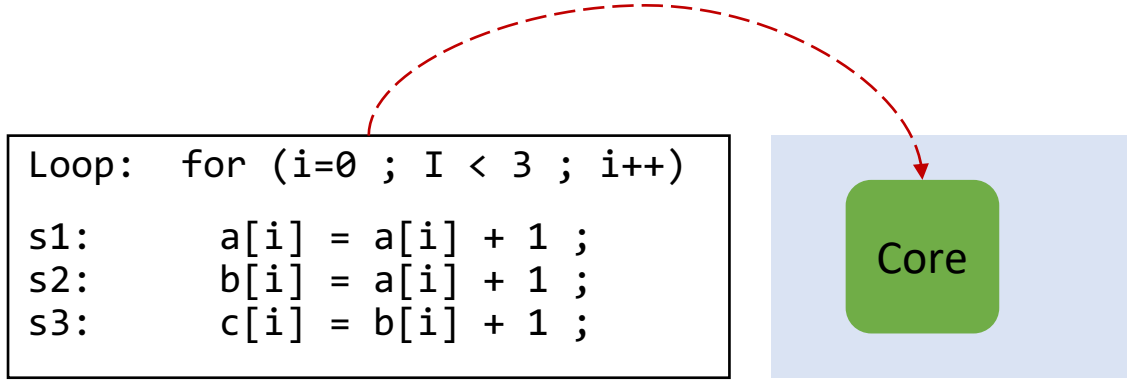
Agenda

- Motivation & Background
- Problem Formulation
- Solution Methodology
- Cannons Algorithm Case Study
- Experimental Evaluation
- Future Work & Conclusions

Software Pipelining

Software Pipelining is one of the most successful loop compilation technology in the exploitation of *Instruction Level Parallelism*

- ✓ Instruction Level Parallelism
- ✓ Fine Grained
- ✓ Single Core

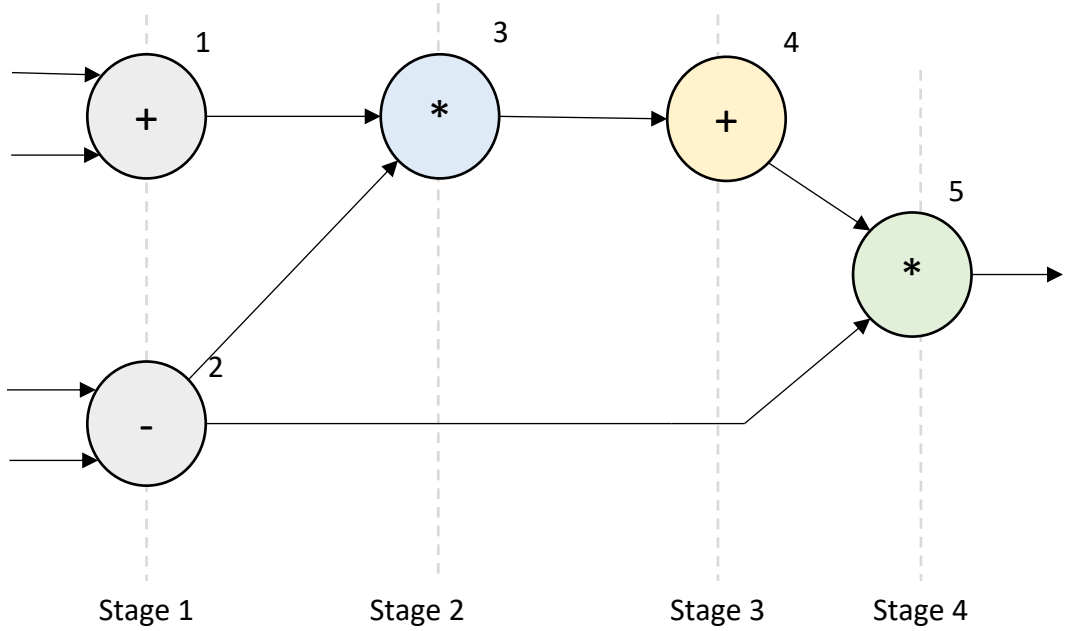


Background

Dataflow Software Pipelining

- ✓ Fine Grained
- ✓ Instruction Level Parallelism
- ✓ Maximum Throughput is achieved with balancing using *FIFO Buffers*

4 Stage Dataflow Software Pipeline



- ➢ Dataflow Software Pipelining is compile time as well as runtime technique.
- ➢ The naturally available information about dependencies is used at runtime for scheduling.
- ➢ Tokens from various iterations of the loop are executed.

- 5 Nodes, each is an Instruction
- Assume each instruction takes 1 cycle.

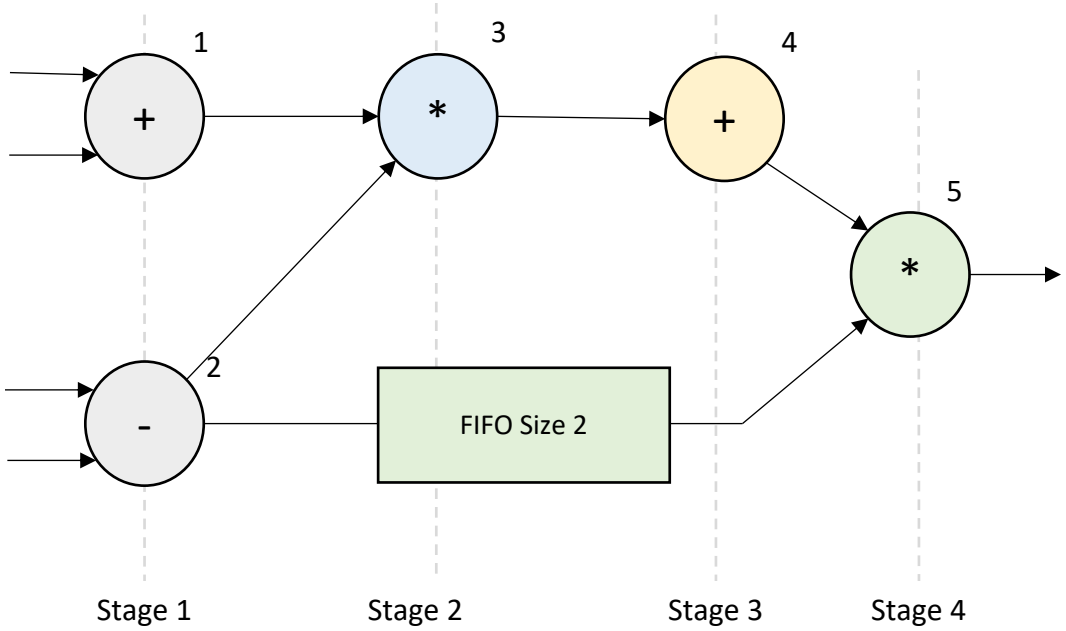
Reference: Guang R. Gao, Algorithmic aspects of balancing techniques for pipelined data flow code generation, Journal of Parallel and Distributed Computing, Volume 6, Issue 1, 1989.

Background

Dataflow Software Pipelining

- ✓ Fine Grained
- ✓ Instruction Level Parallelism
- ✓ Maximum Throughput is achieved with balancing using *FIFO Buffers*

4 Stage Dataflow Software Pipeline



- ➢ Dataflow Software Pipelining is compile time as well as runtime technique.
- ➢ The naturally available information about dependencies is used at runtime for scheduling.
- ➢ Tokens from various iterations of the loop are executed.

- 5 Nodes, each is an Instruction
- Assume each instruction takes 1 cycle.
- Balanced graph using *FIFO Buffer* of Size 2

Reference: Guang R. Gao, Algorithmic aspects of balancing techniques for pipelined data flow code generation, Journal of Parallel and Distributed Computing, Volume 6, Issue 1, 1989.

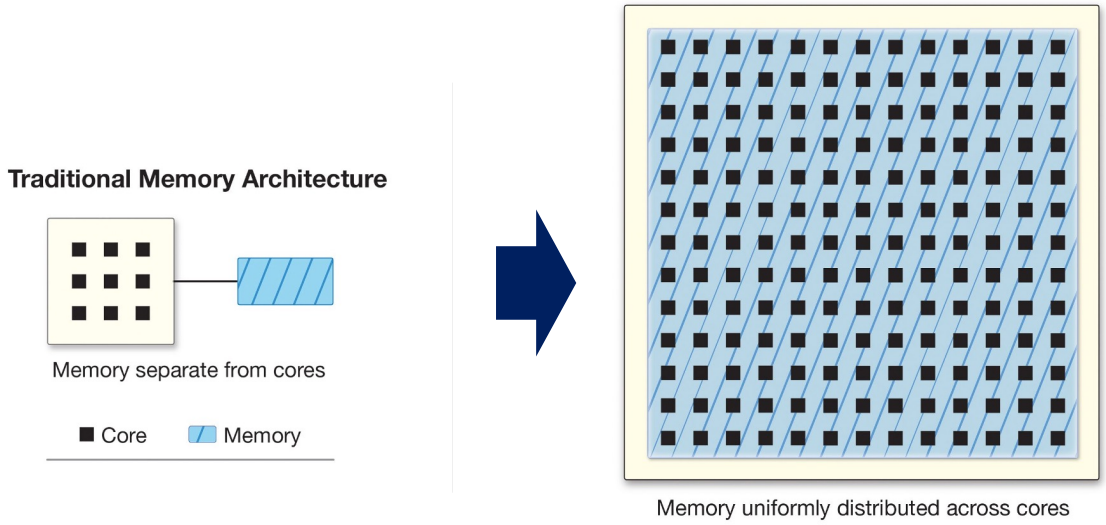
Background

Rise of Many Core Chips

- Rise of Many Core Architectures
- Changes is Computer Architecture

Challenges to extend this for multiple cores

- Variability of instruction timing between cores
- Loop carried dependencies must be realized across different cores.
- Variable runtime traffic in the on-chip network.



Cerebras

Habana

Samanova

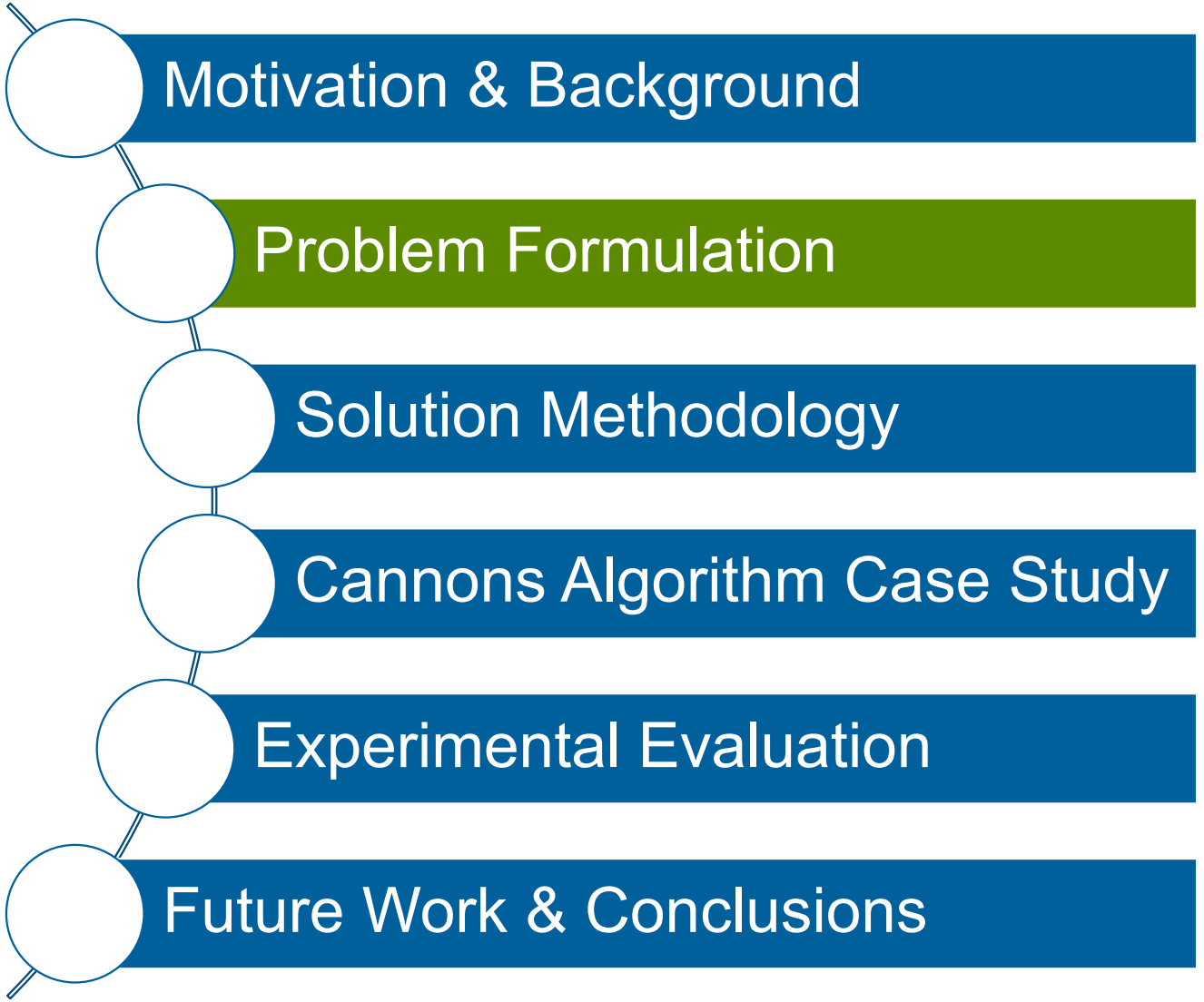
Graphcore

Groq

Reference: John L. Hennessy, David A. Patterson, A New Golden Age for Computer Architecture, Communications of the ACM, February 2019

https://www.alcf.anl.gov/alcf-ai-testbed

Agenda

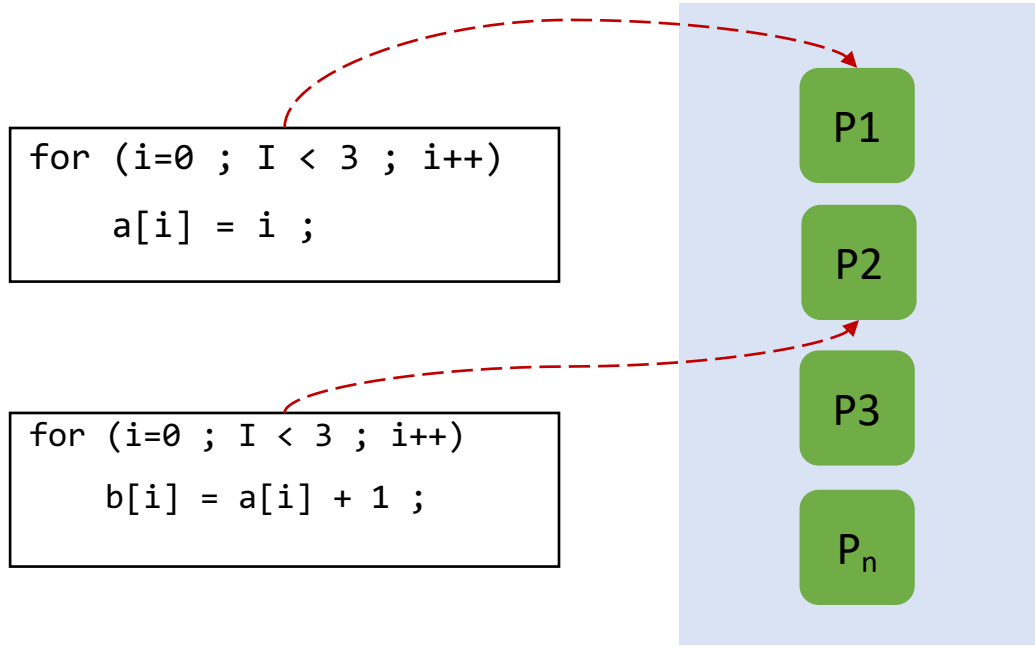


Problem Formulation

How should the success of
software pipelining
&
Dataflow Software Pipelining
can be exploited under
the new many core architecture era?

A model which will leverage

coarse grain parallelism at Codelet graph level
&
fine grain parallelism at Codelet level.

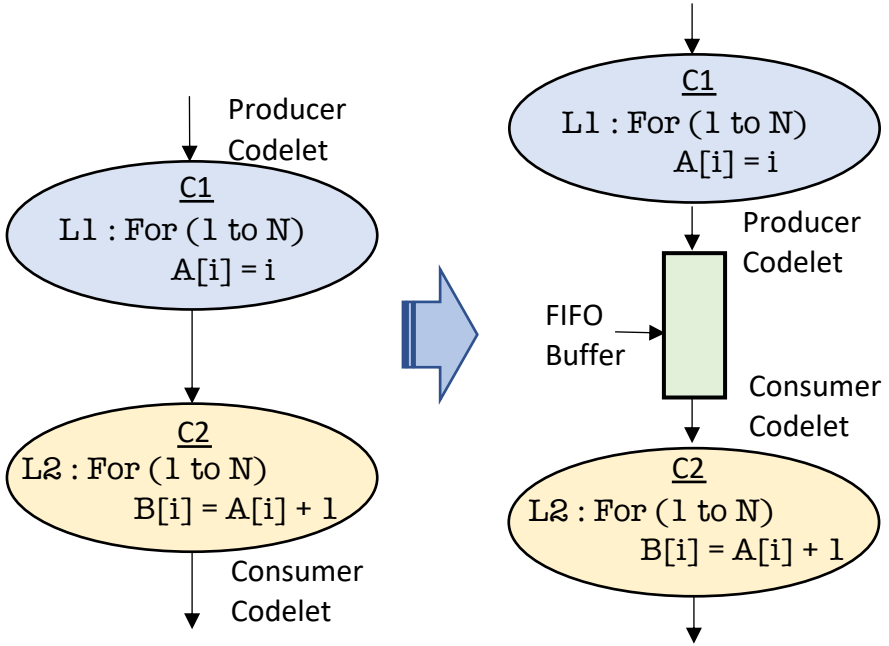


Problem Formulation

How should the success of dataflow software pipelining be exploited in the many core architecture era?

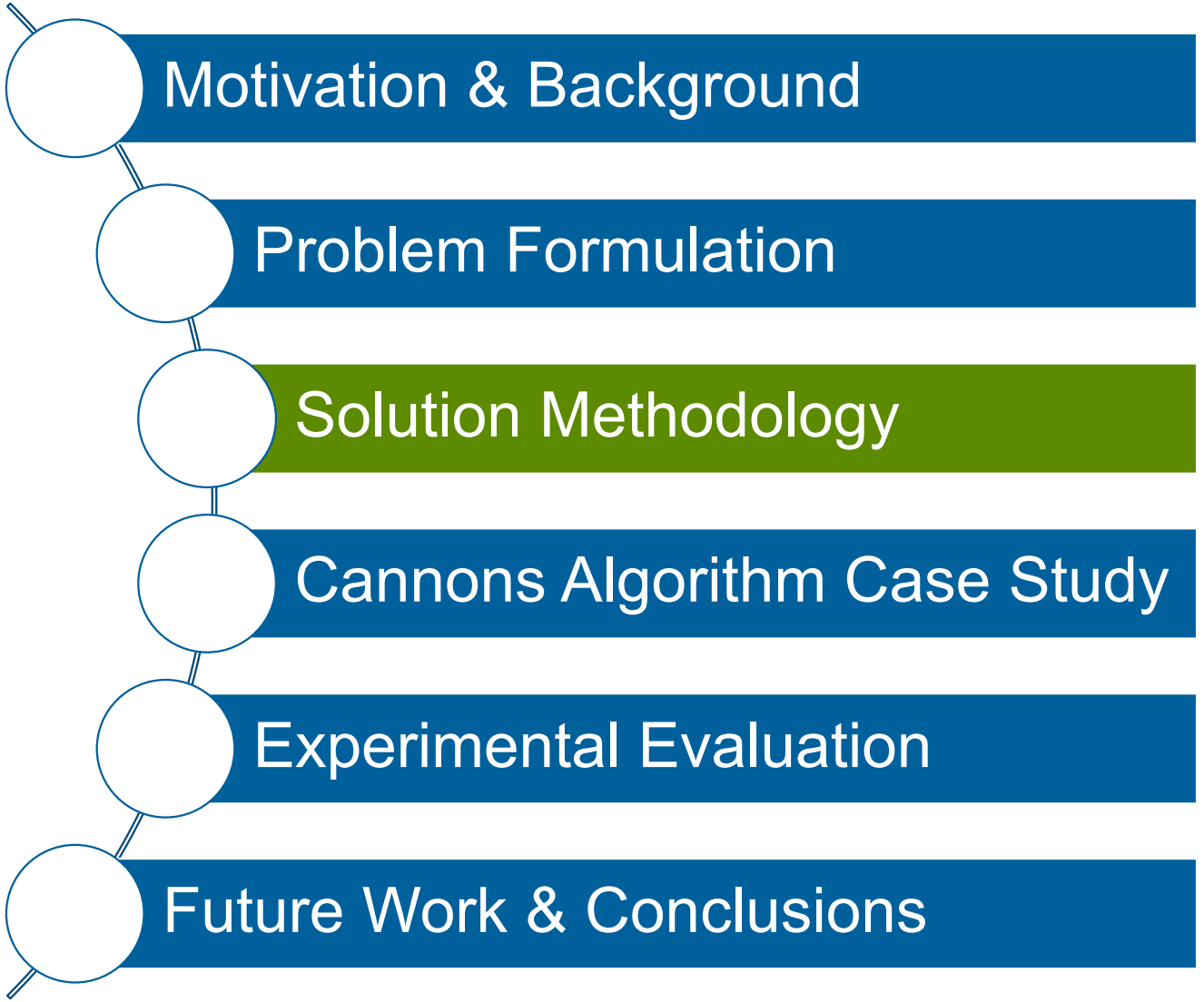
A programming model which will leverage coarse grain parallelism at Codelet graph level
&
fine grain parallelism at Codelet level.

- Extension to Codelet Model
  - Activity Model
  - Synchronization Model
- Extension to Codelet Abstract Machine to support efficient implementation of FIFO buffers



Reference : S. Raskar, T. Applencourt, K. Kumaran and G. Gao, "Position Paper: Extending Codelet Model for Dataflow Software Pipelining using Software-Hardware Co-Design," *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, Milwaukee, WI, USA, 2019

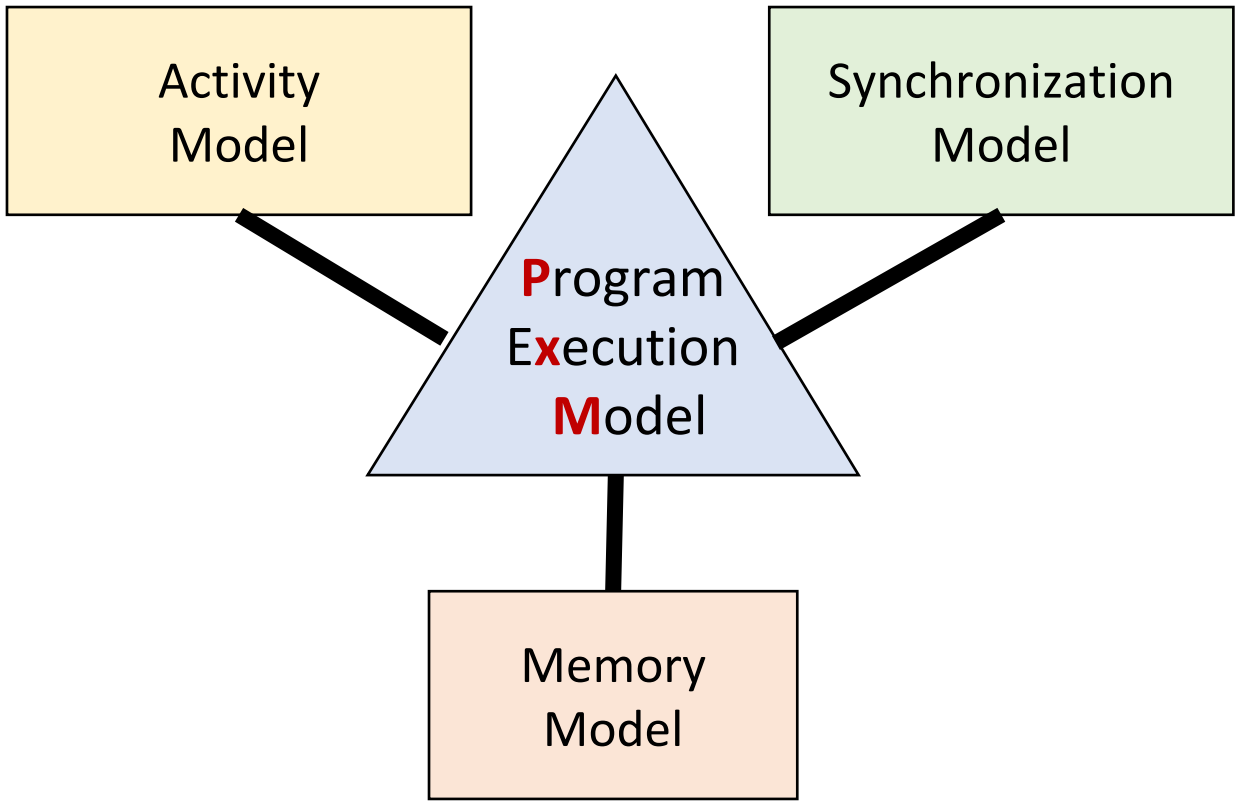
Agenda



Codelet Program Execution Model (PXM)

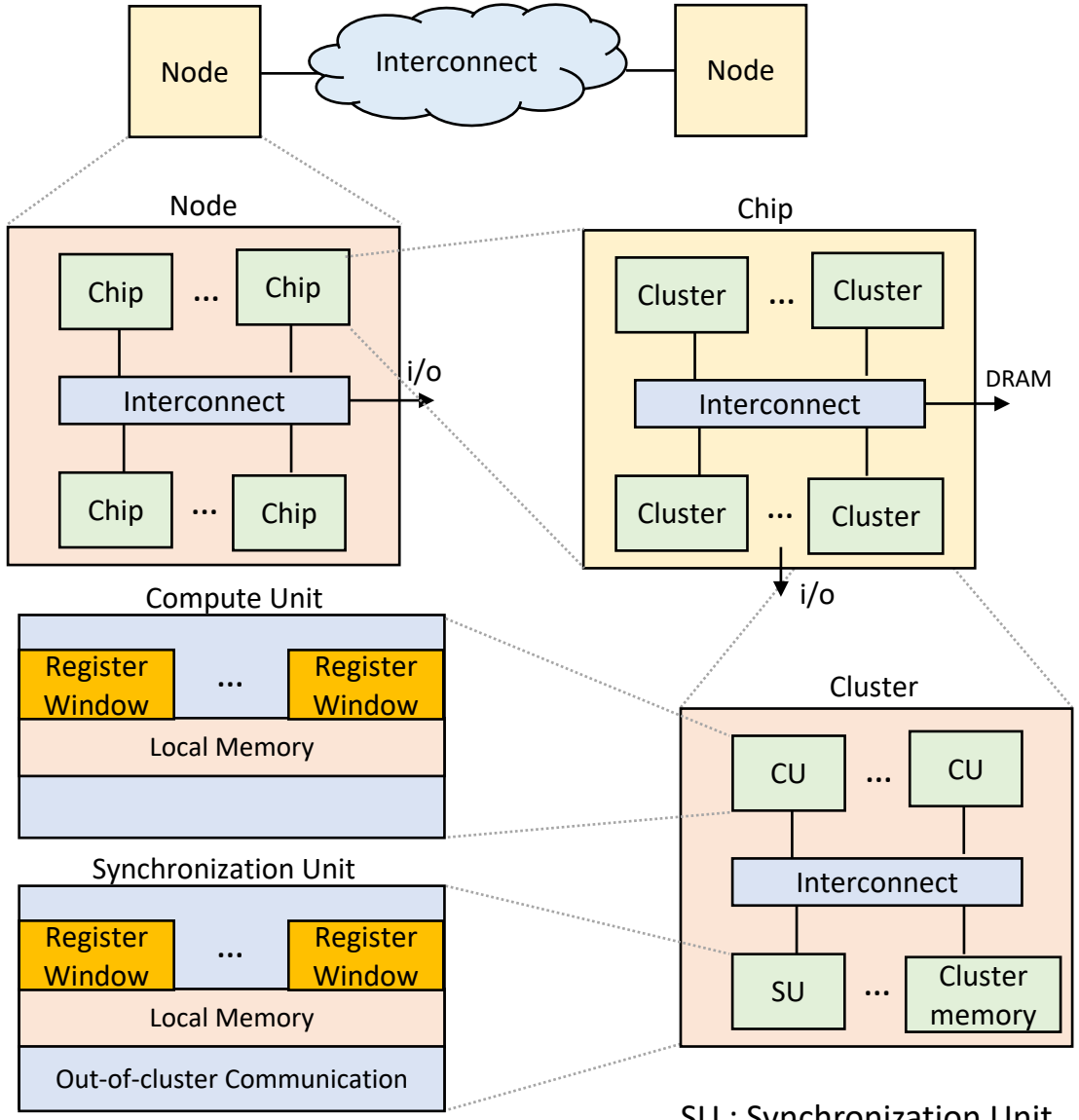
Defines Parallel activities

Defines interactions between activities



Specifies addressing model, results of memory operations & memory state transition

Codelet Abstract Machine (CAM)

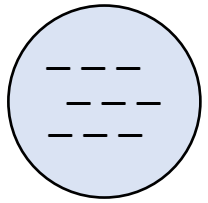


SU : Synchronization Unit
CU : Compute Unit

Codelet Model

Codelet

- Sequentially Executed
- Atomically Scheduled
- Non-Preemptive

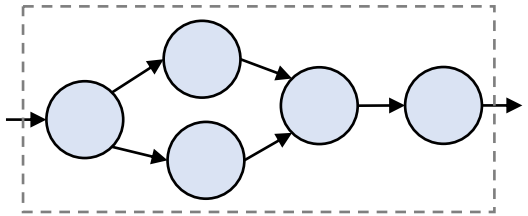


Codelet States

- Dormant
- Enabled
- Ready
- Fire

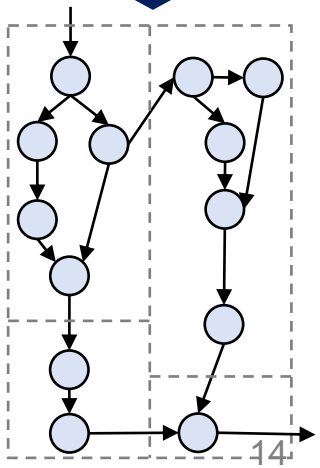
Threaded Procedure

- Collection of Codelets
- Shared input parameters
- Shared context & local variables



Codelet Graph

- Program representation
- Nodes -> Codelets
- Edges -> Dependencies



Firing Rules

- A codelet becomes *enabled* once tokens are present on each of its input arcs.
- An enabled codelet can be *fired* if it has acquired all its required resource.
- A codelet fires by consuming tokens on its input arcs, performing the operations within the codelet, and producing a token on each of its output arcs.

Activity Model

Synchronization Model

Codelet Model

Original Firing Rules

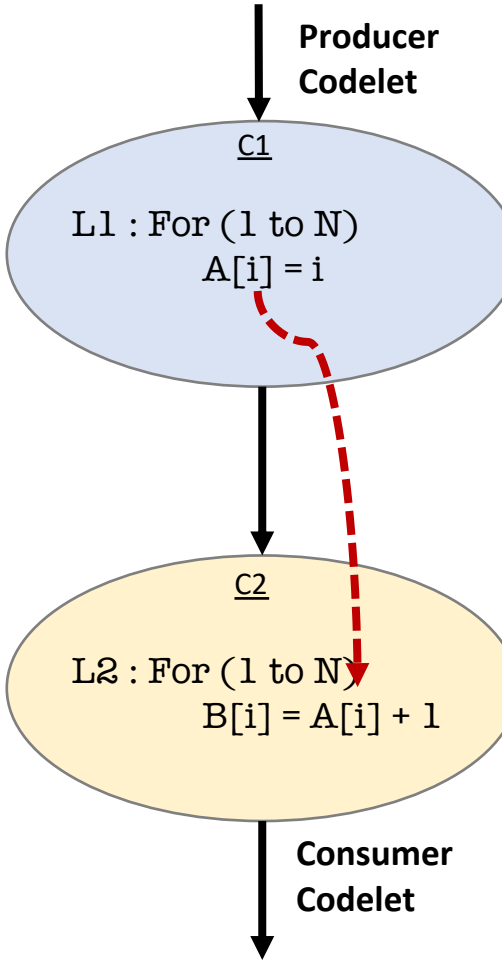
- A codelet becomes *enabled* once tokens are present on each of its input arcs.
- An enabled codelet can be *fired* if it has acquired all its required resource.
- A codelet fires by consuming tokens on its input arcs, performing the operations within the codelet, and producing a token on each of its output arcs.

- Consumer Codelet can **NOT** begin its execution while Producer Codelet is executing
- When Producer finishes its entire execution and sends event or data to the Consumer Codelet then only Consumer codelet can begin.

Extended Firing Rules

- For certain class of Codelet Graphs, Codelets marked by programmer or compiler can *enabled* even though input tokens from all iterations are not yet present on its input arcs.
- These enabled codelets will *fire* as soon as tokens from some iterations are present on its input arcs.

- Consumer Codelet **CAN** begin its execution while Producer Codelet is executing
- Tokens can stream from producer to consumer codelet



Extension to Codelet Abstract Machine

Why need to extend Codelet Model?

- HPC systems becoming more diverse, heterogeneous
  e.g. CPU, GPU, FPGA, ASIC

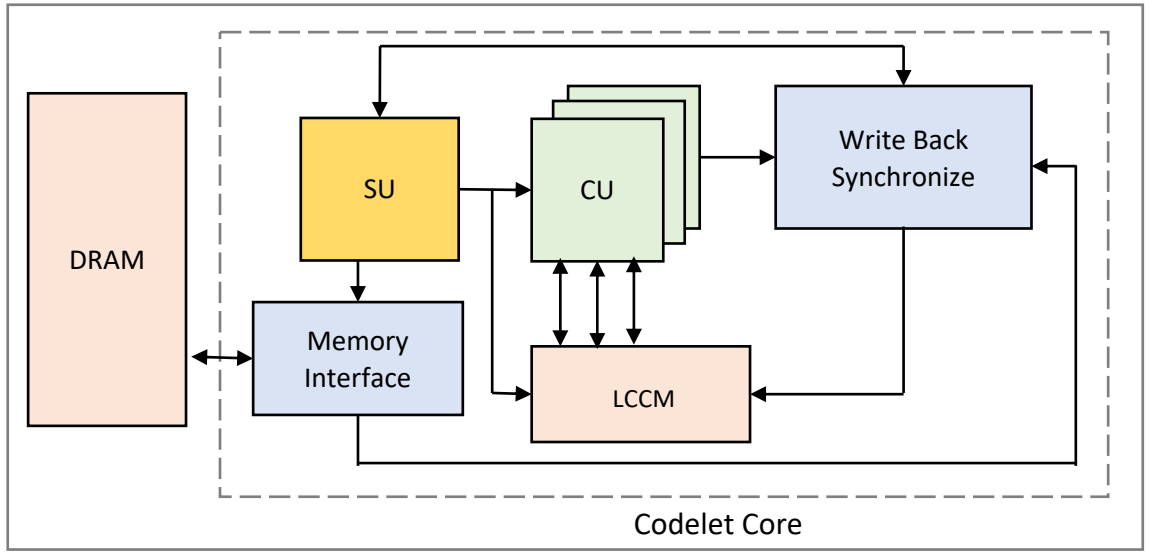
- Memory systems becoming more diverse
  e.g. Unified memory, Scratchpad memory, multiple levels of cache.

Codelet Core.

- Moving memory out of Codelet level core to hide latencies in memory operations.

Local Codelet Core Memory **(LCCM)**

- Efficient FIFO Buffers



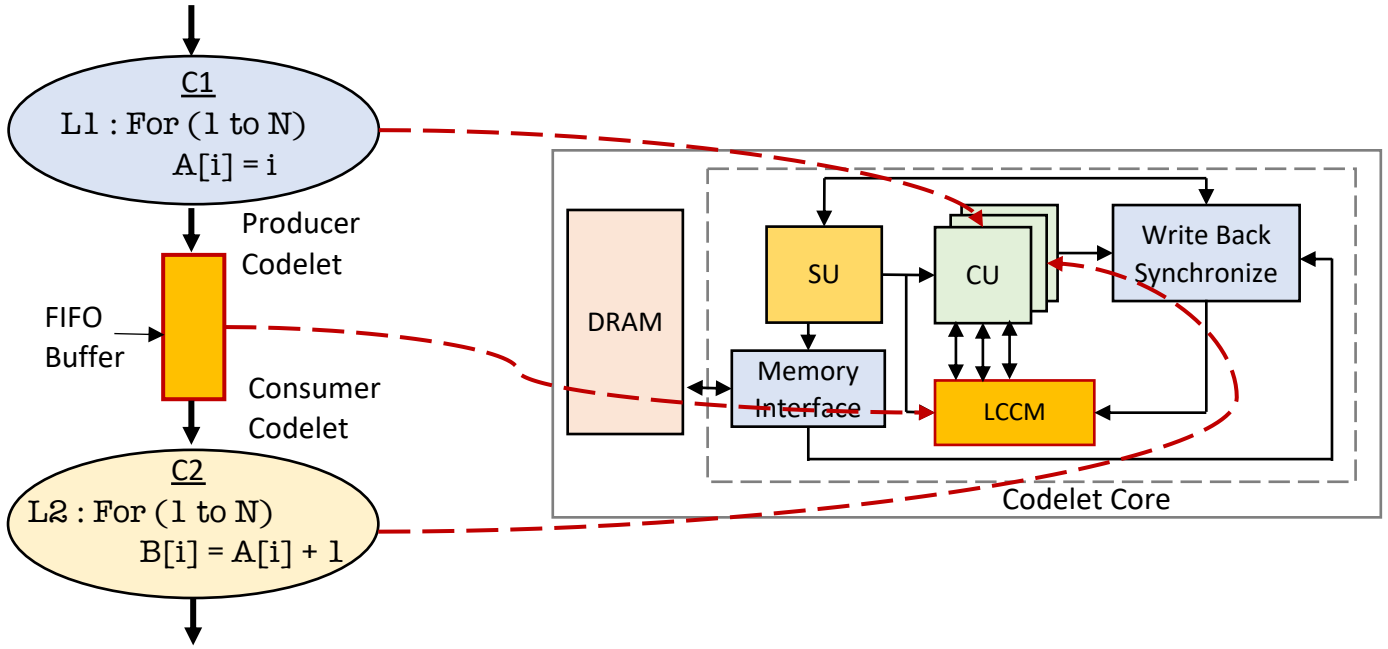
Extended Codelet Abstract Machine Model (xCAM)

SU : Synchronization Unit
CU : Compute Unit
LCCM : Local Codelet Core Memory

Extension to Codelet Abstract Machine

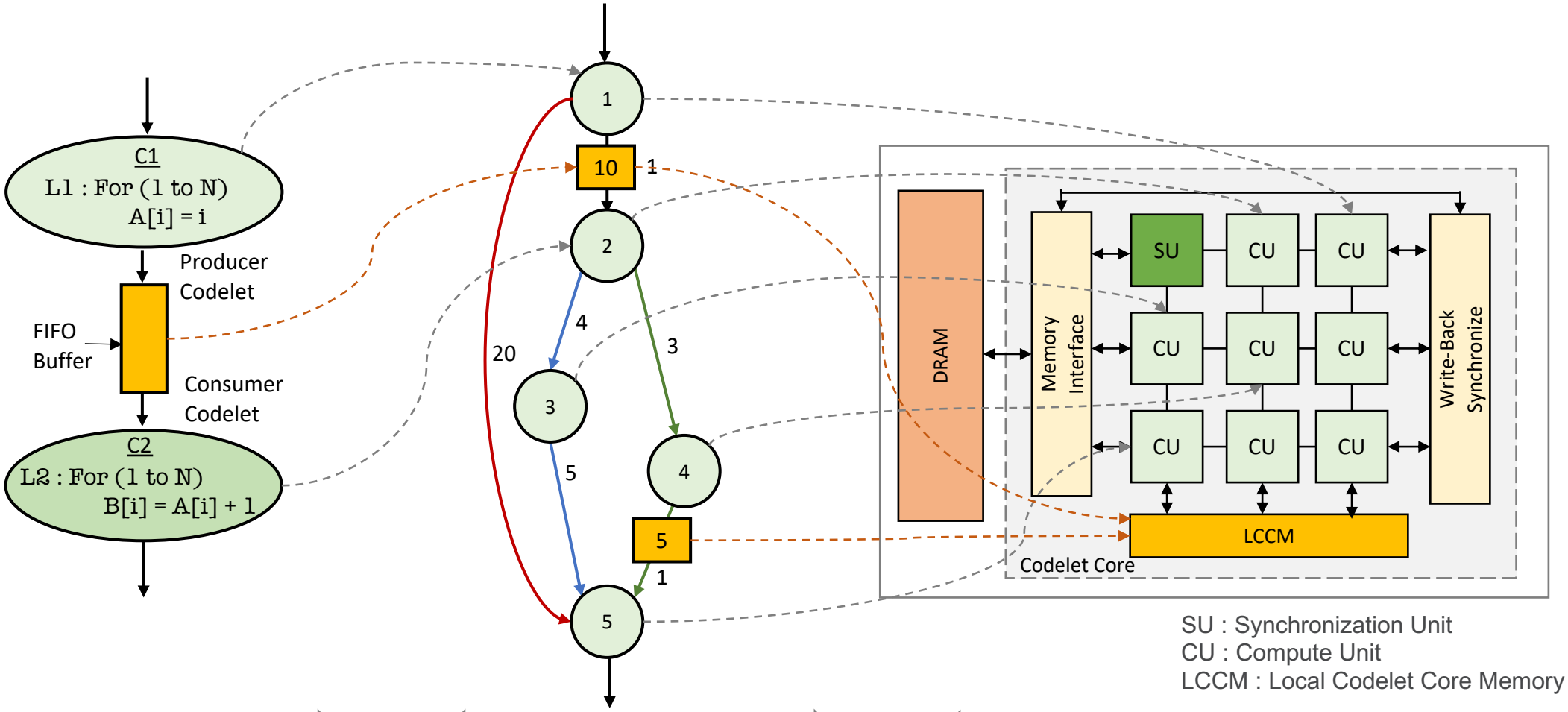
Producer & Consumer Codelets -

- ✓ Execute *simultaneously* on the same *Codelet Core*
- ✓ Tokens *stream* continuously from Producer to Consumer via FIFO buffers
- ✓ FIFO buffers are mapped to LCCM for efficient execution.



Solution Methodology

Extended Codelet Model For Dataflow Software Pipelining



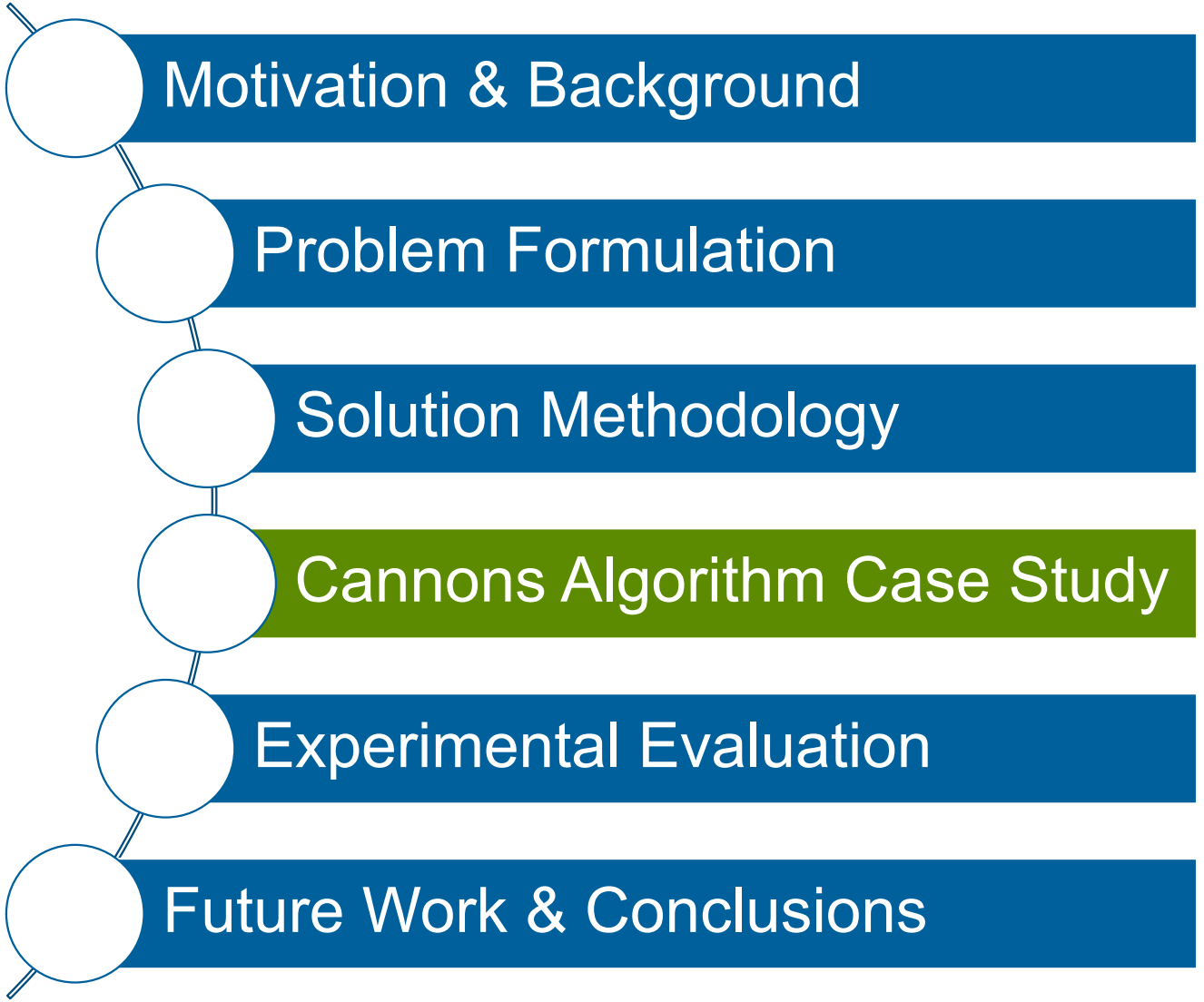
SU : Synchronization Unit
CU : Compute Unit
LCCM : Local Codelet Core Memory

Producer-Consumer Codelets in a CDG

Codelet Graph (CDG)

Extended Codelet Abstract Machine(xCAM)

Agenda



Case Study

Cannons Algorithm under DARTS

Cannons Algorithm

- Matrix Multiplication is important kernel behind many scientific as well for Machine Learning application domain
- Over lapping of *computation* and *communication* phase gives opportunity to demonstrate advantage of dataflow software pipelining techniques
- Satisfies our specification for the class of codelet graph both at Codelet Graphs level & Codelet level.

Delaware Adaptive Run-Time System (DARTS)

- implementation of the Codelet Model on x86.
- Open Source, Written in C++, 42000 lines of code
- Classes are used to represent Codelets and Threaded Procedures
- Data transmission through shared memory, signal transmission through function calls

Extensions to Codelet Model

With FIFO Buffers

Cannons Algorithm Pseudocode

```
forall i = 0 : n - 1
      left circular shift row i by i,
      so that A_{i,j} is assigned to A_{i , (j+1) mod n}
```

Matrix A

Shift **left** each element in **row i by i times**.

```
forall j = 0 : n – 1
      upward circular shift column j by j,
      so that B_{i,j} is assigned B_{(j+1) mod n , j}
```

Matrix B

Shift **up** each element in column j by j times.

Step 1 : Skew / Initialize the Matrices

```
for k = 1 : n
      forall i = 0 : n -1
            forall j = 0 : n - 1
                  C_{i,j}  =  C_{ij} + A_{ij} . B_{ij}
```

Computation

```
                  left circular shift each row of a by 1,
                  so A_{ij} is assigned A_{i, (j+1) mod n}
                  upward circular shift each column of b by 1,
                  so B_{ij} is assigned B_{(i+1) mod n, j}
```

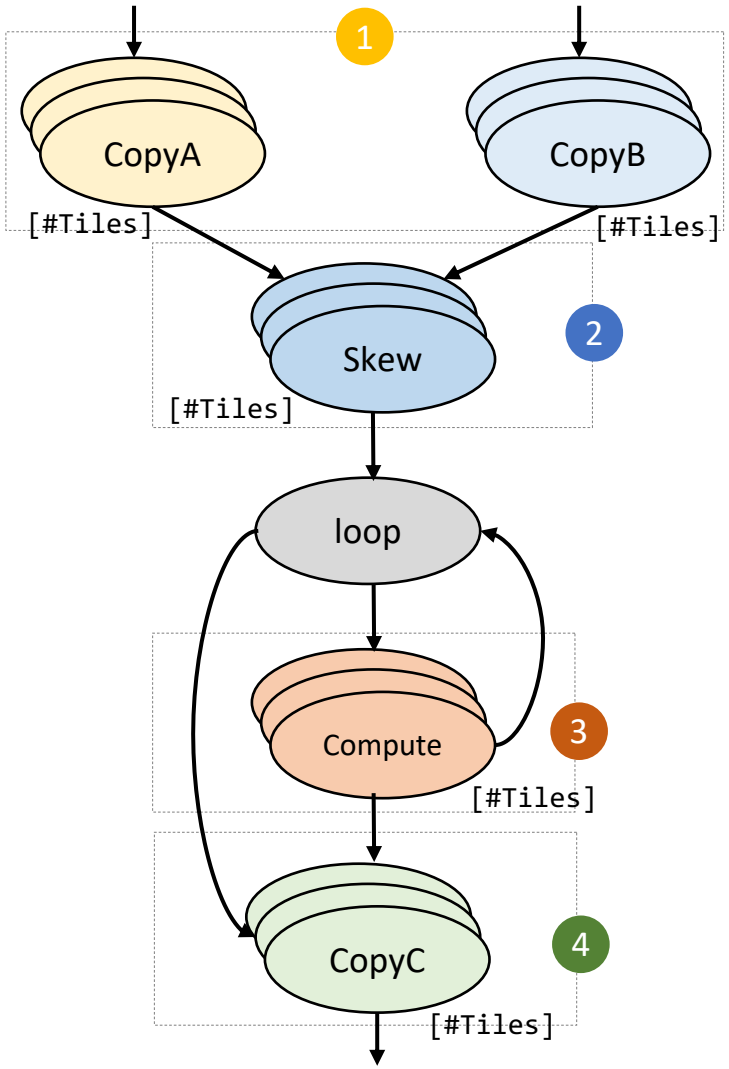
Communication

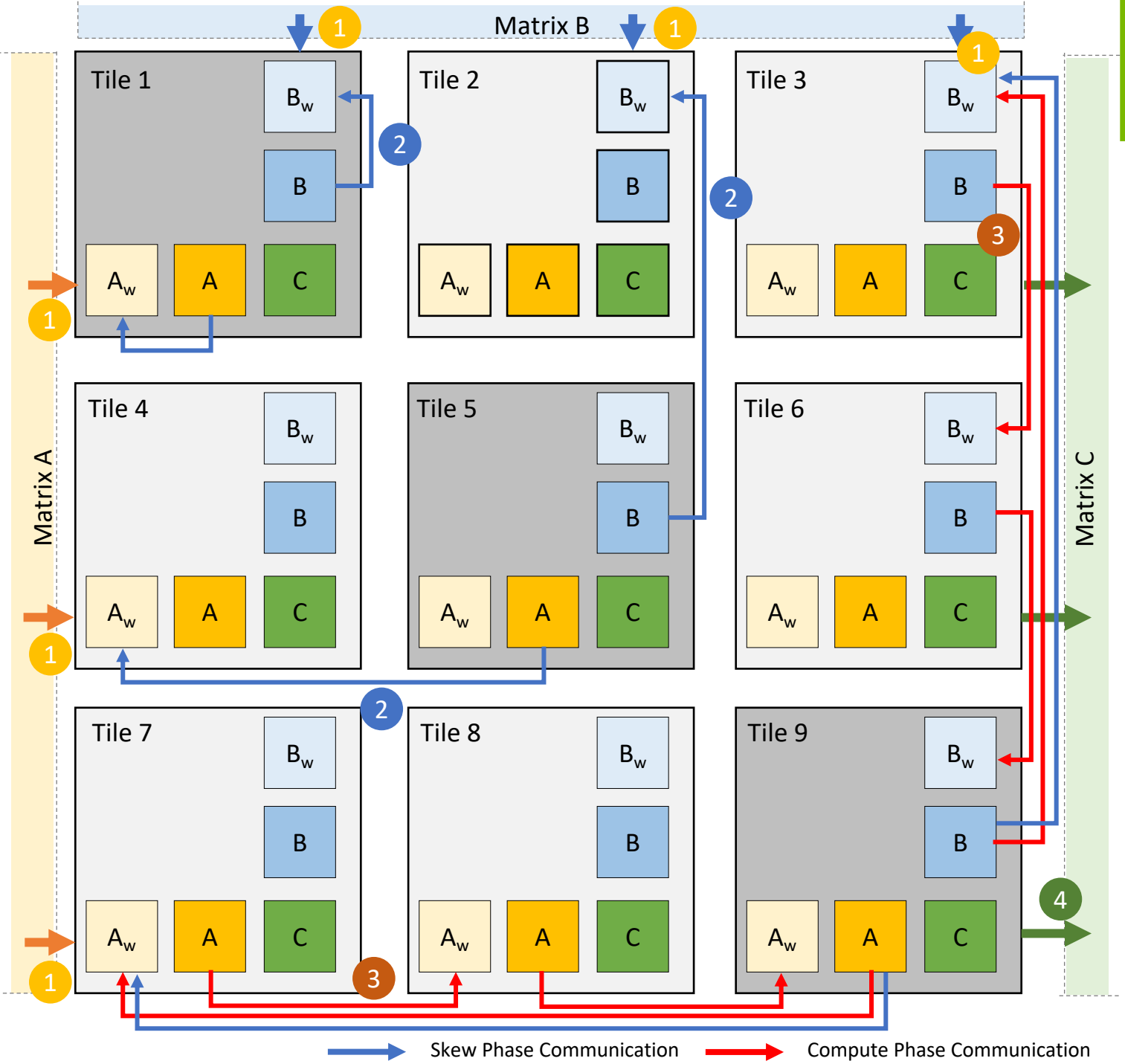
Step 2 : Shift & Multiply

Case Study

CDG : Without DF-SWP

- **CopyA and CopyB:**
  Copy original matrix `A` and `B` to tile memory local to each codelet.
- **Skew:** skew/initialize matrix `A` and `B`
- **loop:**
  iterates `P` times.
  acts as a `barrier` between different instances of compute codelet.
- **Compute:**
  multiplies sub-matrix `A` and `B`, stores results in sub-matrix `C`.
  Circularly shifts sub-matrix `A` and `B`.
  sends a signal to `loop` codelet when finished.
- **CopyC:** When `loop` codelet finishes its `P` iterations, resultant sub-matrix `C` computation is complete. Now, `CopyC` codelet simply copies sub-matrix `C` back to main memory from tile memory





Case Study

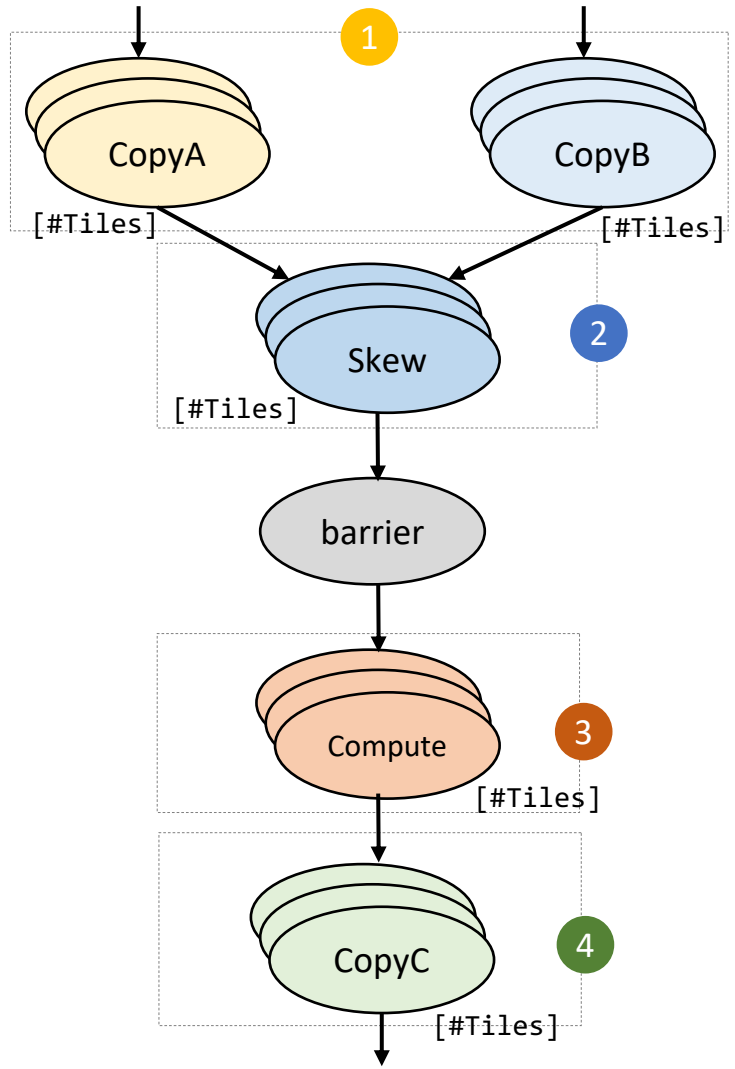
Communication Without DF-SWP

- **Stage 1:** `CopyA` and `CopyB` codelets copy sub-matrix `A` and `B` from main memory to tile memory of codelets. This is shown with the bold arrows on (top and left periphery)
- **Stage 2:** Sub-matrix `A` and `B` are skewed. Skew codelet per-forms this operation. Sub-matrix blocks for `A` and `B` along with `Aw` and `Bw` are used with `skew` phase communication shown using blue arrows.
- **Stage 3:** Sub-Matrix `C` is calculated using `Compute` codelet. Sub-matrix `A` and `B` are circularly shifted causing `computation` phase communication also shown using red arrows.
- **Stage 4:** Sub-matrix `C` is copied back to main memory from tile memory of codelets. This is shown using bold arrows (right side periphery)

Case Study

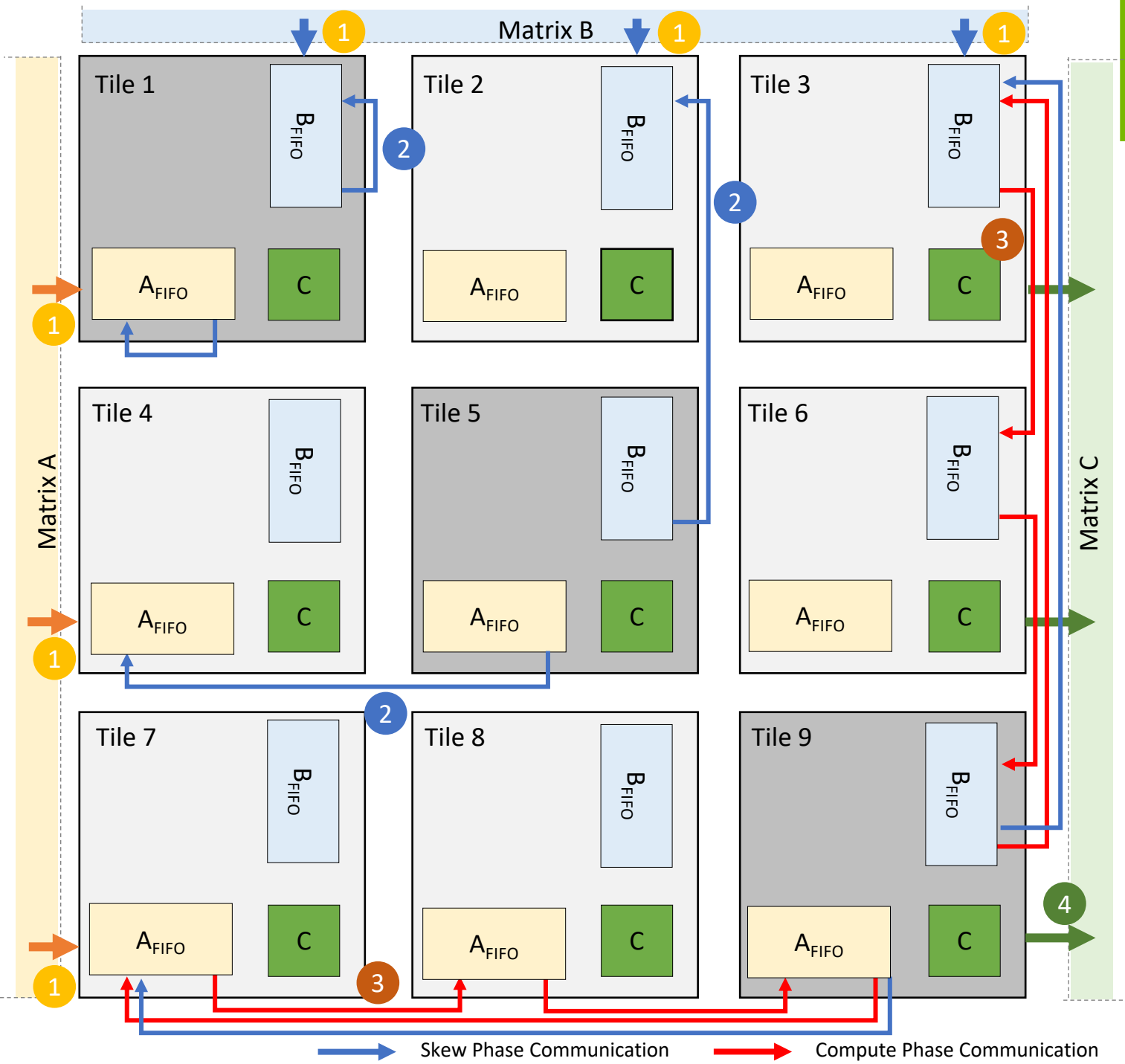
CDG : With DF-SWP

- **loop:** This codelet iterates `P` times. This codelet acts as a `barrier` between different instances of compute codelet.
- **Compute:** This codelet, multiplies sub-matrix `A` and `B`, stores results in sub-matrix `C`. It also circularly shifts sub-matrix `A` and `B`. It sends a signal to `loop` codelet when finished.
- **CopyC:** When `loop` codelet finishes its `P` iterations, resultant sub-matrix `C` computation is complete. Now, `CopyC` codelet simply copies sub-matrix `C` back to main memory from tile memory



Case Study

Communication With DF-SWP



- **Stage 1:** `CopyA` and `CopyB` codelets copy sub-matrix `A` and `B` from main memory to tile memory of codelets. This is shown with the bold arrows on (top and left periphery)
- **Stage 2:** Sub-matrix `A` and `B` are skewed. Skew codelet per-forms this operation. Sub-matrix blocks for `A` and `B` along with `Aw` and `Bw` are used with `skew` phase communication shown using blue arrows.
- **Stage 3:** Sub-Matrix `C` is calculated using `Compute` codelet. Sub-matrix `A` and `B` are circularly shifted causing `computation` phase communication also shown using red arrows.
- **Stage 4:** Sub-matrix `C` is copied back to main memory from tile memory of codelets. This is shown using bold arrows (right side periphery)

Evaluation

Experimental Setup

Evaluate Cannon's algorithm using dataflow-based runtime DARTS

- **Without DF-SWP (Baseline):** This implementation uses a loop codelet as a barrier between iterations of compute codelets.
- **With DF-SWP:** extend baseline implementation with dataflow software pipelining by using FIFO buffers

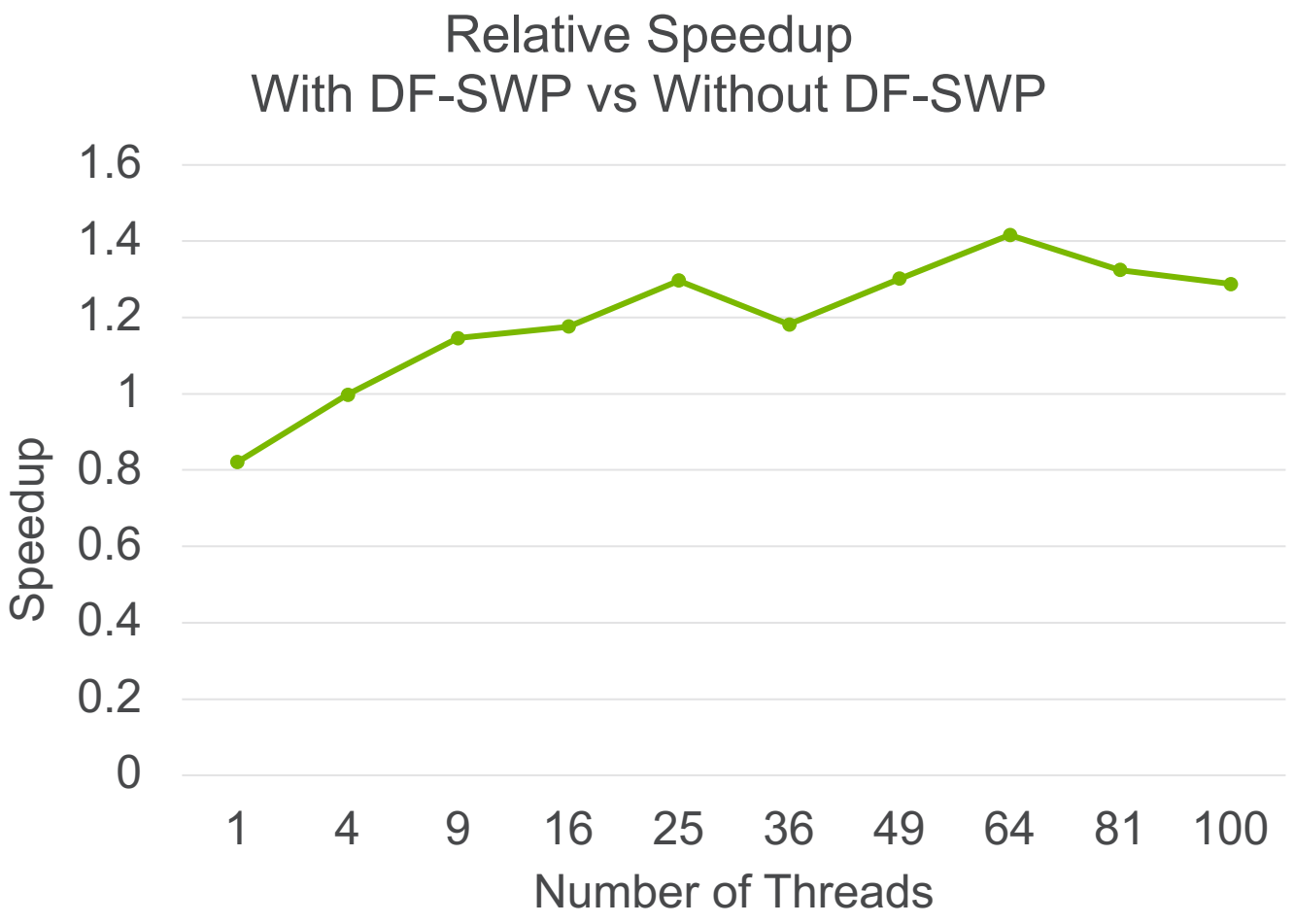
- Two sockets, 28 cores per socket
- Intel Xeon Platinum 8180M(Skylake) processor clocked at 2.5GHz with Hyper-Threading (HT)
- $32KB$ private L1, 1MB private unified L2 caches
- 383GB of DRAM divided into two NUMA
- Red Hat Enterprise Linux 7.5
- GCC 8.2 with optimizations set to -O3

- map threads on separate cores until all cores were assigned at least one thread
- KMP_AFFINITY parameter and set it to BALANCED with granularity as CORE
- fine-tune DARTS AMM by setting the scheduler affinity policy as COMPACT_NO_SMT (SUs and CUs are pinned down to physically contiguous cores without using Hyper-Threading until all physical cores are used)
- restricted to only square matrices.

Agenda

- Motivation & Background
- Problem Formulation
- Solution Methodology
- Cannons Algorithm Case Study
- Experimental Evaluation
- Future Work & Conclusions

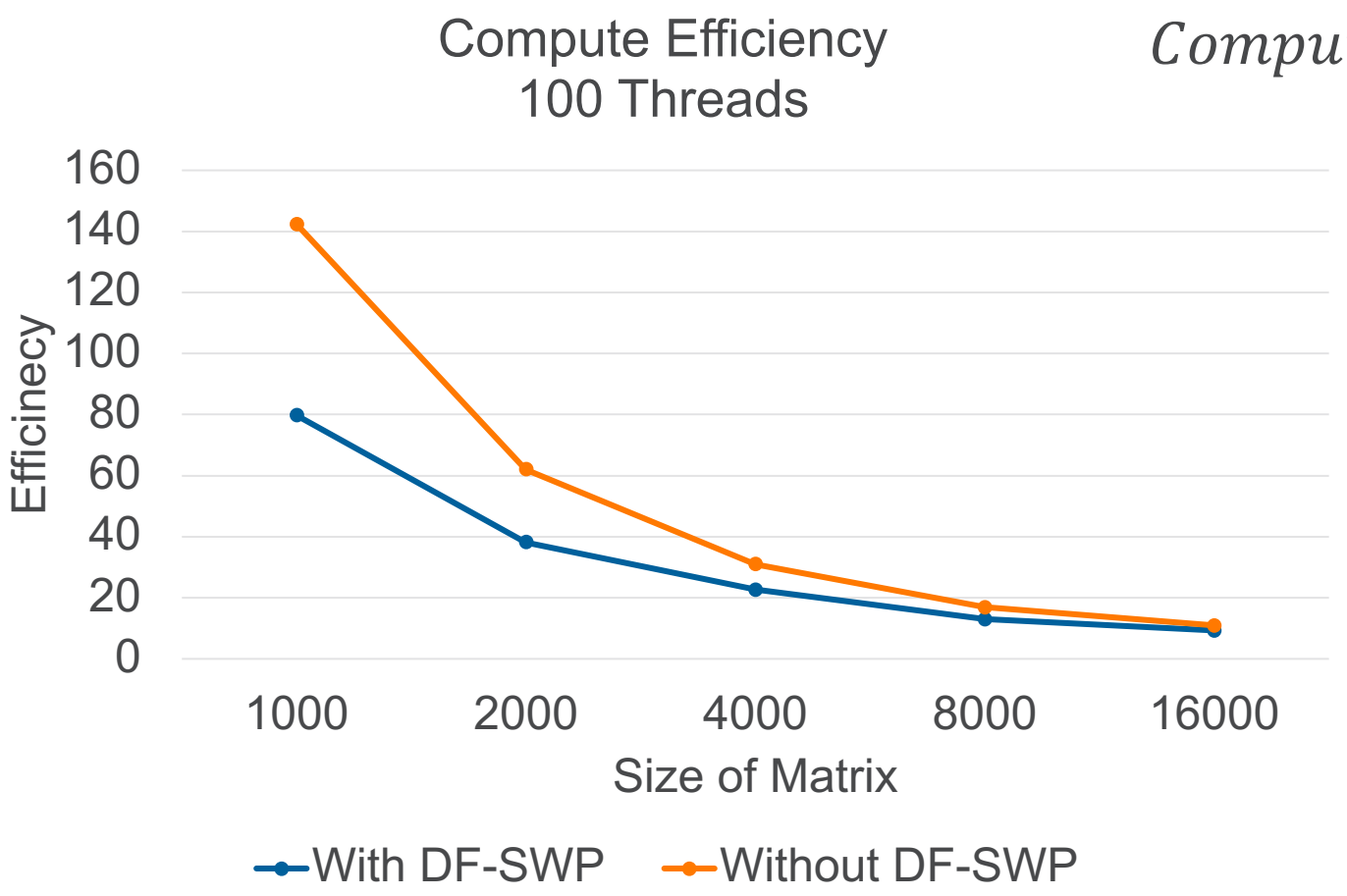
Relative Speedup



$$Speedup_{relative} = \frac{Time_{with\ dfswp}}{Time_{without\ dfswp}}$$

Relative Speedup of $1.4x$ is achieved with dataflow software pipeline enabled.

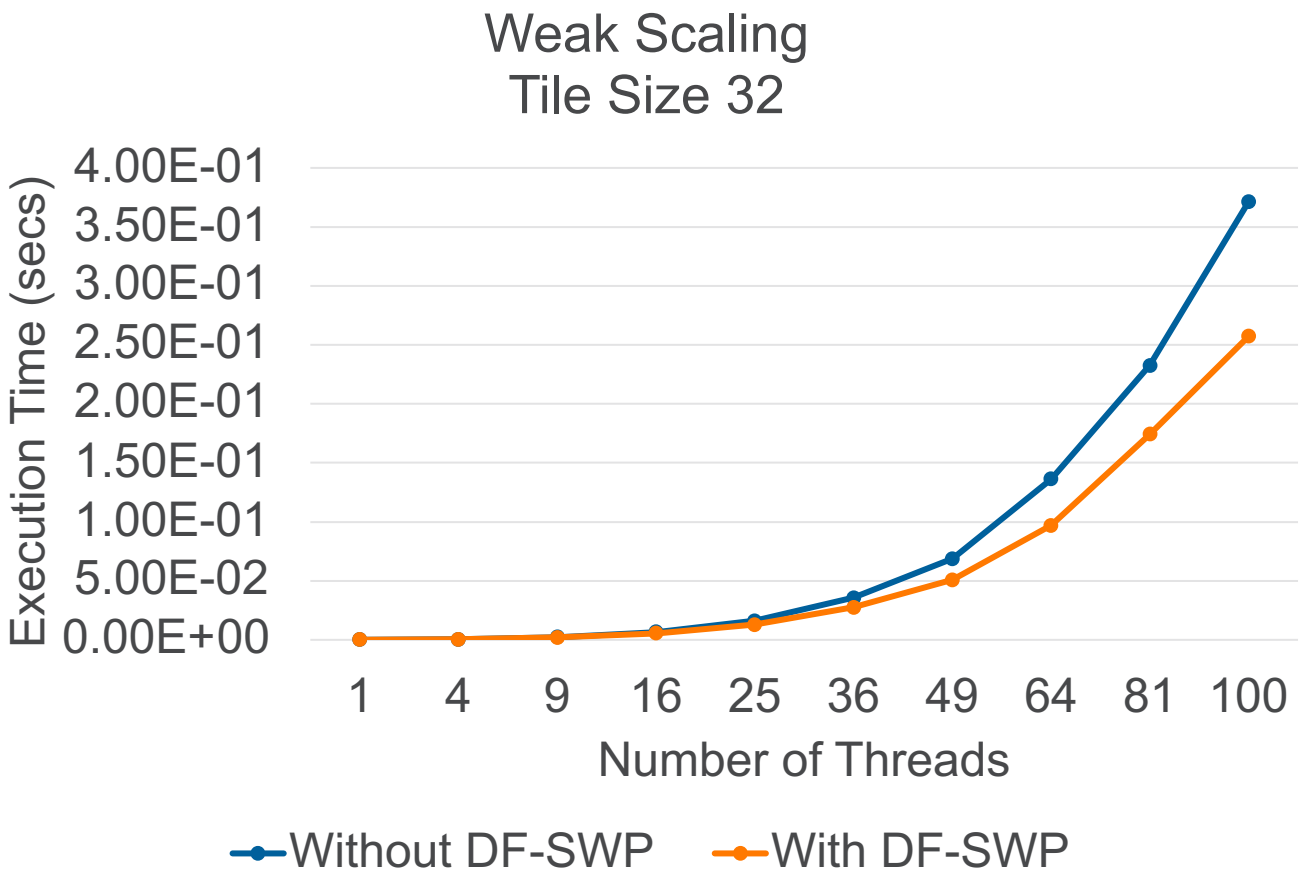
Compute Efficiency



$$ComputeEfficiency = \frac{Number\ of\ Threads}{Speedup}$$

Better compute efficiency is observed with dataflow software pipelining enabled.

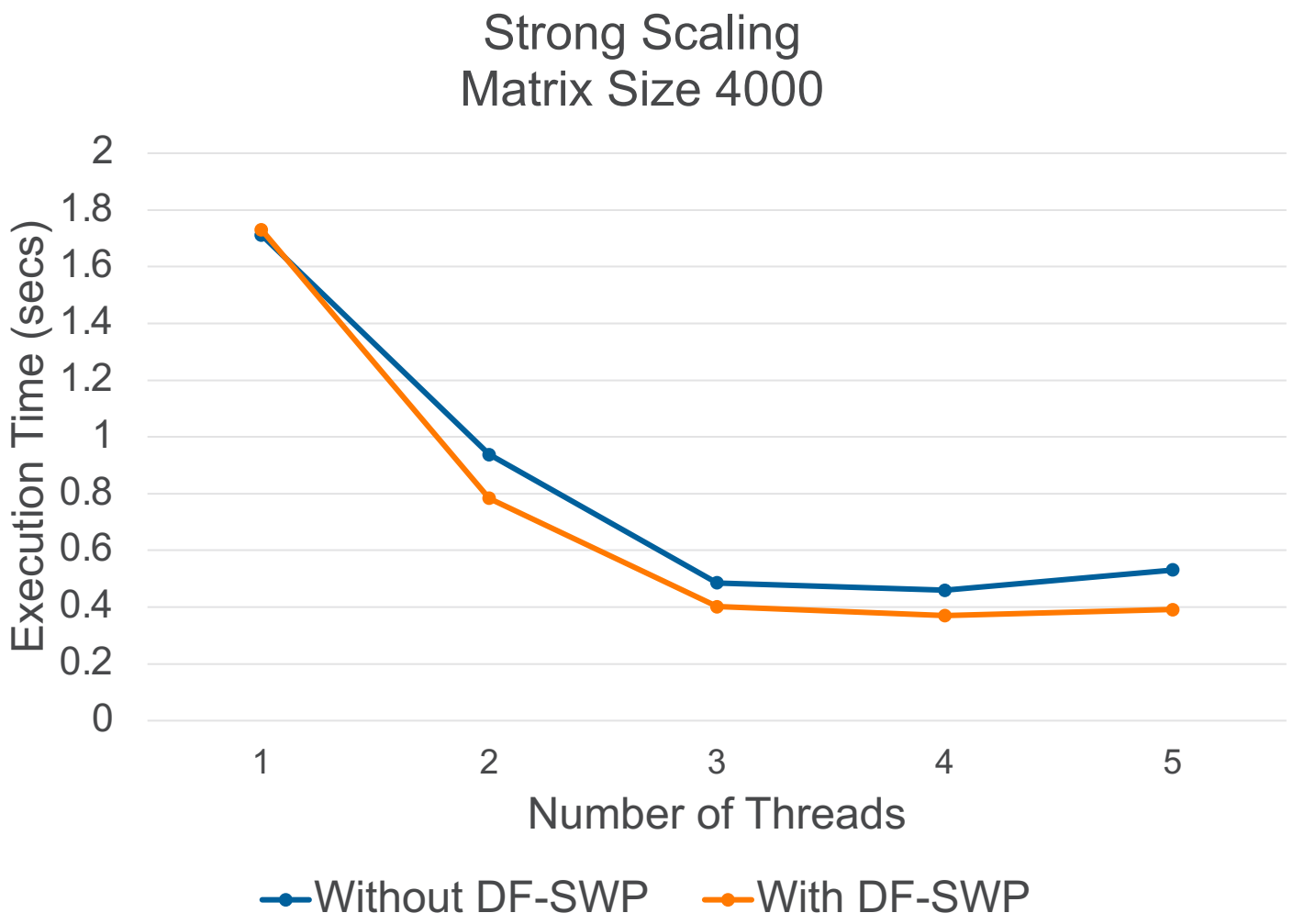
Weak Scaling



the problem size assigned to each processing element stays constant and additional elements are used to solve a larger total problem.

We observe consistent better results with dataflow software pipelining

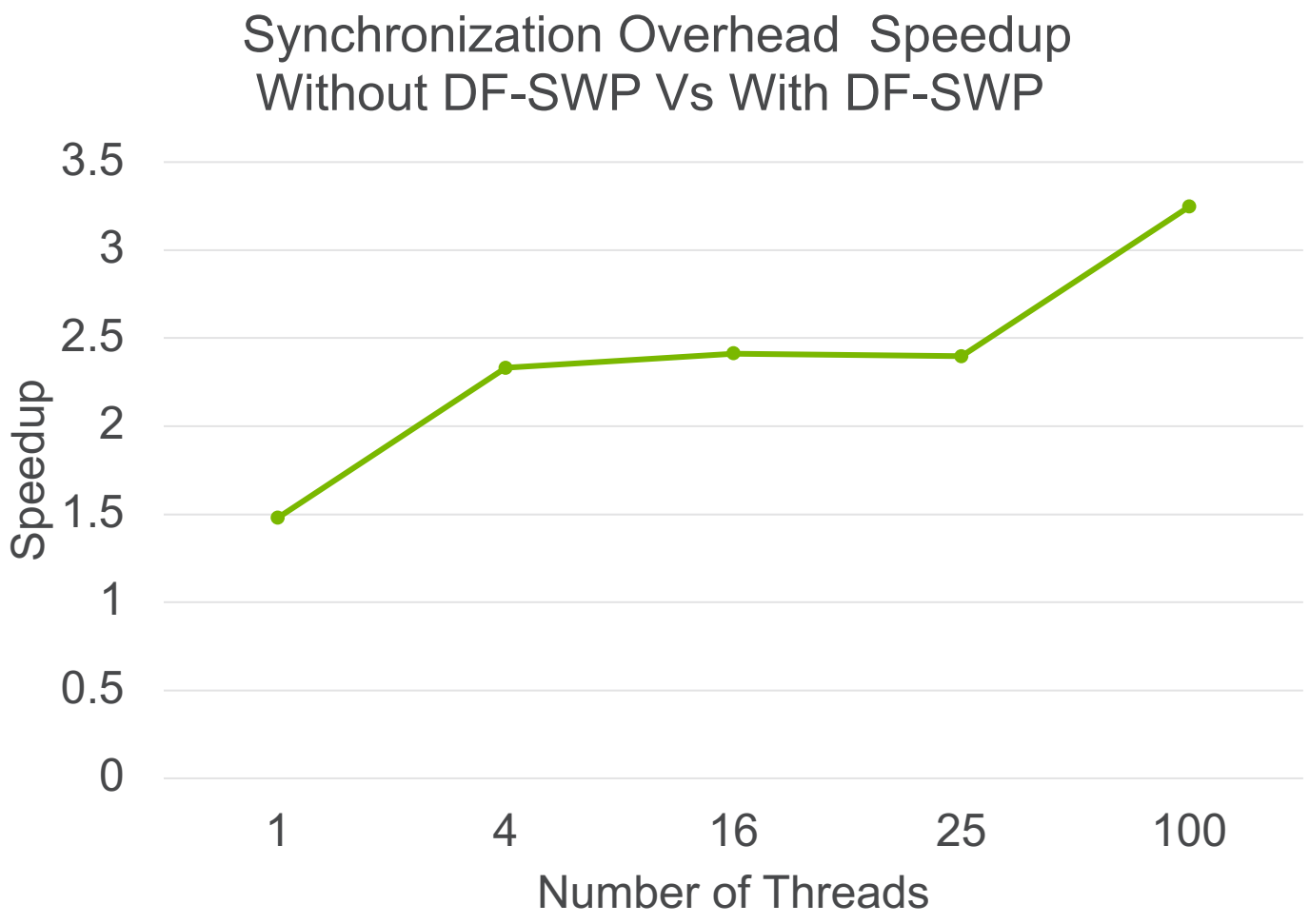
Strong Scaling



the problem size assigned to each processing element stays constant and additional elements are used to solve a larger total problem.

We observe consistent better results with dataflow software pipelining

Synchronization Overhead



synchronization overhead by omitting time consumed by compute codelets from the total execution time

Synchronization overhead decreases with dataflow software pipelining enabled.

The best speedup of $3.2x$ is observed for high thread counts of 100.

Agenda

- Motivation & Background
- Problem Formulation
- Solution Methodology
- Cannons Algorithm Case Study
- Experimental Evaluation
- Future Work & Conclusions

Future Work

- Fully exploit the potential of Dataflow Software Pipelining techniques is to explore hardware-software co-design techniques.

- hardware architectures that support features like programmer addressable fast scratchpad memory which can be used to implement FIFO Buffers while taking advantage of locality
  - Cerebras CS-2
  - Intel X^e GPU
  - Graphcore IPU

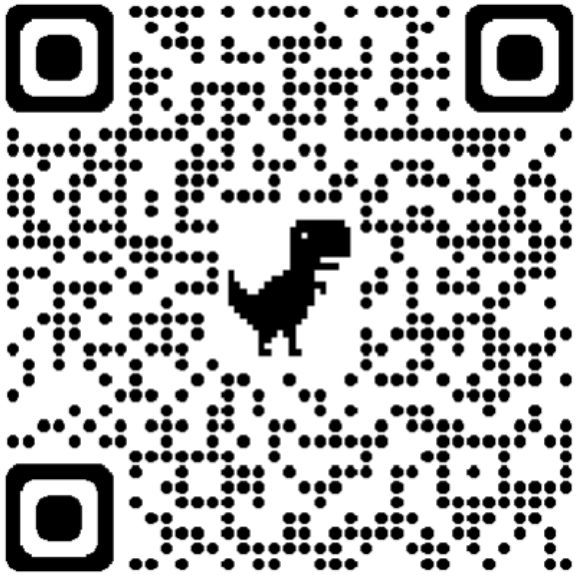
Conclusion

- ✓ Extend software pipeline techniques to the coarse grain to exploit pipelined parallelism across loops.
- ✓ Extensions to the dataflow-based Codelet Model to efficiently support dataflow software pipelining.
- ✓ Detailed case study of Cannon's algorithm

- Lays strong groundwork for research in this direction in the era of many-core architectures
- Using proved techniques in the traditional single-core architecture era

Thank You

Github Repo: Balancing Techniques
https://github.com/sraskar/cannon-dfswp



Sid Raskar
sraskar@anl.gov

Acknowledgements: This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. We gratefully acknowledge the assistance provided by the Argonne Computational Scientists. This work is partially supported by the National Science Foundation, under award SHF-1763654.