Implementation of Dataflow Software Pipelining for Codelet Model

Siddhisanket Raskar
Argonne National Laboratory
Lemont, IL, USA
sraskar@anl.gov

Jose M Monsalve Diaz
Argonne National Laboratory
Lemont, IL, USA
jmonsalvediaz@anl.gov

Thomas Applencourt
Argonne National Laboratory
Lemont, IL, USA
tapplencourt@anl.gov

Kalyan Kumaran
Argonne National Laboratory
Lemont, IL, USA
kumaran@anl.gov

Guang Gao
University of Delaware
Newark, DE, USA
ggao@udel.edu

ABSTRACT

Computer architectures have evolved from single core to chips with thousands of cores. Loop and instruction level parallelism techniques like software pipelining that are successful for single cores have limitations in the multi-core era. We extend the software pipelining technology beyond the limits of fine-grained, instruction-level parallelism. We accomplish this through dataflow software pipelining technology and its extension. Specifically, we present extensions to dataflow-based codelet model and its abstract machine to exploit pipelined parallelism across loops.

We extend the runtime implementation of the codelet model with our proposed extensions to take advantage of dataflow software pipelining principles using efficient single-owner First-In-First-Out (FIFO) buffer across Codelet's dependencies. We show promising improvements with the use of dataflow software pipelining techniques by performing an in-depth case study of Cannon's algorithm for matrix multiplication.

CCS CONCEPTS

• **Computer systems organization → Pipeline computing**; **Multicore architectures**.

KEYWORDS

Dataflow Model, Software Pipelining, Dataflow Software Pipelining, Codelet Model, Program Execution Model, Many-core architecture, FIFO buffers, Exa-scale

ACM Reference Format:
Siddhisanket Raskar, Jose M Monsalve Diaz, Thomas Applencourt, Kalyan Kumaran, and Guang Gao. 2023. Implementation of Dataflow Software Pipelining for Codelet Model. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3578244.3583734

1 INTRODUCTION

The supercomputing landscape has fundamentally changed in the past fifteen years [26]. Chips have evolved from single-core to multi-threaded, multi- or even many-core chips. Chip architectures are shifting from fewer, faster, functionally heavy cores to abundant, slower, simpler cores to address pressing physical limitations such as energy consumption and heat expenditure. To overcome these challenges, hardware architectures like GPU, FPGA, and domain-specific accelerators [5–8, 11, 30] that consist of multiple cores are becoming more common.

Software Pipelining is an important code mapping scheme to exploit pipelined parallelism in a loop. It has been successfully applied in compilers to exploit *Instruction Level Parallelism* (ILP) in a loop body, capable of scheduling 10s (or up to a couple hundred) machine instructions in pipelined execution. However, rapid advances in chip technology and computer architecture have enabled the design and production of chips with thousands of cores - some even reach hundreds of thousands of cores on a wafer [3]- far beyond the limit of ILP. An open challenge is determining if the software pipeline technology be extended and applied to meet such challenges.

Dataflow Software Pipelining is a code mapping technique to exploit *Instruction Level Parallelism* (fine-grain) to yield high throughput. However, exploiting pipelined parallelism across loops (coarse-grain parallelism) remains an open question under dataflow software pipelining. In this paper, we study how to apply dataflow software pipelining techniques to exploit large-scale parallelism beyond the limit of fine-grain instruction-level parallelism.

Today's chip architectures and their execution models are based on the sequential *Von Neumann* model dominated by data parallelism, while there has been little progress in parallelism via Software Pipelining techniques. The Codelet model is a hierarchical, multi-threading, event-driven, multi-grained model rooted in the dataflow model. It is possible to build upon the base Codelet Model to exploit asynchronous task parallelism through dataflow software pipelining techniques.

In this work, we address above mentioned open question by extending the software pipelining technology beyond its limit of fine-grained, instruction-level parallelism with the help of dataflow software pipelining technology and its extension. The major contributions of this work can be summarized as:

- We extend *software pipeline* techniques to the coarse grain to exploit pipelined parallelism across loops. This is accomplished by leveraging *dataflow software pipelining* principles, eliminating the limits of fine-grain parallelism.
- We propose extensions to the dataflow-based Codelet Model to efficiently support dataflow software pipelining. These extensions are implemented for a runtime based on Codelet model.
- We perform a detailed case study of Cannon's algorithm to demonstrate the effectiveness of dataflow software pipelining techniques and lay the groundwork for the seminal work in this direction.

The remainder of this paper is organized as follows: Section 2 discusses the background terminology based on which the work in the paper is based. Section 3 uses a motivating example to describe our inspiration behind this work and formally defines the scope of the problem this work targets to solve. Section 4 introduces the extension to codelet model to take advantage of principles of dataflow software pipelining. Section 5 walks through the Cannon's algorithm case study and its implementation details. Section 6 discusses the experimental results as well as their significance. Section 7 provided an overview of the related and inspirational work behind current efforts, while section 8 discusses some of our ideas on the future directions and next steps. Finally, we conclude our discussion in section 9.

2 BACKGROUND

In this section, we introduce basic terminology and point to essential references with an aim to provide essential background for the rest of this paper.

2.1 Software Pipelining

Software Pipelining [12, 28, 31, 42, 45] is one of the most important out-of-order, loop scheduling methods used by parallelizing compilers. It overlaps operations from various loop iterations in order to exploit instruction level parallelism. Effective software pipelining takes into account several constraints like instruction latency, resource availability, and register restrictions into account for the target architecture. Finding a code sequence that satisfies these constraints is NP-complete problem, a fact that has led to number heuristic techniques. The pioneering work for formulation of the software pipelining process for single basic block loops was stated by B. Ramakrishna Rau et al. [42]. The classical software pipelining survey work by Rau et. al. [41] and Allan Et al. [13] provide a good overview of the field.

Today, these pioneering techniques are standard in modern compilers optimizing execution for sequential code. In our work, we wish to leverage upon these pioneering techniques mentioned above as well as all the work that followed those techniques in the decades to come. We leverage fine-grained *Instruction Level Parallelism* at Codelet level (further details in section 3.2).

2.2 Dataflow Software Pipelining

Dataflow Software Pipelining [23] is a code mapping technique to generate code that can be executed in a pipelined fashion with high throughput. This is achieved by keeping the pipeline busy and computation balanced [22]. Special dataflow ID nodes [14] or *FIFO buffers* [24, 27] are used to optimally balance the dataflow graph for maximum pipelining [22]. The minimum buffer allocation and scheduling aspects of the dataflow software pipelining is an NP-complete problem. The seminal work by Ning[36] shows that one can allocate the minimum number of buffers to variables in polynomial time under an extended static dataflow execution and architecture model.

We extend dataflow software pipelining techniques to exploit coarse-grained pipelined parallelism across codelets (further details in section 4.1)

2.3 Codelet Model

Codelet model [21, 48] is a hierarchical, multi-threaded, event-driven, multi-grained, hybrid von Neumann-dataflow execution model designed for extreme-scale systems in mind that draws its roots from the dataflow model. The Codelet execution model is designed to leverage previous knowledge of parallelism, and to develop a methodology for exploiting parallelism for a much larger scale machine. Codelet model's roots in a dataflow model provides several important features including fine-grain synchronization, functional programming, composability, and determinate execution.

A Codelet (CD) consists of a sequence of machine instructions that execute non-preemptively and atomically. Codelets are linked together to form a Codelet Graph (CDG). In a CDG, each codelet represents a producer and/or consumer while the edge in between them represents an event. A Codelet is scheduled upon availability of specific resources, the primary one being data. Codelets may be grouped into Threaded Procedure (TP). A TP is an asynchronous function that acts as a container for a CDG and the data accessed by its codelets giving sense of locality to codelets within a TP.

The Codelet Model relies upon its Codelet Abstract Machine (CAM) which is mapped at runtime to the target many-core system. Cores can be either Synchronization Unit (SU) or Compute Unit (CU). A SU is in charge of allocating TPs and scheduling codelets to CUs.

2.4 Delaware Adaptive Runtime System

Delaware Adaptive Runtime System (DARTS) [29, 43] is currently the most faithful runtime implementation of the Codelet Model and its Abstract Machine Model (AMM). A runtime provides flexibility in satisfying a particular execution model's requirements by bridging the gap between software and hardware. DARTS provides two levels of parallelism, event-driven codelets permitting fine-grained parallelism and invoked threaded procedures which ensures locality.

A programmer uses DARTS runtime interface to define a CDG comprised of CDs and grouped into TPs. TP and CD are defined as class objects at compile time and are instantiated and scheduled at runtime. A Programmer sets the mapping between the CAM and the available machine hardware through the DARTS AMM. DARTS is written for shared memory x86 architectures, written in C++ and distributed as free and open-source software [1].

3 MOTIVATION AND PROBLEM FORMULATION

Dataflow Software Pipelining for codelet graphs is a broad field of study that covers various techniques applicable to the general class of codelet graphs. Section 3.1 provides a simple motivating example to highlight the importance and need of dataflow software pipelining for codelet model. In section 3.2, we define the class of codelet graphs for which the dataflow software pipelining techniques discussed in this paper will be applicable.

3.1 Motivating Example

Consider a scenario such as that presented in Figure 1a. Here we depict a part of CDG, where two codelets are in a producer-consumer relation. `C1` is *producer* codelet, producing array `A` of size `N`, while `C2` is *consumer* codelet which is consuming array `A` and writing results to array `B` of size `N`.

Let us assume:

- System has sufficient resources to schedule codelets on different CUs.
- SU schedules `C1` & `C2` on separate CUs.
- Each iteration of loops `L1` and `L2` take 1 unit time.
- Sending signal/ data from `C1` to `C2` takes 1 unit time.

Now, consider execution scenarios with and without dataflow software pipelining for codelet model.

Scenario 1: Under original Codelet Model, execution will proceed as follows:

- `C1` will receive a signal and start its execution.
- `C1` will completely finish executing its code by performing the `N` iterations of the loop.
- `C1` signals `C2` that data is available. `C2` will receive a signal and start its execution.
- `C2` will completely finish executing its code by performing `N` iterations of the loop.

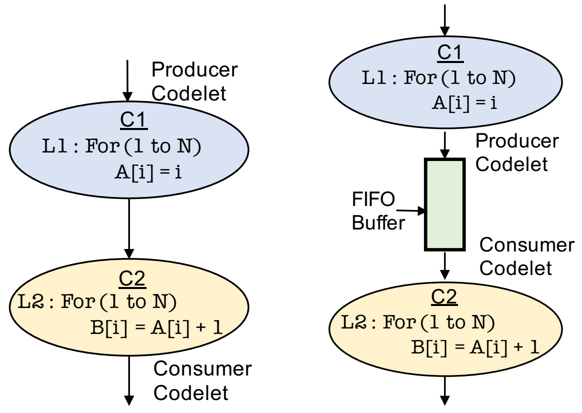
In this case, we can see that both `C1` and `C2` each will take `N` units of time. The total execution time in this scenario will be at least `N+N+1` unit time.

Scenario 2: Assuming perfect Dataflow Software Pipelining for the Codelet Model, execution will proceed as follows:

- `C1` will receive a signal and start its execution.
- `C1` will send signal to `C2` once it finishes the 1st iteration of loop `L1`.
- `C2` will begin execution of loop `L2` while `C1` continues execution of loop `L1`.

In this case, `C1` & `C2` each take `N` units of time—same as scenario 1 above. Since, we allow execution of `C2` to begin before the execution of `C1` finishes, the minimum total execution time for this case will be `N+1` units time for consumer codelet to wait until first iteration of producer codelet finishes.

The producer-consumer behavior explained in this example is quite common in scientific applications. This example shows the importance of dataflow software pipelining for a simple case with `N` iterations in loop. However, one can easily see that this technique will also be very useful for streaming applications where input or output data can potentially be infinite (e.g. $N \to \infty$).



(a) Codelets **(b) Codelets with FIFO**

Figure 1: Producer-Consumer Codelets in CDG

3.2 Problem Formulation

Traditional software pipelining techniques are mostly static, compile-time, loop-optimization techniques, while dataflow software pipelining techniques are dynamic (due to nature of dataflow graphs) that go beyond just loop optimization and consider the entire dataflow graph for optimization opportunities. The high-level objective to achieve through this formulation is to develop a Program Execution Model (PXM) which can leverage coarse grain parallelism at the CDG level and fine grain parallelism at the CD level by exploiting traditional software pipelining.

With these high-level goals in mind, we categorize our formulation at two levels:

1. Codelet Level: Our intentions are to leverage decades of work done in the field of *Instruction Level Parallelism* and *Software Pipelining* to exploit performance at the codelet level.

To achieve this, we will be restricting the loops inside our codelets to the class of loops which are *analyzable* by traditional software pipelining and loop optimization techniques. Primarily, this means the dependencies inside the loop should be known at compile time. For the sake of further simplicity, at this stage of formulation, we will also be restricting the loop dependencies to simple *affine* functions—functions composed of a linear function and a constant, taking the form:

$$Y = u \times i + v$$

Furthermore, we will be restricting the variable values in affine function. Here are further details:

- Restrictions on ***u:*** We restrict u to only 1. This is to ensure a single stride access to the array in both, the producer and the consumer.
- Restrictions on ***i:*** We restrict i to only 1. This is to ensure simple access patterns inside array (prohibit access of type $\texttt{i}^2$ or `3i`).
- Restrictions on ***v:*** We restrict v to positive values. This will ensure only forward access inside an array.

The aforementioned formulation and restrictions described in this section ensure constant and uniform production and consumption of tokens between consecutive codelets. These simplifications allows us to focus on dataflow software pipelining in between codelets.

2. Codelet Graph Level: At this level, our intentions are to exploit coarse grain parallelism in the codelet graph. We will be restricting our formulation to class of CDG that satisfy the following conditions -

- Class of Codelet graphs that are *Directed Acyclic Graph* (DAG).
- Each node (codelet) in the DAG is a "nice loop" meaning body is a simple statement and the index expression for the loop is a simple affine function of the index, as explained above.

To further clarify, please note that we are not allowing cycles only at the coarse grain, codelet graph level. Cycles inside any codelet are permitted as long as they follow these restrictions and can be optimized by traditional software pipelining techniques.

4 EXTENSIONS TO ENABLE DATAFLOW SOFTWARE PIPELINING FOR CODELET MODEL

In this section, we will cover some ideas and the underlying thought process to enable dataflow software pipelining for codelet model.

4.1 Extensions to Codelet Model

4.1.1 Extension to Codelet Program Execution Model (PXM). The *Original Codelet Model* semantics do not allow the *consumer* codelet to begin its execution before the output tokens are generated by the *producer* codelet. In the case of codelets with loop, the *consumer* codelet has to wait until all loop iterations of the *producer* codelet finish and produce a data token (similar to our motivating example in section 3.2). Under certain situations, we need to allow the consumer codelet to begin its execution before the producer codelet has finished its entire execution to allow dataflow software pipelining in the codelet model.

The primary challenge to enable dataflow software pipelining for the codelet model is to synchronize between codelets. We need a mechanism to inform the *consumer* codelet when to start its execution after the *producer* codelet has finished part of its execution and it has produced relevant data tokens needed for the *consumer* codelet. Also, we need to ensure order of data tokens in-between producer and consumer codelets.

This can be achieved by introducing a new type of event in the PXM. One of the possible implementation can be achieved using 2-bit flag and flip-flop-like logic in shared memory systems. However, such approach could lead to inefficient implementations due to the overhead carried by shared memory systems, especially in NUMA(Non-uniform memory access) systems. Hence we have extended our CAM to support this mechanism efficiently.

4.1.2 Extension to Codelet Abstract Machine Model (CAM). Extending the PXM is just the first step. In order to efficiently support Dataflow Software Pipelining extension discussed above, they need to be supported in the Abstract Machine as well in order to exploit optimal performance.

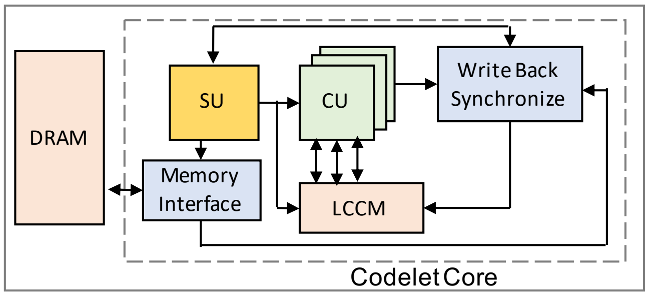


Figure 2: Extended Codelet Abstract Machine Model

To accomplish this, there have been on-going efforts [17, 18] to extend the Codelet Model. Figure 2 shows the portion of the Extended Codelet Abstract Machine (xCAM) most relevant for this paper. The main differences between the original CAM and xCAM can be summarized as follows:

- *Codelet Core* is the most relevant part to focus on to understand mechanism needed to enable dataflow software pipelining for codelet model.
- *Local Codelet Core Memory (LCCM)* is inside the Codelet Core and uses smart implementation based on software-hardware co-design. We envision its implementation as an extended scratchpad memory or lower level cache with FIFO queue-like capabilities.
- Memory sub-system has been moved out of *Codelet Core* to hide memory operation latency. We envision its implementation in architecture as a higher-level cache or SDRAM.
- CU is now part of Codelet Core and multiple CUs can part of it. CU can be mapped to CPU or GPU cores, or other heterogeneous hardware in the architecture.

The incorporation of LCCM in the extended codelet model plays a vital role in enabling software pipelining between codelets. LCCM is shared between different CUs on the same codelet core to explicitly share frequently needed data. This avoids long latency memory operation trips to the main memory and hence improves the overall performance. Codelets scheduled on the same codelet core can use LCCM as FIFO buffers to enable dataflow software pipelining in-between them. The extended codelet model can make intelligent use of LCCM to support dataflow software pipelining of codelets.

4.2 FIFO Buffers

One of the most intuitive methods to store intermediate results and ensure order of data tokens in-between two computation units, especially when we have single producer and single consumer, is to use temporary storage units like **FIFO** buffers. They can provide a highly efficient data communication channel without relying on operating system constructs such as semaphores, mutexes, or monitors for data transfer.

In figure 1b, we show conceptual visualization of a FIFO channel between *producer* and *consumer* codelets using same example from section 3.2. The execution scenario with a FIFO buffer can be summarized as:

- Both C1 & C2 will receive a signal and are enabled for execution. However, only C1 will start its execution as only data needed for C1 is available.
- After first iteration of loop L1, the resulting data token will be generated and stored at the head of FIFO buffer.
- C2 will receive this data token from FIFO buffer and start its execution.

5 IMPLEMENTATION OF DATAFLOW SOFTWARE PIPELINING FOR CODELET MODEL

In this section, we offer details of how we implement dataflow software pipelining extensions in the Codelet Model. We first provide a brief background on Cannon's Algorithm in section 5.1. Following this, section 5.2 discusses mapping and implementation of Cannon's algorithm for the Codelet Model. Section 5.3 is a comparitive discussion of how we extend the implementation in section 5.2 to leverage Dataflow Software Pipelining techniques using FIFO Buffers.

5.1 Cannon's Algorithm

Cannon's Algorithm is a distributed algorithm for matrix multiplication of two-dimensional meshes, first described in 1969 by Lynn Elliot Cannon [16] and suitable for computers laid out in an N × N mesh. The Cannon's algorithm assumes availability of P processes for matrix multiplication, where P is square number. A & B are input matrix while C is the resultant matrix. All A,B & C are square matrices decomposed into block matrices and mapped to processes. The main advantage of the algorithm is that its storage requirements remain constant and are independent of the number of processors.

Earlier in section 3.2, we formulated certain constraints at the Codelet Graph level as well as at the Codelet level to define an important class of Codelet programs where pipelined parallelism can be efficiently exploited across loops by Dataflow Software Pipelining techniques. The way we map Cannon's algorithm to Codelet model satisfies our constraints at the Codelet Graph level, as the CDG for Cannons algorithm is Directed Acyclic Graph. Inside each codelet, a matrix multiplication operation is followed by a shifting operation which can take advantage of classical software pipelining.

The Cannon's algorithm is described in Algorithm 1. It is not the intent of this work to analyze and optimize the steps in the original Cannon's algorithm. The implementations discussed in section 5.2 and 5.3 stay faithful to the original algorithm. We will not go into a detailed discussion of Cannon's algorithm here, since that is not the primary purpose of the discussion. Instead, the main steps of Cannon's algorithm relevant to this work can be summarized as follow:

Algorithm 1: Cannon's Algorithm for Matrix Multiplication

```
Data: Two Square Input Matrices
Result: Square Resultant Matrix
Initialization- Skew Matrices
for $i \leftarrow 0$ to $N-1$ do
    Left circular shift row i by i,
    so that A(i,j) is assigned A(i, (j+i) mod N);
end
for $j \leftarrow 0$ to $N-1$ do
    Upward circular shift column j by j,
    so that B(i,j) is assigned B((j+i)mod N, j);
end
Computation & Communication
for $k \leftarrow 0$ to $N$ do
    for $i \leftarrow 0$ to $N-1$ do
        for $j \leftarrow 0$ to $N-1$ do
            C(i,j) = C(i,j) + A(i,j) * B(i,j) ;
            Left circular shift each row of A by 1,
            so A(i,j) is assigned A(i,(j+1)modN) ;
            Upward circular shift each column of B by 1,
            so B(i,j) is assinged B((i+1)modN,j) ;
        end
    end
end
```

(1) **Initial Skew Step:** Initialize Matrix A & B by performing left and upward *circular shift* in order to align matrices for multiplication.
(2) **Computation Step:** Each process P multiplies its local submatrices and partial result C is calculated.
(3) **Communication Step:** Each process *circular shifts* its submatrix of A & B to the left by 1 step and all sub-matrices to the up by 1 step. But, the result matrix C does not move.
(4) Repeat step two and three for square root of *P* times.

When the above steps are complete, each process holds resultant sub-matrix *C*.

At this stage of discussion, we would like to point out that, the implementation of Dataflow Software Pipelining techniques discussed later in detail in section 5.3 achieve performance gains by optimizations in-between Computation (Step 2) and Communication (Step3) phase. This is achieved by overlapping these two steps and storing intermediate results in the FIFO buffers for all processes.

5.2 Cannon's Algorithm without Dataflow Software Pipelining under Codelet Model

Figure 3b shows the CDG for the Cannon's algorithm without dataflow software pipelining (DF-SWP). We divide all matrices among P tiles and associate them to P codelets similar to algorithm 1. Here P is the number of tiles in which matrices are divided. The functionality of various codelets can be summarized as follows:

- **CopyA and CopyB:** These codelets copy original matrix A and B to tile memory local to each codelet. P instances of these codelets are created. Matrices are copied to tiles for locality reasons.
- **Skew:** These codelets skew/initialize matrix A and B as described in the skew step of algorithm 1. P instances of these codelets are created.

- **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.

There is communication between the `Skew` phase and `Compute` phase taking place inside this Codelet Graph. To further illustrate this, please refer to Figure 3a, which shows these communications as well as how matrices are divided and how codelets communicate with the simple 3x3 tiles. For example:

- Each tile consists of blocks for sub-matrix `A`, `B` and `C`. Each tile also contains blocks `Aw` and `Bw` which are used to write sub-matrix from neighboring tiles in the skew and shifting steps.
- The blue arrows show the data movement during the `skew` phase while red arrows show the data movement during the `compute` stage.
- To avoid congestion of communication arrows, we only show skew phase communication for diagonal tiles highlighted with darker gray. However, those communications are carried out by all tiles.

There are four stages for Cannon's algorithm implementation under Codelet Model. For further clarification, we map these stages from Codelet Graph to our sample 3x3 tiles example. The numerical representation of stages (like ❶,❷,❸,❹) in Figure 3b correspond to those in Figure 3a.

- **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 performs 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).

5.3 Cannon's Algorithm with Dataflow Software Pipelining under Codelet Model

In this section, we will talk about a peculiar structure of Cannon's algorithm and the opportunities it provides to map it to the Codelet Model to exploit pipelined parallelism across loops (Codelets). Figure 4b shows the CDG for the Cannon's algorithm with dataflow software pipelining (DF-SWP), while Figure 4a shows the sample 3x3 tiles example. These two figures are very synonymous with their counterparts in the earlier section 5.2 and follow similar logistics. Instead of explaining those all again, here we focus on the key differences between two approaches:

- **FIFO Buffers:** The main difference between Codelet Graph of implementation without dataflow software pipelining (figure 3b) and that of with dataflow software pipelining (figure 4b) is the removal of `loop` codelet which acted as a barrier between various iterations of `compute` codelet. However, such a barrier is not needed anymore. Since we use **single owner FIFO buffers** in between these various iterations of codelets. Once the Compute codelet calculates its result, it can simply `Push` it the FIFO buffer and continue with its next iterations without waiting for the barrier. Similarly, the next iteration of compute codelet does not need to wait for the barrier as it can simply `Pop` its input from the FIFO buffer and continue with its execution.
- To accomplish this, sub-matrix blocks for `A` and `B` are replaces with **FIFO buffers** for respective matrices in figure 4a.
- The other difference is addition of explicit `Barrier` codelet to act as synchronization point between `skew` and `compute` phases. This is necessary as `loop` codelet was acting as this synchronization barrier in the implementation of figure 3b. Since we do not need `loop` codelet anymore, we need a mechanism to make sure `skew` phase finishes completely and matrices are in the correct order before `compute` codelets can start execution in order to avoid data races.

6 EXPERIMENTAL RESULTS

In this section, we present the results of the runtime implementation of dataflow software pipelining extensions to codelet model. To being, in section 6.1, we summarize the important results and observations. The section 6.2 overviews the hardware as well as software parameters for our experimental setup. The sections 6.3 and 6.4 go into the details of each observation made in the section 6.1. Finally, we discuss the significance of these results and draw conclusions in section 6.5.

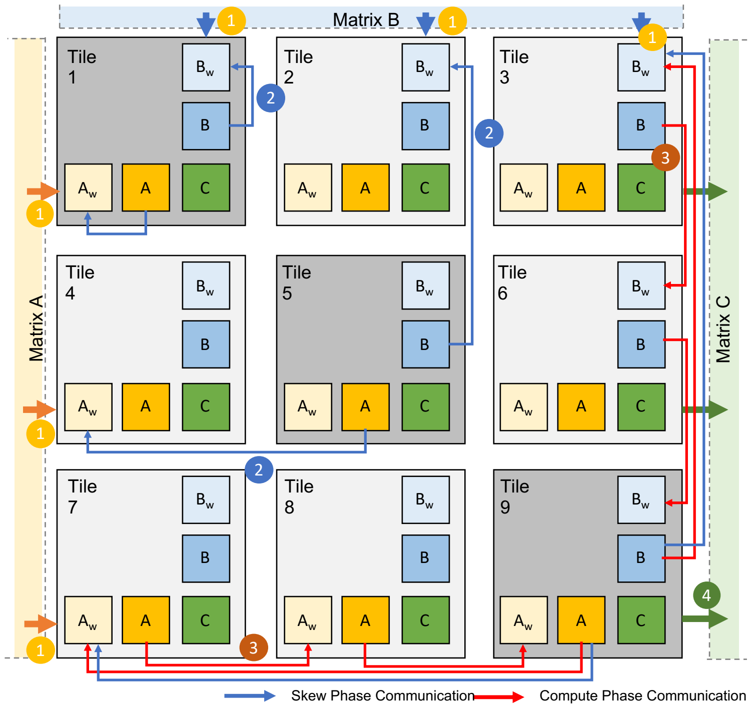
6.1 A Summary of Results

Here are the major observations from our experimental results -

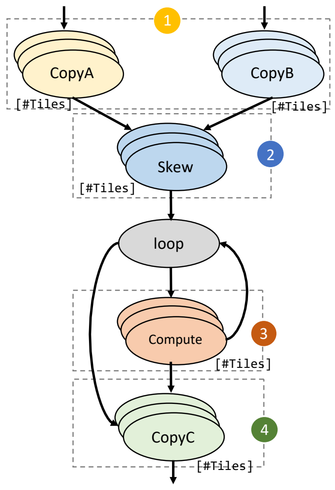
- **Observation 1 (Refer Section 6.3):** Relative Speedup of 1.4x is achieved with dataflow software pipeline enabled. In general, the speedup increases with the increase in the core count. Performance is negatively affected when codelets are mapped to a different socket, and later when hyper-threading comes into effect.
- **Observation 2 (Refer Section 6.4):** Better compute efficiency is observed with dataflow software pipelining enabled. As the size of matrix increases, a much better compute efficiency is observed.
- **Observation 3 (Refer Section 6.4):** Synchronization overhead decreases with dataflow software pipelining enabled. The best speedup of 3.2x is observed for high thread counts of 100.

6.2 Experimental Setup

6.2.1 Experimental Testbed. We performed our experiments on an Intel-based shared memory machine. All cores feature 32KB private L1 instruction and data caches, and 1MB private unified L2 caches. Each node has two sockets, with 28 cores per socket, featuring Intel

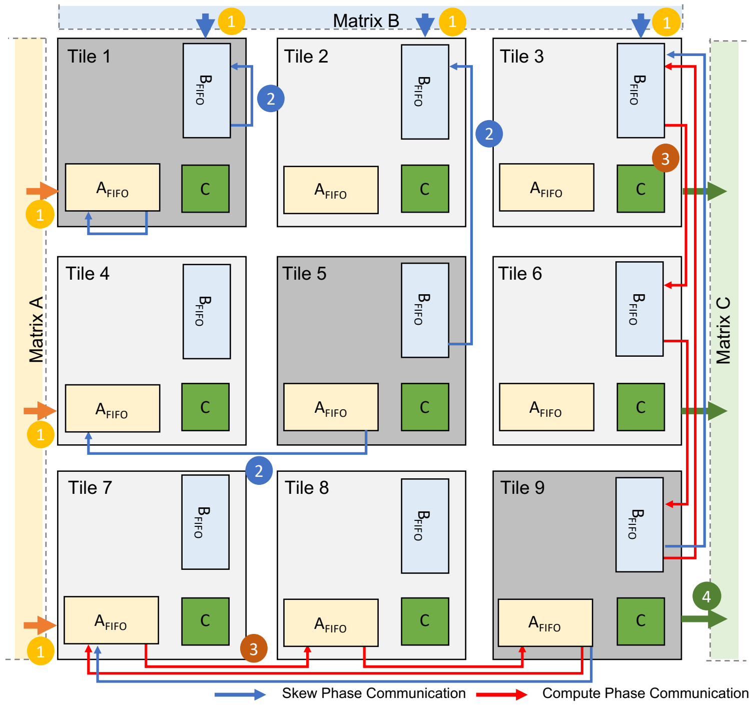


(a) Matrix tiling & Communication

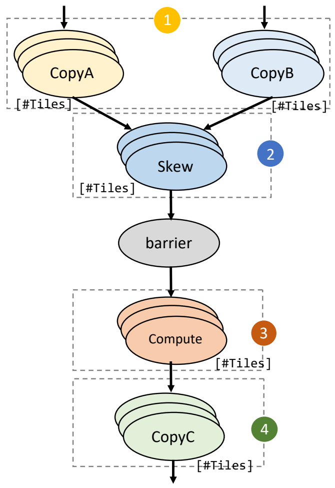


(b) CDG `without DF-SWP`

Figure 3: Cannon's Algorithm Implementation: Without Dataflow Software Pipelining



(a) Matrix tiling & Communication



(b) CDG `with DF-SWP`

Figure 4: Cannon's Algorithm Implementation: With Dataflow Software Pipelining

Xeon Platinum 8180M (Skylake) processor clocked at 2.5GHz with Hyper-Threading (HT). A 38.5*MB* unified L3 cache is shared by all the cores in the same socket. The total memory is 383*GB* of DRAM divided into two NUMA domains. The system runs Red Hat Enterprise Linux 7.5 with Linux kernel 3.10. All experiments are compiled with GCC 8.2 with optimizations set to `-O3`.

6.2.2 Experimental Design. Results shown in the following sections were taken after performing 30 runs for each experiment. Unless otherwise specified, each result is the average calculated from those runs. No major changes or additional optimizations other than those discussed here were introduced in the source code of experiments in order to perform a fair comparison.

In order to study the effects of enabling dataflow software pipelining extensions, we map threads on separate cores until all cores were assigned at least one thread. We chose this in order to minimize the effects of Hyper-Threading (HT) on our results. To achieve this, we use `KMP_AFFINITY` parameter and set it to `BALANCED` with granularity as `CORE`. We 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). In addition, we used `Work Stealing` scheduling policy for TP scheduler as well as Micro-scheduler.

As explained earlier, our current implementation of Cannon's algorithm is restricted to only square matrices. Each element of matrix is of type `double` with 8 bytes. The source code for experiments is publicly accessible in a repository [9, 10].

6.2.3 Terminology. In section 6.3, we compared Matrix multiplication performance of various techniques. Here is brief information about each of those techniques:

- **Without DF-SWP:** We map Cannon's algorithm to codelet model and implement it in dataflow-based runtime DARTS [43]. This implementation uses a loop codelet as a barrier between iterations of compute codelets as explained in the section 5.2. We refer to this implementation as `Without DF-SWP` in our experimental evaluation. This is our baseline implementation.
- **With DF-SWP:** We extend the above baseline implementation with dataflow software pipelining by using FIFO buffers as explained in the section 5.3. We refer to this implementation `With DF-SWP` in our experimental evaluation.

6.3 Performance Evaluation

In this section, we discuss the experiments performed to quantify the performance gains of dataflow software pipelining extensions.

6.3.1 Relative Speedup. The *Speedup* is defined as the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on P processors. We calculate the *Relative Speedup* as the ratio of the execution time of `With DF-SWP` implementation to that of `Without DF-SWP` implementation.

$$Speedup_{relative} = \frac{Time_{WithDF-SWP}}{Time_{WithoutDF-SWP}}$$

Figure 5 shows the relative speedup achieved with the use of dataflow software pipelining extensions compared to the implementation that does not use dataflow software pipelining extensions. We ran a Matrix-Matrix multiplication experiment with a tile size of 64 per codelet/thread while using increasing the number of threads/codelets. We achieved a maximum speedup of 1.4*x* for a thread count of 64. In general, the speedup increases with the increase in core count. We see a drop in speedup at thread count 26 as execution starts using the second socket at that point. We see another drop in speedup after thread count 64 as hyper-threading kicks in and multiple codelets get scheduled on the same physical cores.

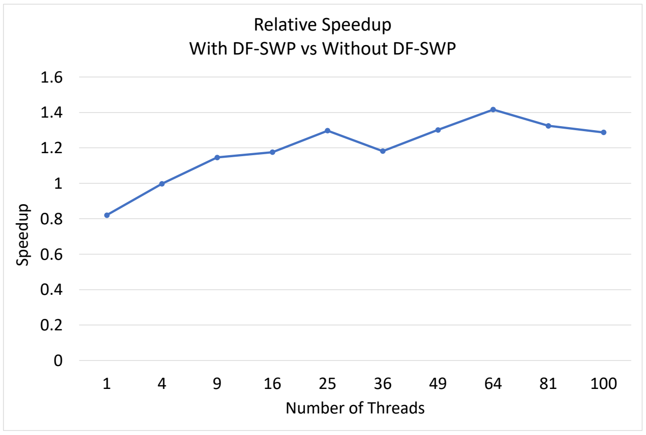


Figure 5: Relative Speedup of `With DF-SWP` implementation

6.3.2 Compute Efficiency. efficiency is a performance metric related to speedup. Speedup is a metric to determine how much faster parallel execution is, while efficiency indicates how well software utilizes the computational resources of the system. To calculate the efficiency of parallel execution, we take the observed speedup and divide it by the number of threads used. This number is then expressed as a percentage.

$$ComputeEfficiency = \frac{NumberofThreads}{Speedup}$$

The figure 6 shows the compute efficiency of `Without DF-SWP` & `With DF-SWP` implementations plotted for increasing size of matrices with fixed 100 count of threads/ codelets. In general, the compute efficiency of both methods improves as the size of matrix increase. We can see that `With DF-SWP` implementation achieves better compute efficiency compared against `Without DF-SWP` method. As the size of the matrices increases, the results converge as the elapsed time is dominated by computation, and the relative synchronization cost is reduced.

6.4 Scalability Analysis

In this section, we talk about the experiments that we perform to quantify scalability achieved by enabling dataflow software pipelining extensions.

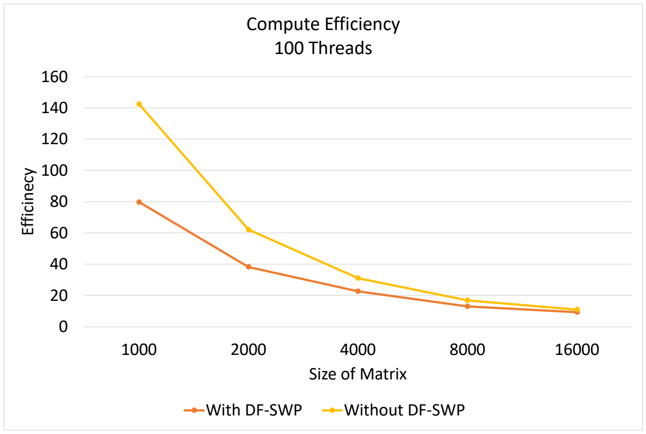


Figure 6: Compute Efficiency comparison of `Without DF-SWP` and `With DF-SWP` implementations

6.4.1 Weak Scaling. In *Weak Scaling*, the problem size assigned to each processing element stays constant and additional elements are used to solve a larger total problem. Figure 7 shows the weak scaling analysis of `Without DF-SWP` and `With DF-SWP` implementations. The tile size here is 32 for each matrix. We chose this size as both input matrices and output matrix fit into L1 cache for this size and tiles can use FIFO buffer using cache. In general, the weak scaling curve shows better results for `With DF-SWP` method.

6.4.2 Strong Scaling. In *Strong Scaling*, the problem size stays fixed but the number of processing elements is increased. Figure 8 shows the strong scaling analysis of `Without DF-SWP` & `With DF-SWP`. The size of matrices is 4000 for this study. We chose this size since it's the smallest matrix size that doesn't fit in the L3 cache. In general, a strong scaling curve shows better results with `With DF-SWP` implementation.

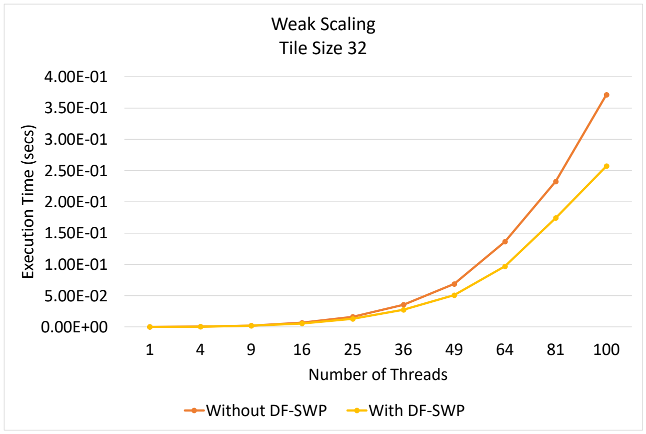


Figure 7: Weak Scaling analysis of Barrier and FIFO Buffer methods for tile size 32 per thread/Codelet

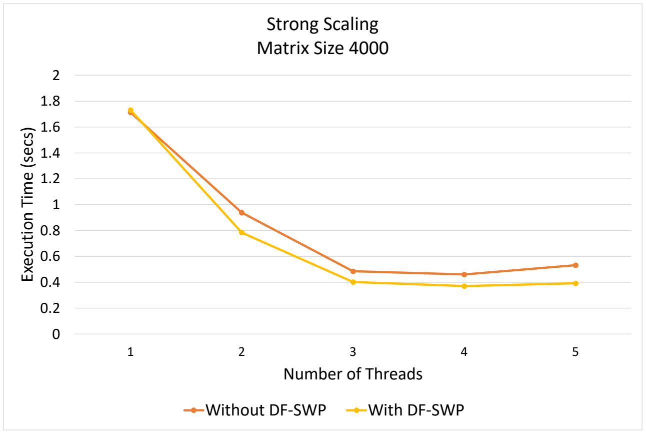


Figure 8: Strong Scaling analysis of Barrier and FIFO Buffer methods for Matrix of Size 4000

6.4.3 Synchronization Overhead. Figure 9 shows the synchronization overhead analysis of `Without DF-SWP` and `With DF-SWP` implementations. We compute synchronization overhead by omitting time consumed by compute codelets from the total execution time. The synchronization overhead tells us about the time consumed by memory operations and synchronization between codelets. These times typically grow as the number of compute units increase. We achieve up to a 3.2*x* time decrease in synchronization overhead when using `FIFO Buffers`. This also gives us insights into the weak and strong scaling results discussed earlier.

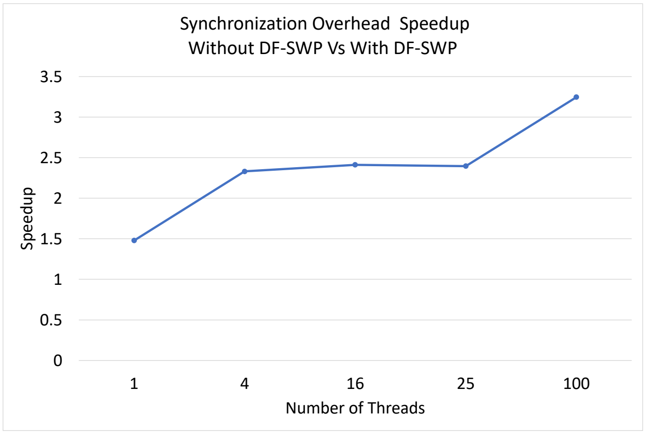


Figure 9: Synchronization Overhead analysis of Barrier and FIFO Buffer methods for matrix of size 16000

6.5 Discussion

Overall, enabling Dataflow Software Pipelining extension for Codelet Model achieve better results in terms of total throughput, latency, compute efficiency, and scalability. The speedup of 1.4*x* may look insignificant at first glance. However, it is achieved with the `FIFO` buffers implemented via software under DARTS runtime on x86 architecture. This significantly impacts the true potential of this approach. Despite these limitations, our `With DF-SWP` implementation using `FIFO` buffers achieve better scalability which results in the overall reduction in the synchronization overhead between codelets.

The proposed work assumes the responsibility of identifying suitable codelets for pipelining on the shoulders of the system that creates Codelets (e.g. programmer or compiler). Once such a pair is identified, a system can rely on the main contribution of this paper to see performance benefits. However, the task of identifying the optimal pair of codelets is out of the scope of this work. As a future direction, we are exploring how the granularity of the Codelet and the unit of synchronization in the FIFO queue can be balanced to achieve higher performance. This topic is still an open question.

The proposed approach only works if the loops can be divided up into a pair of codelets with a clear producer/consumer relationship that allow us to for the directed acyclic Codelet graphs as explained in Section 3.2. If there are multiple loops in the Codelets that is fine. More complex iteration spaces would require an extension of the supported affine functions described in section 3.2. As demonstrated by state-of-the-art polyhedral optimizations, the proposed approach is beneficial for inference-related machine learning (ML) workloads where data is streamed from layer to layer (Layers represented as Codelets/ Threaded Procedure and pipelined using proposed FIFO buffers).

It should be noted that the intention behind this work is to establish the feasibility and the advantages of Dataflow Software Pipelining techniques using readily available runtime DARTS for the codelet model. This is the beginning of seminal work of its type and its implementation in open source DARTS platform provides opportunities for others to further explore these techniques. The results that we observe here through the extensions to PXM are only one side of the coin. To exploit the full potential of this work, we need to explore extensions to CAM. This will require efficient implementation of *FIFO buffers* in the xCAM to take advantage of specific hardware features. We shed more light on this ongoing work in section 8.

7 RELATED WORK

Here we provide a brief overview of other works that we find related and inspiration behind the work discussed in this paper. The works referenced here along with the references mentioned in those works will give you a good idea of the overall landscape of related works.

Synchronous Dataflow (SDF) [32] is one of the most popular dataflow models of computation where each arc is a FIFO queue (buffer) which is used to pass data from one node to another in the dataflow graph. The efficient buffer management technique called *shift buffering* [19] has been proposed for automatic code synthesis for synchronous dataflow graphs [32]. *Reconfigurable Dataflow* [20] extends SDF with transformation rules that specify how the topology and actors of the graph may be reconfigured. There is also work on minimizing buffer sizes of dynamic dataflow implementations [40] without introducing deadlocks or reducing the performance. Implementation, validation, and comparison of several buffer size optimization techniques for the generic class of dynamic dataflow model of computation called the *Dataflow Process Network* is studied. This work presents a heuristic capable of finding a close-to-minimum buffer size configuration for deadlock-free executions. A unified algorithmic framework [37] is proposed for concurrent scheduling and register allocation to support time-optimal software pipelining. Register allocation is treated as a constraint to the software pipelining scheduling process — to derive, among all time-optimal schedules, the ones which have the potential to use the minimum number of registers. There is also work to exploit high parallelism for loop processing where *Pipelining Loop Optimizatio*n method (PLO) [44] is proposed. This makes iterations in loops flow in the processing element (PE) array of the dataflow accelerator.

The Open Computing Language (OpenCL) [25, 34] provides an attractive programming interface to express parallel execution while abstracting away many of the implementation details. *OpenCL Pipes* are part of the OpenCL Specification since version 2.0 [2]. They can be used as a channel of communication between two running kernels. The API is similar to FIFO queue ("`read_pipe`", "`write_pipe`") and may, in theory, be used to stream data from one kernel to another. SYCL [4] is a single source, C++-based offload accelerator programming framework from the Khronos group. *Dataflow Pipes* [33] describes the FIFO pipe extension to SYCL, proposed by Intel in their DPC++ compiler, that exposes a pipe abstraction close to the OpenCL design. Only FPGA implementations exist as of yet.

The multi-core evolution has presented a new dimension of challenges— namely, how to orchestrate the best software pipelining schedule in the face of resource-constrained architecture. A unified Integer Linear Programming (ILP) formulation [46] has been proposed that combines the requirement of both rate-optimal software pipelining and the minimization of inter-core communication overhead. *COStream* [47] is a programming language based on a Synchronous Data Flow execution model for data-driven applications. *Stream Dataflow* [38] is a general architecture (a hardware-software interface) that can more efficiently express programs with high computational intensity using - simple control patterns and dependencies, and simple streaming memory access and reuse patterns. This work explores the hardware and software implications along with its detailed micro-architecture and performs an evaluation. The approach of mapping streaming applications onto heterogeneous architectures using a Process Network (PN) model of computation [15] for exploiting coarse-grain pipeline parallelism exposed by a dataflow graph is also explored. The mapping onto CPU-GPU architecture is explored using 4-slot FIFO stream buffers implementations on shared memory systems where the cost of data movement outweighs the computation time.

The problem of computing a software pipeline schedule of dataflow programs with dynamic constructs [35] (like conditional paths in a dataflow program) for self-timed execution on multi-core platforms has been explored. A software pipeline scheduling technique is proposed that reduces the variation in execution time across software pipeline iterations. Hybrid Von Neumann/Dataflow approach [39] is explored which can take advantage of both out-of-order and explicit-dataflow availability in one processor. significant performance and energy improvements are observed when cores can benefit from dynamic switching during certain phases of an application's lifetime.

8 FUTURE WORK

As briefly discussed previously in the section 6.5, the next step to further exploit the potential of Dataflow Software Pipelining techniques is to explore *hardware-software co-design* techniques. Specifically, the efficient implementation of `Single Owner FIFO Buffer` is crucial. To achieve this we are looking at hardware architectures that support features like programmer addressable fast `scratchpad memory` which can be used to implement `FIFO Buffers` while taking advantage of locality.

We also wish to extend cannon's algorithm case study with more benchmarks from scientific as well as machine learning application domains. The extensions to codelet model to take advantage of dataflow software pipelining enable efficient streaming of data through the codelet graph and hence we would like to study streaming applications.

9 CONCLUSION

In this paper, we extend *software pipeline* techniques to the coarse-grain to exploit pipelined parallelism across loops by leveraging *dataflow software pipelining* principles, eliminating the limits of fine-grain parallelism. We extend the dataflow-based codelet model to efficiently support dataflow software pipelining. We perform a detailed case study of Cannon's algorithm demonstrating a relative speedup of 1.4 times when dataflow software pipelining extensions are enabled. We also show that overall synchronization overhead is reduced by 3.2 times.

This improved performance is only part of the story which is achieved from extensions to Codelet *PXM* and leaves room for further performance exploitation by extending *CAM* to support these extensions. Dataflow Software Pipelining achieved through the design principals of *Software-Hardware Co-Design* should truly unlock the scalability and performance demands of the next generation of exa-scale systems.

ACKNOWLEDGEMENT

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.

REFERENCES

[1] 2013. The Codelet Execution Model. Computer Architecture And Parallel Systems Laboratory, Newark, DE. https://www.capsl.udel.edu//codelets.shtml

[2] 2015. OpenCL Specification version 2.0 (API). (July 2015). https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf

[3] 2019. Cerebras CS-1 System. https://cerebras.net/blog/introducing-the-cerebras-cs-1-the-industrys-fastest-artificial-intelligence-computer/.

[4] 2019. "The SYCL 1.2.1 Specification. (November 2019). https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf

[5] 2021. Cerebras CS-2 System. https://cerebras.net/wp-content/uploads/2021/04/Cerebras-CS-2-Whitepaper.pdf.

[6] 2021. Graphcore Intelligence Processing Unit. https://www.graphcore.ai/products/mk2/ipu-m2000-ipu-pod4.

[7] 2021. SambaNova Reconfigurable Dataflow Unit. https://sambanova.ai/wp-content/uploads/2021/04/SambaNova_Accelerated-Computing-with-a-Reconfigurable-Dataflow-Architecture_Whitepaper_English.pdf.

[8] 2022. Habana Gaudi 2 Whitepaper. https://habana.ai/wp-content/uploads/pdf/2022/gaudi2-whitepaper.pdf.

[9] 2023. Cannons Algorithm in DARTS using Dataflow Software Pipelining. https://github.com/sraskar/cannon-dfswp.

[10] 2023. Cannons Algorithm in DARTS using Dataflow Software Pipelining. https://doi.org/10.5281/zenodo.7641196.

[11] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E.R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kurtz. 2020. Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 145–158. https://doi.org/10.1109/ISCA45697.2020.00023

[12] A. Aiken and A. Nicolau. 1988. Optimal Loop Parallelization. *SIGPLAN Not.* 23, 7 (June 1988), 308–317. https://doi.org/10.1145/960116.54021

[13] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. 1995. Software Pipelining. *ACM Comput. Surv.* 27, 3, 367–432. https://doi.org/10.1145/212094.212131

[14] Arvind and R. S. Nikhil. 1990. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Comput.* 39, 3 (March 1990), 300–318. https://doi.org/10.1109/12.48862

[15] A. Balevic and B. Kienhuis. 2011. An Efficient Stream Buffer Mechanism for Dataflow Execution on Heterogeneous Platforms with GPUs. In *2011 First Workshop on Data-Flow Execution Models for Extreme Scale Computing*. 53–57. https://doi.org/10.1109/DFM.2011.10

[16] Lynn Elliot Cannon. 1969. *A Cellular Computer to Implement the Kalman Filter Algorithm.* Ph. D. Dissertation. USA. AAI7010025.

[17] Jose M Monsalve Diaz. 2021. *Sequential Codelet Model A SuperCodelet Program Execution Model and Architecture.* PhD Thesis. University of Delaware, Newark, DE.

[18] Jose M Monsalve Diaz, Kevin Harms, Rafael A. Herrera Guaitero, Diego A. Roa Perdomo, Kalyan Kumaran, and Guang R. Gao. 2022. The SuperCodelet Architecture. In *Proceedings of the 1st International Workshop on Extreme Heterogeneity Solutions* (Seoul, Republic of Korea) *(ExHET '22)*. Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3529336.3530823

[19] N. Dutt, S. Ha, and H. Oh. 2005. Shift buffering technique for automatic code synthesis from synchronous dataflow graphs. In *2005 Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'05)*. 51–56. https://doi.org/10.1145/1084834.1084852

[20] P. Fradet, A. Girault, R. Krishnaswamy, X. Nicollin, and A. Shafiei. 2019. RDF: Reconfigurable Dataflow. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1709–1714. https://doi.org/10.23919/DATE.2019.8714987

[21] G. Gao, J. Suetterlein, and S. Zuckerman. 2011. CAPSL Technical Memo 104 :Toward an Execution Model for Extreme-Scale Systems - Runnemede and Beyond.

[22] Guang R. Gao. 1989. Algorithmic Aspects of Balancing Techniques for Pipelined Data Flow Code Generation. *J. Parallel Distrib. Comput.* 6, 1, 39–61. https://doi.org/10.1016/0743-7315(89)90041-5

[23] Guang R. Gao. 1990. *A Code Mapping Scheme for Dataflow Software Pipelining.* Kluwer Academic Publishers, Norwell, MA, USA.

[24] G. R. Gao and R. Tio. 1989. Instruction set architecture of an efficient pipelined dataflow architecture. In *[1989] Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume 1: Architecture Track*, Vol. 1. 385–392 vol.1. https://doi.org/10.1109/HICSS.1989.47180

[25] Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. 2012. Heterogeneous Computing with OpenCL. (2012).

[26] Al Geist and Robert Lucas. 2009. Major Computer Science Challenges At Exascale. *The International Journal of High Performance Computing Applications* 23, 4 (2009), 427–436. https://doi.org/10.1177/1094342009347445 arXiv:https://doi.org/10.1177/1094342009347445

[27] R. Govindarajan, Guang R. Gao, and Palash Desai. 2002. Minimizing Buffer Requirements under Rate-Optimal Schedule in Regular Dataflow Networks. *Journal of VLSI signal processing systems for signal, image and video technology* 31, 3, 207–229. https://doi.org/10.1023/A:1015452903532

[28] Richard A. Huff. 1993. Lifetime-Sensitive modulo Scheduling. *SIGPLAN Not.* 28, 6 (June 1993), 258–267. https://doi.org/10.1145/173262.155115

[29] Suetterlein Joshua. 2014. *Darts: A runtime based on the codelet execution model.* Master's thesis. University of Delaware, Newark, DE.

[30] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, and Jonathan

Ross. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. https://arxiv.org/pdf/1704.04760.pdf

[31] M. Lam. 1988. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. *SIGPLAN Not.* 23, 7, 318–328. https://doi.org/10.1145/960116.54022

[32] E. A. Lee and D. G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (Sep. 1987), 1235–1245. https://doi.org/10.1109/PROC.1987.13876

[33] John Freeman Michael Kinsner. 2019. Data Flow Pipes: A SYCL Extension for Spatial Architectures. (November 2019). https://h2rc.cse.sc.edu/papers/lightning_2_3_Kinsner.pdf

[34] A. Munshi. 2009. The OpenCL specification. , 314 pages. https://doi.org/10.1109/HOTCHIPS.2009.7478342

[35] Y. Murarka, P. Gode, S. K. Pasupuleti, and S. Kohli. 2014. Software pipelining of dataflow programs with dynamic constructs on multi-core processor. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. 340–347. https://doi.org/10.1109/ICCD.2014.6974703

[36] Qi Ning. 1993. *Register allocation for optimal loop scheduling.* PhD Thesis. McGill University, Canada.

[37] Qi Ning and Guang R. Gao. 1993. A Novel Framework of Register Allocation for Software Pipelining. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. ACM, New York, NY, USA, 29–42. https://doi.org/10.1145/158511.158519

[38] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam. 2017. Stream-dataflow acceleration. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 416–429. https://doi.org/10.1145/3079856.3080255

[39] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. 2015. Exploring the Potential of Heterogeneous von Neumann/Dataflow Execution Models. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 298–310. https://doi.org/10.1145/2749469.2750380

[40] A. A. A. Rahman, S. Casale-Brunet, C. Alberti, and M. Mattavelli. 2014. A methodology for optimizing buffer sizes of dynamic dataflow fpgas implementations. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 5003–5007. https://doi.org/10.1109/ICASSP.2014.6854554

[41] B. Ramakrishna Rau and Joseph A. Fisher. 1993. Instruction-level parallel processing: History, overview, and perspective. *The Journal of Supercomputing* 7, 1 (01 May 1993), 9–50. https://doi.org/10.1007/BF01205181

[42] B. R. Rau and C. D. Glaeser. 1981. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. In *Proceedings of the 14th Annual Workshop on Microprogramming* (Chatham, Massachusetts, USA) *(MICRO 14)*. IEEE Press, Piscataway, NJ, USA, 183–198. http://dl.acm.org/citation.cfm?id=800075.802449

[43] Joshua Suettlerlein, Stéphane Zuckerman, and Guang R. Gao. 2013. An Implementation of the Codelet Model. In *Proceedings of the 19th International Conference on Parallel Processing* (Aachen, Germany) *(Euro-Par'13)*. Springer-Verlag, Berlin, Heidelberg, 633–644. https://doi.org/10.1007/978-3-642-40047-6_63

[44] Xu Tan, Xiao-Chun Ye, Xiao-Wei Shen, Yuan-Chao Xu, Da Wang, Lunkai Zhang, Wen-Ming Li, Dong-Rui Fan, and Zhi-Min Tang. 2018. A Pipelining Loop Optimization Method for Dataflow Architecture. *Journal of Computer Science and Technology* 33, 1 (2018), 116–130. https://doi.org/10.1007/s11390-017-1748-5

[45] Nancy J. Warter, Grant E. Haab, Krishna Subramanian, and John W. Bockhaus. 1992. Enhanced modulo Scheduling for Loops with Conditional Branches. *SIGMICRO Newsl.* 23, 1–2 (Dec. 1992), 170–179. https://doi.org/10.1145/144965.145796

[46] H. Wei, J. Yu, H. Yu, M. Qin, and G. R. Gao. 2012. Software Pipelining for Stream Programs on Resource Constrained Multicore Architectures. *IEEE Transactions on Parallel and Distributed Systems* 23, 12 (Dec 2012), 2338–2350. https://doi.org/10.1109/TPDS.2012.41

[47] Haitao Wei, Stéphane Zuckerman, Xiaoming Li, and Guang R. Gao. 2014. A Dataflow Programming Language and its Compiler for Streaming Systems. *Procedia Computer Science* 29 (2014), 1289 – 1298. https://doi.org/10.1016/j.procs.2014.05.116 2014 International Conference on Computational Science.

[48] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. 2011. Using a "Codelet" Program Execution Model for Exascale Machines: Position Paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era* (San Jose, California, USA) *(EXADAPT '11)*. ACM, New York, NY, USA, 64–69. https://doi.org/10.1145/2000417.2000424