Study of Dataflow Software Pipelining under Codelet Model using Cannons Algorithm

Siddhisanket Raskar\textsuperscript{12}, JoseMMonsalveDiaz\textsuperscript{12}, ThomasApplencourt\textsuperscript{1}, Kalyan Kumaran\textsuperscript{1}, GuangGao\textsuperscript{2}

\textsuperscript{1}Argonne National Lab, Lemont, IL.
\textsuperscript{2}University of Delaware, Newark, DE.

\{tapplencourt,kumaran\}@anl.gov
\{sraskar,josem,ggao\}@udel.edu

CAPSL Technical Memo 135 Version 1
February, 2020

Copyright © 2020 CAPSL at the University of Delaware
Contents

1 Introduction and Background 5
   1.1 Software Pipelining .................................................. 5
   1.2 Dataflow Software Pipelining ...................................... 5
   1.3 Codelet Model .......................................................... 6
   1.4 Open Questions ....................................................... 6

2 Motivation and Problem Formulation 7
   2.1 Motivating Example ................................................... 8
   2.2 Problem Formulation .................................................. 9

3 Major Contributions 10

4 Extensions to enable Dataflow Software Pipelining for Codelet Model 10
   4.1 Extensions to Codelet Model ......................................... 11
      4.1.1 Extension to Codelet Program Execution Model (PXM) .......... 11
      4.1.2 Extension to Codelet Abstract Machine Model (CAM) .......... 11
   4.2 FIFO Buffers ............................................................ 12

5 Implementation of Dataflow Software Pipelining for Codelet Model 13
   5.1 Cannon’s Algorithm ................................................... 13
      5.1.1 Why Cannon’s Algorithm? ....................................... 14
   5.2 DARTS ........................................................................ 15
   5.3 Cannon’s Algorithm under Codelet Model ............................ 15
   5.4 Cannon’s Algorithm with Dataflow Software Pipelining under Codelet Model 17

6 Experimental Results 18
   6.1 A Summary of Results .................................................. 19
   6.2 Experimental Setup ..................................................... 19
      6.2.1 Experimental Testbed ................................................ 19
      6.2.2 Experimental Design ............................................... 19
      6.2.3 Terminology .......................................................... 20
   6.3 Performance Evaluation ............................................... 20
      6.3.1 Relative Speedup .................................................... 20
      6.3.2 Compute Efficiency ................................................. 21
   6.4 Scalability Analysis .................................................... 21
      6.4.1 Weak Scaling ........................................................ 21
      6.4.2 Strong Scaling ....................................................... 22
      6.4.3 Synchronization Overhead ....................................... 22
   6.5 Discussion .................................................................. 22

7 Related Work 23
   7.1 Buffer Related ............................................................ 24
   7.2 Programming Models ................................................... 25
   7.3 Streaming Dataflow ...................................................... 26
   7.4 Others ........................................................................ 26

8 Future Work 26

9 Conclusion 27
Abstract

Chip architectures are shifting from few, faster, functionally heavy cores to abundant, slower, simpler core designs, however, it has not been possible to truly exploit software pipelining at higher levels. Dataflow Software Pipelining is a code mapping technique to generate code which can be executed in a pipelined fashion with high throughput. Traditional architectures and their execution models are based on conventional sequential Von Neumann model. Parallelism has been achieved by aggregating multiple traditional cores together, but under current trends of chip and system architecture, it has been data parallelism that has dominated while there has been little progress in parallelism through Software Pipelining techniques. Dataflow is a sound, simple, and powerful model of parallel computation with no notion of a single point or locus of control (Program Counter) which breaks through the bottleneck of sequential program execution models. Dataflow models allows taking advantage of software pipelining but it needs to be properly embedded in the execution model. The Codelet model is a hierarchical, multi-threading event-driven, fine-grained model which draws its roots from the dataflow model. Existing implementations based on Codelet model have already shown that they are on par with control-centric execution models, such as OpenMP for regular workloads and outperform classical coarse-grain approaches for irregular workloads.

Thanks to the Dataflow influence of the Codelet model, it is possible to build upon it to leverage Codelets to exploit asynchronous task parallelism through software pipelining techniques. We extend the base Codelet program execution model to take advantage of dataflow software pipelining principals at the Codelet graph level by implementing efficient single owner FIFO buffers across Codelet’s dependencies.

In this work, we extend our existing implementation of the Codelet Model namely DARTS to allow Software Pipelined Codelet execution. We also use the Cannons algorithm for matrix multiplication to study performance implications of using the Extended Codelet model. We show improvement of 1.4x with use of dataflow software pipelining extensions for codelet model in comparison to a the basic Cannon’s algorithm implementation under codelet model.
1 Introduction and Background

In this section, we will introduce you to the basic terminology with the purpose of providing essential background for the rest of this paper.

1.1 Software Pipelining

Software pipelining\cite{1, 2, 3, 4, 5} 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. The pioneering work for formulation of the software pipelining process for single basic block loops was stated by B. Ramakrishna Rau\cite{6}. He also presented a condition that has to be met by any legal software pipelined schedule, the modulo constraint and derived lower bounds on the rate at which successive iterations of the loop can be started, that is, the initiation interval (II). A simple, deterministic software pipelining algorithm based on the modulo scheduling algorithm, was shown to achieve the minimum II, thereby yielding an asymptotically optimal schedule. Please refer to \cite{7, 8} for introductory survey of classical software pipelining mechanics.

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. This work is standard part of modern compilers. We wish to leverage all these advancements at Coldelet level. This is described in further details in section 2.2.

1.2 Dataflow Software Pipelining

Dataflow Software Pipelining\cite{9} is a code mapping technique to generate code which can be executed in a pipelined fashion with high throughput. This is achieved by keeping the pipeline busy and computation balanced\cite{10}. Special dataflow ID nodes\cite{11} or FIFO data buffers\cite{12} are used to optimally balance the dataflow graph for maximum pipelining\cite{10}.

Figure 1 shows a four stages dataflow software pipeline with seven dataflow actors. The applicative nature of the dataflow model allows flexible scheduling of enabled actors within the pipeline. The ideal dataflow scheduler\cite{?} will execute each actor as soon as its input data is available on its input arcs and there is space to store output data on the output arc. As a result, execution of seven operations overlap each other. Performing operations on different data elements concurrently is called dataflow software pipelining.

We take our inspiration to use FIFO data buffers to enable dataflow software pipelining for codelet graphs from this work. This is described in further details in section 4.2.
1.3 Codelet Model

The Codelet Model[13, 14] is a hybrid von Neumann-dataflow execution model designed for extreme-scale systems in mind. 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.

The Codelet Abstract Machine (CAM) shown in figure 2 consists of many nodes connected together via an interconnection network. Each node is expected to have several chips containing hundreds of cores. Interconnects with varying latency will connect components at the multiple levels. We envision two types of cores. The first is a Compute Unit (CU) which is responsible for performing operations. The second is a Synchronization Unit (SU) which is responsible for steering computation into the compute units.

A Codelet is a unit of computation that can be atomically scheduled for execution and it is the principal scheduling quantum in Codelet based execution models. It consists of a sequence of machine instructions that execute non-preemptively upon availability of specific resources, the primary one being data. Other resources may include bandwidth requirements, maximal power envelope, the use of an accelerator, network access, etc. Codelets are linked together to form a Codelet Graph (CDG). In a CDG, each codelet acts as a producer and/or consumer.

The original codelet model or its abstract machine do not support dataflow software pipelining. Traditional Software pipelining (as explained above) is implicitly supported by instructions inside of each codelet, our intention is to extend the codelet model to support dataflow software pipelining in-between codelets.

1.4 Open Questions

In this work, we are interested in exploring following questions -

- How Codelet Model can be extended to enable base Dataflow Software Pipelining?
• What are the advantages of extending codelet model in terms of performance characteristics such as throughput, latency, computations efficiency?

• How does the performance scale in the current and future exa-scale systems in terms of strong scaling, weak scaling and synchronization cost?

2 Motivation and Problem Formulation

*Dataflow Software Pipelining* for codelet graphs is a broad field of study, and it covers various techniques applicable to general class of codelet graphs. In the beginning, in section 2.1 we will discuss simple motivating example to highlight the importance and need of dataflow software pipelining for codelet model. Following that, in section 2.2, we will define the class of codelet graphs for which the dataflow software pipelining techniques discussed in this paper will be applicable.
2.1 Motivating Example

In this section, we will study a simple example to motivate our work. Consider a scenario shown in Fig.3a. Here we are depicting a part of CDG where two codelets are in a producer-consumer fashion. 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.
- SUs schedules C1 & C2 on separate computation units.
- Each iteration of loops L1 and L2 take 1 unit time.
- Sending signal/data from C1 to C2 take 1 unit time.

Now, we will describe 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 the signal and start its execution.
- C2 will completely finish executing its code by performing the 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 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. $N \rightarrow \infty$).
2.2 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 considers the entire dataflow graph for optimization opportunities. The high level objective that we are trying 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 Codelet 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 upon 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. An *affine function* is a function composed of a linear function and a constant. The equation for an affine function is:

   \[ Y = u \times i + v \]

   Furthermore, we will be restricting the variable values in affine function. Here are further details -
• Restrictions on $u$: We will 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 will restrict $i$ to only 1. This is to ensure simple access patterns inside array. This means, we will not allow access of type $i^2$ or $3i$.

The idea behind the aforementioned formulation and restrictions described in this section is to 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 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 could be optimized by traditional software pipelining techniques.

3 Major Contributions

The major contributions of this work can be summarized as -

• We formulate the detailed problem of dataflow software pipelining for Codelet Model, propose solution methodology to extend the original codelet model and implement these extensions in dataflow based runtime - DARTS.

• We perform detailed analysis of performance characteristics like throughput and computing efficiency to show the advantages of dataflow software pipelining extensions.

• We perform detailed scalability study using Cannon’s algorithm for Matrix multiplication to show importance of dataflow software pipelining extensions in current and future exascale systems.

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 codelets 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 2.2). 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 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 implementations can be achieved using 2 bit flag & 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, specially in NUMA systems. Hence we have extended our CAM to support this mechanism differently.

4.1.2 Extension to Codelet Abstract Machine Model (CAM)

Extending the Program Execution Model 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.

To accomplish this, there has been on-going efforts [15] to extend the Codelet Model. Figure 4 shows part of the Extended Codelet Abstract Machine (xCAM) which is most relevant for this paper. The main differences between the original and our extended 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 cores, GPU or other kind of heterogeneous hardware in the architecture.

The incorporation of LCCM in the extended codelet model plays a vital role to enable 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 improving 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 method to store intermediate results and ensure order of data tokens in-between two computation units, specially when we have single producer and single consumer, is to use temporary storage units like FIFO (First-In-First-Out) buffers. They can provide highly efficient data communication channel without relying on operating system constructs such as semaphores, mutexes, or monitors for data transfer.

In figure 3b, we show conceptual visualization of a FIFO channel between producer and consumer codelets using same example from section 2.2. 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 1st 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 will discuss details of how we implemented dataflow software pipelining extensions to Codelet Model discussed in the above section 4. Before, we dive into actual implementation details, we will first provide brief background on Cannon’s Algorithm (5.1) & DARTS - Codelet based runtime (5.2) we use to implement these extensions. Following this, section 5.3 discusses mapping and implementation of Cannon’s algorithm for Codelet Model. In section 5.3, we will discuss how we map Cannon’s algorithm to Codelet Model. In section 5.4, we have comparative discussion on how we extend the implementation in 5.3 to leverage Dataflow Software Pipelining techniques using FIFO Buffers.

5.1 Cannon’s Algorithm

Cannon’s Algorithm is a distributed algorithm for matrix multiplication for two-dimensional meshes first described in 1969 by Lynn Elliot Cannon [16] suitable for computers laid out in an N × N mesh. The main advantage of the algorithm is that its storage requirements remain constant and are independent of the number of processors.

The Cannon’s algorithm is described in Algorithm 1. The 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 steps of Cannon’s algorithm [16] from 1 can be summarized as follow:

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 sub-matrices and partial result C is calculated.

3. **Communication Step**: Each process circular shifts its sub-matrix 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 above steps 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 details in section 5.4 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.

13
Algorithm 1: Cannon’s Algorithm for Matrix Multiplication

Data: Two Square Input Matrices
Result: Square Resultant Matrix

Initialization- Skew Matrices
for $i ← 0$ 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 $j ← 0$ 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 $k ← 0$ to $N$ do
  for $i ← 0$ to $N − 1$ do
    for $j ← 0$ 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 $A(i, (j+1)\mod N)$;
      Upward circular shift each column of $B$ by 1,
      so $B(i,j)$ is assigned $B((i+1)\mod N, j)$;
    end
  end
end

5.1.1 Why Cannon’s Algorithm?

At this point, astute reader might be wondering about why we chose Cannon’s algorithm in particular out of plenty available options from scientific and machine learning domains to demonstrate effectiveness of dataflow software pipelining techniques. The answer lies in the peculiar structure of Cannon’s algorithm and the opportunities it provides to map it to the Codelet Model.

Earlier in section 2.2 we formulated certain constraints at Codelet Graph level as well as at Codelet level. The way we map Cannon’s algorithm to Codelet model satisfies our constrains at Codelet Graph level as the CDG for Cannons algorithm is Directed Acyclic Graph. Inside each codelet, we have matrix multiplication operation followed by shifting operation which are affine functions and can take advantage of traditional software pipelining.
5.2 DARTS

*Delaware Adaptive Runtime System* (DARTS) \[17\] \[18\] is currently the most faithful runtime implementation of the Codelet Model and its Abstract Machine Model (AMM). It is written in C++ and distributed as free and open-source software \[19\].

In DARTS, *Threaded Procedures (TP)* and *Codelets (CD)* are defined as class objects at compile time and are instantiated at runtime. A TP is instantiated when a codelet performs a TP invocation. This operation creates a TP handle and sends a request to the runtime to allocate memory for the TP instance and the instances of the codelets it contains. Once this is done, codelet instances are executed upon satisfaction of their dependencies.

A two-level scheduling mechanism, comprising *TP schedulers* and *Micro-schedulers*, is used for the scheduling and execution of codelets. A TP scheduler is a core which has been assigned the role of *Synchronization Unit (SU)* and has a TP queue (TPQ) with TP instantiation requests and a ready-codelets queue (RCQ). A micro-scheduler is a core acting as a *Compute Unit (CU)* with a *Local Codelet Queue (LCQ)*, which is fed by its local TP scheduler with codelets from its *Remote Codelet Queue (RCQ)*.

DARTS allows users to determine the “shape” of the AMM by setting the number of clusters in the machine and the number of CUs associated with an SU. This can be done based on the topology of the target architecture, such as the number of sockets or NUMA nodes. Hence, the two-level parallelism exposed by the Codelet Model is preserved and configurable in DARTS.

Users can also fine-tune DARTS AMM by setting the scheduler affinity policy as *COMPACT* (SUs and CUs are pinned down to physically contiguous cores), *SPREAD* (schedulers are assigned in a round-robin fashion to the different sockets of the platform), or user-defined using an environment variable that follows the same notation as GCC’s *GOMP_CPU_AFFINITY*.

In addition, different scheduling policies are available: namely, *standard* and *work-stealing*. With a standard policy, a TP scheduler only initializes TPs assigned locally by one of its micro-schedulers and distributes codelets in round-robin fashion to them. With work-stealing, on the other hand, a TP scheduler may also steal TP handles from another TP scheduler’s TPQ. A microscheduler with standard policy only executes codelets pushed into its local queue by its TP scheduler. However, if its policy is work-stealing, it may steal codelets from the TP scheduler’s RCQ.

5.3 Cannon’s Algorithm under Codelet Model

The figure 6 shows the Codelet Graph (CDG) for the Cannon’s algorithm. We divide all matrices among \( P \) tiles and associate them to \( P \) codelets similar to algorithm 1. Here \( P \) is number of tiles in which matrices are divided. The functionality of various codelets can be summarized as follows-

- **CopyA & CopyB**: These codelets copy original matrix \( A \) & \( B \) 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/initilize matrix A & B as described in the skew step of algorithm. 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 & B, stores results in sub-matrix C. It also circularly shifts sub-matrix A & B. It sends 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 to back to main memory from tile memory.

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

- Each tile consists of blocks for sub-matrix A, B & C. Each tile also contains blocks Aw & Bw for which are used to write sub-matrix from neighbouring tile 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 Coldelet Model. For further clarification, we map these stages from Codelet Graph to our sample 3x3 tiles example. The numbers in figure correspond to the numbers in figure with each number for each stage.

- **Stage 1**: CopyA & CopyB codelets copy sub-matrix A & B from main memory to tile memory of codelets. This is show with the bold arrows on (top and left periphery)

- **Stage 2**: Sub-matrix A & B are skewed. Skew codelet performs this operation. Sub-matrix blocks for A & B along with Aw & Bw are used with skew phase communication shown using blue arrows.

- **Stage 3**: Sub-Matrix C is calculated using Compute codelet. Sub-matrix A & 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 show using bold arrows (right side periphery).
5.4 Cannon’s Algorithm with Dataflow Software Pipelining under Codelet Model

The figure 8 shows the Codelet Graph(CDG) for the Cannon’s algorithm with dataflow software pipelining while figure 7 shows the sample 3x3 tiles example. These two figures are very synonymous to their counterparts in the earlier section 5.3 and follow the 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 (6) and that of with dataflow software pipelining (8) is the removal of loop codelet which acted as barrier between various iterations of compute codelet.

However, such 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 don’t need to wait for the barrier as its can simply Pop its input from the FIFO buffer and continue with its execution.

- To accomplish this, sub-matrix blocks for A & B are replaces with FIFO buffers for
respective matrices in figure 7.

- 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 6. Since, we don’t need **loop** codelet anymore, we need mechanism to make sure **skew** phase finishes completely and matrices are in correct order before **compute** codelets can start execution in order to avoid data races.

![Figure 7: Cannon’s Algorithm implementation use of FIFO buffers and Communication between matrix tiles divided in 3x3](image)

![Figure 8: Codelet Graph(CDG) for Cannon’s algorithm where FIFO buffers are used between Compute Coldelets to enable Dataflow Software Pipelining](image)

6 Experimental Results

In this section, we present the results of 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 experimental setup parameters. The sections 6.3 & 6.4 go in the details of each observation made in the 6.1. Finally, we discuss the significance of results and draw conclusions in the section 6.5.
6.1 A Summary of Results

Our experimental results show that -

- **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 core count. Performance is negatively affected when codelets are mapped to different socket and then 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, the much better compute efficiency is observed.

- **Observation 3 (Refer Section 6.4):** Synchronization overhead decreased 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 a Intel based shared memory node. All cores feature 32KB private L1 instruction and data caches, and 1MB private unified L2 cahces. Each node has two sockets, with 28 cores per socket, featuring Intel Xeon Platinum 8180M(Skylake) processor clocked at 2.5GHz with Hyper-Threading (HT). A 38.5MB unified L3 cache is shared by all the cores in the same socket. The total memory is 383GB 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. Intel MKL 2020 [20].

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 effects of enabling dataflow software pipelining extensions, we places threads on separate cores until all cores were assigned at least one thread. We chose this in order to minimize 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 over). 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.

### 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 -

- **Barrier**: As a baseline, we map Cannon’s algorithm to codelet model and implement it in dataflow based runtime DARTS [17]. The implementation uses a loop codelet as barrier between iterations compute codelets as explained in the section 5.3. We refer to this implementation as Barrier in all result graphs.

- **FIFO Buffer**: We extend the above base implementation with dataflow software pipelining by using FIFO buffers as explained in the section 5.4. We refer to this implementation FIFO Buffer in the graphs in this section.

- **Intel MKL**: To put above results into perspective of what near theoretical performance looks like on x86 architecture, we perform Matrix Multiplication using Intel MKL. We refer to this implementation MKL in the graphs in this section.

### 6.3 Performance Evaluation

In this section, we talk about the experiments that we perform to quantify the performance gains of dataflow software pipelining extensions.

#### 6.3.1 Relative Speedup

The figure 9 shows the relative speedup achieved with use of dataflow software pipelining extensions compared against the implementation that doesn’t use dataflow software pipelining extensions. We ran Matrix-Matrix multiplication experiment with tile size of 64 per codelet/thread while using increasing the number of threads/codelets. We achieved maximum speedup of 1.4x for thread count of 64.

In general, the speedup increases with the increase in core count. We see drop in speedup at thread count 26 as execution starts using 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.
6.3.2 Compute Efficiency

The figure 10 shows the compute efficiency of Barrier & FIFO Buffer implementations plotted for increasing size of matrices with fixed 100 count of threads/codelets. In general, the compute efficiency of both methods improve as the size of matrix increase.

We can see that FIFO buffer achieves better compute efficiency compared against Barrier method. As the size of the matrices increase, the results converge as the elapsed time is dominated by computation and the relative synchronization cost is reduced. We also show results of Intel Math Kernel Library (MKL) with dotted gray line to put results in the perspective.

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.

6.4.1 Weak Scaling

The figure 11 shows the weak scaling analysis of Barrier & FIFO Buffer. 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 FIFO Buffer method.
6.4.2 Strong Scaling

The figure 12 shows the strong scaling analysis of Barrier & FIFO Buffer. 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, strong scaling curve shows better results for the FIFO buffer method.

6.4.3 Synchronization Overhead

The figure 13 shows the synchronization overhead analysis of Barrier & FIFO Buffer methods. 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 3.2x time decrease in synchronization overhead when using FIFO Buffers. This also gives us insights about the weak and strong scaling results discussed earlier.

6.5 Discussion

Overall, we can infer from results that by enabling Dataflow Software Pipelining extension for Codelet Model achieve better results in terms of total throughput, latency, compute efficiency and scalability. However, we can also see that Intel MKL [20] implementation performs notice-
ably better. This is in part due to various x86 architecture specific optimizations that MKL takes advantage of. We can consider MKL implementation as the near theoretical peak.

The intention behind this work is to establish the advantages of Dataflow Software Pipelining techniques. In this work, we implement these extensions in the Codelet Program Execution Model (PXIM) to achieve above discussed speedup. To exploit further speedup, we need to explore efficient implementation of FIFO buffer in the LCCM in the Extended Codelet Abstract Machine(xCAM). To achieve this, we need to implement xCAM to take advantage of hardware architecture specific features. This will be explored further in the future work.

7 Related Work

In this section, we give brief overview of other works that we find related to the work discussed in this document. The works referenced here along with the references mentioned in those works will give you a good idea of overall landscape of related works. Though these work domains overlap with each other in some aspects, we have tried to classify them for ease readability.
7.1 Buffer Related

*Synchronous Dataflow (SDF)* [21] 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* [22] has been proposed for automatic code synthesis for synchronous dataflow graphs [21]. The proposed shift buffering method shifts samples rather than moving buffer indices based on optimal shift buffering algorithms to minimize the number of shifted samples.

SDF (and most of its variants) lacks the capability to express the dynamism needed by modern streaming applications. In particular, the applications that have a strong need for reconfigurability to accommodate changes in the input data, the control objectives, or the environment. *Reconfigurable Dataflow* [23] 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 [24] 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 an heuristic capable of finding a close-to-minimum buffer size configuration for deadlock-free executions.

A unified algorithmic framework [25] 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 [26] to exploit high parallelism for loop processing where Pipelining Loop Optimization method (PLO) is proposed. This makes iterations in loops flow in the processing element (PE) array of dataflow accelerator.

### 7.2 Programming Models

The Open Computing Language (OpenCL) [27] [28] provides a 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 [29]. They can be used as a channel of communication between to running kernels. They 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 [30] is a single source, C++-based offload accelerator programming framework from the Khronos group. Dataflow Pipes [31] 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 implementation exist yet.

![Synchronization Overhead Speedup Barrier Vs FIFO Buffer](image)

Figure 13: Synchronization Overhead analysis of Barrier and FIFO Buffer methods for matrix of size 16000
7.3 Streaming Dataflow

The multi-core evolution has presented a new dimension of challenges: that is how to orchestrate the best software pipelining schedule in the face of resource constrained architecture. A unified Integer Linear Programming (ILP) formulation\cite{32} has been proposed that combines the requirement of both rate-optimal software pipelining and the minimization of inter-core communication overhead. \textit{COStream}\cite{33} is a programming language based on Synchronous Data Flow execution model for data-driven application. COStream compiler framework is for general-purpose multi-core architecture where inter-thread software pipelining scheduler is used to exploit the parallelism among the cores.

\textit{Stream Dataflow}\cite{34} is a general architecture (a hardware-software interface) which can more efficiently expresses 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 perform evaluation.

The approach of mapping of streaming applications onto heterogeneous architectures using a Process Network (PN) model of computation\cite{35} 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.

7.4 Others

The problem of computing a software pipeline schedule of dataflow programs with dynamic constructs\cite{36} (like conditional paths in a dataflow program) for self-timed execution on multi-core platforms has been explored. Software pipeline scheduling technique is proposed that reduces the variation in execution time across software pipeline iterations.

Hybrid Von Neumann/Dataflow approach\cite{37} 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 dynamically switching during certain phases of an application’s lifetime.

8 Future Work

As briefly discussed previously in the section\cite{6.5}, the next step to further exploit potential of Dataflow Software Pipelining techniques, hardware architecture specific implementation which can take advantage of specific hardware features to implement FIFO buffers is very important. To achieve this, we are looking at the hardware platforms with features like scratchpad memory. We can allocate memory manually on the scratchpad memory for the FIFO buffers and take advantage of locality and buffer memory model.
In this work, we study Cannon’s algorithm for Matrix-Matrix multiplication which shows the importance of Dataflow Software Pipelining techniques in the domain of scientific applications. We also see these extensions to be very useful for the machine learning application domain and applications where data streams through the codelet graph. In the future, we will explore these domains to show the advantages of dataflow software pipeling on top of scientific application domain.

9 Conclusion

In this paper, we propose the extensions for Codelet Model to enable Dataflow Software Pipelining. We implement these extensions for Codelet based runtime DARTS. We map Cannon’s algorithm for matrix multiplication to Codelet Model and extend our base implementation with dataflow software pipelining. We perform experimental evaluation demonstrating 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. These performance improvements only exploit extensions to the and leaves.

This improved performance is only part of the story which is achieved from extensions to Codelet Program Execution Model(PXM) and leaves room for further performance exploitation by extending Codelet Abstract Machine(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.

References


