Extending Codelet Model for Dataflow Software Pipelining using Software-Hardware Co-design

Siddhisanket Raskar\textsuperscript{12}, Thomas Applencourt\textsuperscript{1}, Kalyan Kumaran\textsuperscript{1}, Guang Gao\textsuperscript{2}

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

CAPSL Technical Memo 133.1
June, 2019

Copyright © 2019 CAPSL at the University of Delaware
Abstract

Trends in processor and system architecture, driven by power and complexity, point super-computing landscape toward very high and heterogeneous core count designs. As the number of cores inevitably increase traditional ways of performing large scale computations will also need to evolve from legacy models like OpenMP & MPI. However, such models are very much wedded to a control-flow vision of parallel programs, making it difficult to express asynchrony in programs. To address these challenges, codelet model was developed which is fine-grained, event-driven asynchronous program execution model. In-spite of its initial design goals, the Codelet model is rife with opportunities for further improvements to provide high-performance for both data and control regular applications.

The major inspiration behind this work is to leverage the decades of research done to exploit instruction level parallelism (ILP) for the machine instructions inside codelet while build upon the dataflow software pipelining principals at codelet graph level to further enhance performance. In this paper, we propose hardware assisted extensions to the original codelet program execution model in order to implement efficient dataflow software pipelining and extend capabilities of the codelet model. This hardware-software co-design focuses on efficient implementation of data FIFO buffers leveraging proposed optimizations like - FIFO ring buffers and multiple-head FIFO buffers and single owner FIFO buffers to further exploit advantages of dataflow software pipelining.

The wide range of scientific, machine learning and specially streaming applications should be able to take advantage of techniques proposed in this paper. Identifying these kernels and bench-marking them are the next anticipated steps for us.

1 Introduction

The supercomputing landscape has fundamentally changed in the past fifteen years [1]. Chips have evolved from single-core to multi-threaded, multi- or even many-core chips. Chip architectures are shifting from few, faster, functionally heavy cores to abundant, slower, simpler cores to address pressing physical limitations such as energy consumption and heat expenditure. Hardware architectures like GPU, FPGA and domain specific accelerators are becoming more common.

From the software side, languages provide a way to create threads and atomic types to enable lock-free synchronization. OpenMP, one of the most popular execution model for parallel programming in shared memory, has been augmented to take into account various types of parallelism, from traditional data-parallel constructs to fine-grain, data-driven ones. Although parallelism is now regarded as inevitable for high-performance computing, the parallelism delivered by software has been limited. A software approach to delivering parallelism indeed yields a high degree of complexity for the programmer.

The Codelet Model [2, 3, 4] is a fine-grain dataflow-inspired and event driven program execution model which was designed to run parallel programs on a combination of such many-core chips architectures. Original Codelet Model [2] addresses many of the shortcomings of the traditional program execution model yet the base Codelet model is rife with opportunities to
provide high-performance for both data and control regular applications. In this paper, we will discuss challenges and suggestions to enable dataflow software pipelining behaviour for codelet model. We discuss software pipelining, dataflow software pipelining and original codelet model in more details in section 2.

We take inspiration for our work in this paper from the pioneering work done in the field of software pipelining[5, 6] and dataflow software pipelining[7, 8]. Inside each codelet, instead of re-inventing the wheel, we want to take advantage of the research done in the field of instruction level parallelism and software pipelining. At Codelet graph level, we take inspiration from the research done in the field of dataflow software pipelining. We explain our approach in further details in the section 3.

We propose to design and develop an abstract machine model for scalable parallel and distributed computing based on fine-grain, event-driven Codelet Program Execution Model (PXM). We propose studying, designing, and implementing hardware-assisted mechanisms to realize the Extended Codelet Abstract Machine (xCAM) instead of relying on off-the-shelf hardware and pure software implementations. We propose a extensions to the original codelet model to enable efficient software pipelining based on the principles of hardware-software co-design. We propose extensions and optimization for efficient software pipelining for the codelet model which are discussed in further details in section 4.

The main contributions of this paper can be summarized as following:

• Formulating the detailed problem of dataflow software pipelining for codelet model.
• Proposing extensions to the original codeletpxm to support dataflow software pipelining.
• Proposing optimization for FIFO buffer by extending Codelet Abstract Machine (CAM) based on hardware-software co-design.

2 Background And Motivation

In this section, we give brief overview of the basic terminology with the purpose of providing essential background for the rest of this paper.

2.1 Software Pipelining

Software pipelining 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 [5]. 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.

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 codelet level. This is described in further details in section 3.

2.2 Dataflow Software Pipelining

Dataflow Software Pipelining \[8\] 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 & computation balanced. Special dataflow ID nodes \[9\] or FIFO data buffers are used to optimally balance the dataflow graph for maximum pipelining.

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 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 3.
2.3 Codelet Model

The Codelet Model [2] [3] 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.

3 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 this section, we will define the class of codelet graphs for which the dataflow software pipelining techniques discussed in this paper will be applicable.

3.1 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 Codelet Graph (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 Compute Units (CUs)
- Synchronization Units (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. $N \to \infty$).
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 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.

4 Solution Methodology

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

4.1 Extended Codelet Model

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). 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 Program Execution Model (PXM). One of the possible implementation 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 Codelet Abstract Machine (CAM) to support this mechanism differently.

There has been on-going efforts [10] 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.

• Compute Unit (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 3. 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.

4.3 Efficient Dataflow Software Pipelining through Software-Hardware Co-Design

At this point, we have established advantages of using FIFO buffers to temporarily store data tokens to enable dataflow software pipelining for the codelet model. However, if the Codelet PXM relies on an operating system to provide this functionality implicitly via shared memory sub-system, then we will be limiting ourselves from fully exploiting the potential of FIFO buffers. This functionality needs to be baked into the abstract machine model and available in the implemented hardware. We propose the following software-hardware co-design based optimization for FIFO data buffers to be incorporated into the CAM.

4.3.1 FIFO Ring Buffer

A circular buffer or a ring buffer is one of the most efficient data structure for implementation of FIFO buffer. It uses a single, fixed-size buffer and address it as if it were connected-end-to-end. Circular buffer makes a good implementation strategy for a queue that has fixed maximum size. Every time a data token is added, the associated tail pointer will be updated. Likewise, everytime a data token is removed, the associated head pointer is updated.

We propose further optimization to the base ring buffers using hardware-software co-design as extensions to the codelet PXM. Figure 6a depicts a FIFO Ring Buffer for single producer, single consumer with Codelet P as producer and Codelet C as consumer. The higher level working functioning can be summarized as -

- The FIFO Ring Buffer data structure is allocated in the LCCM inside the codelet core to ensure data locality between producer and consumer codelets.
- The address of head & tail pointers of the FIFO Ring buffer are maintained in the register files of each CU.
- Every time data tokens are added/removed from FIFO Ring Buffer for dataflow software pipelining only associated pointers in registers are updated in the respective CUS.
4.3.2 Multiple head FIFO Buffer

For a simple Single producer - Single consumer scenarios, FIFO buffers are a good solution. However, not all codelet graphs are this simplistic. Let us consider scenarios shown in figure 5 where we show cases of codelet graphs with multiple consumers are presented.

- Figure 5a shows Chain of Codelets with three codelets. Here C1 is the only producer codelet. All other codelets are consumer as well as producer codelets.

- Figure 5b shows Tree of Codelets. Here codelet P is the only producer codelet. C1, C2 &
C3 are consumer codelets who consume the same data tokens produced by codelet P.

For codelet graphs with multiple consumers, a naive solution is to use separate FIFO buffers for each arc in the codelet graph. However, this will lead to duplication of buffers leading to inefficient management of LCCM where storage space is premium. In Single Producer - Multiple Consumer cases all consumers of the same producer can share the FIFO Ring Buffer. An optimization like this was earlier proposed by Ning [11]. This is realized by utilizing multiple head pointers - one for each consumer codelet. The tail pointer will be updated by only the producer codelet. These pointers will be maintained in the registers of each CU, while the actual multiple-head FIFO ring buffer will utilize LCCM.

Figure 6b shows a conceptual overview of a multiple-head FIFO buffer. The operations of multiple-head FIFO buffer can be summarized as -

- Codelet P is the producer codelet. It writes data to the multiple-head FIFO buffer and updates its tail pointer in register.
- C1, C2 & C3 are consumer codelets. All three codelets are consuming data from different locations in FIFO ring buffer. Once they consume the required data, they update their corresponding head pointer in their corresponding register files.

5 Future Work

We need to evaluate the hardware-software co-design methodologies proposed in the earlier sections. Based on those results, we need to come up with metrics for hardware-software trade-off for each individual aspect. To successfully realize these evaluations there has been an on-going effort [12], an FPGA-based prototype in combination with general-purpose multi-core chips, where we can test the trade-offs of various proposed approaches.

The first step will be to simulate the proposed extensions and verify our hypothesis. We would also like to explore the possibility of extending the existing shared memory based codelet model runtimes like DARTS [13, 14] to demonstrate advantages of dataflow software pipelining.

In our current work, we have a simple coarse grain definition of dataflow software pipelining as explained in section 3. In the future, we would work on extending this formulation to incorporate more realistic and general cases of codelet graphs based on the real applications.

We are currently working on, identifying possible kernels, benchmarks and applications to best demonstrate the advantages of dataflow software pipelining for codelet model as expressed in this work.

6 Conclusion

In this paper, we have focused our attention into improving communication channels in-between codelets to support dataflow software pipelining. To realize this, we proposed extensions to the
codelet program execution model and proposed further extensions to codelet abstract machine to support dataflow software pipelining efficiently. We investigated various approaches to efficiently support data FIFO buffers by using a hardware-software co-design approach at codelet abstract machine level. We have laid down fair ground work to pursue future in depth research to effectively support dataflow software pipelining for codelet model.

Acknowledgment

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


