SEQUENTIAL CODELET MODEL
A SUPERCODELET PROGRAM EXECUTION MODEL AND ARCHITECTURE

by

Jose M Monsalve Diaz

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering

Winter 2021

SEQUENTIAL CODELET MODEL
A SUPERCODELET PROGRAM EXECUTION MODEL AND ARCHITECTURE

by

Jose M Monsalve Diaz

Approved: ____

Guang R. Gao, Professor Emeritus
Professor in charge of dissertation on behalf of the Advisory Committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____

Rudolf Eigenmann, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____

Xiaomin Li, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____

Sunita Chandrasekaran, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ______________________________

Kalyan Kumaran, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

This work is the result of several years at the CAPSL research group, under the supervision of Professor Guang R. Gao. I joined CAPSL in 2013 as an intern, and I never imagined I would stay for this long. Yet, I enjoy every single moment of my Graduate School. Professor Gao has been an inspiration. His work and ambitions, as well as his many hours of insightful conversations were crucial for the creation of this thesis. I also would like to acknowledge the patience and help from other senior CAPSL students through my PhD. In particular, Elkin García, Aaron Landwehr, Kelly Livingston, Joshua Suetterlein, Jaime Arteaga, Pouya Fotouhi, and Sid Raskar. As well as former postdoc Stephane Zuckerman, who I admire deeply, and who has been essential to the CAPSL group even to this day. Additionally, all the new students that I have been collaborating in the past three years. Ryan Kabrick, who brought two amazing students with him, Dawson Fox and Matthew Matusek; and Diego Roa, my old friend who also helped me craft the Matrix Multiplication example presented in this work.

I would also like to acknowledge Professor Sunita Chandrasekaran, one of the best professors I have met. The professionalism and dedication to her students are an inspiration. It is an honor having work with her for the past few years. Her personal and academic advise help me dealt with many difficult moments.

The University of Delaware, and in particular, the Electrical and Computer Engineering Department has supported me in many different ways. While it is difficult to mention everyone by name, I feel obliged to recognize Professor Kenneth Barner, Cynthia McLaughlin, and Gwen Looby, who bore all my crazy administrative questions and ideas. Bryan Youse, and Andrew Roosen, two excellent bosses during my ECE staff days. Amber Spivey, and Wendy Scott, who were always nice, greet me with a smile and help me out whenever I needed it. Additionally, I must acknowledge all my other professors in the department, specially Xiaoming Li, Chengmo Yang, Rudolf Eigenmann, and Andy Novicin. I have learned a great deal from each of you. Likewise, other offices in the University such as the Office of International Students and the Office of Graduate and Professional Education. In particular, Dr. Mary Martin who supported me and helped me graduate twice.

The past couple of years at Argonne National Laboratory have also played an important role in this thesis. I would like to acknowledge Kevin Harm's active participation in this project. He listened to me, and always asked the right questions. Dr. Kalyan Kumaran who also provided valuable input and supervision.

Finally, and most importantly, I must acknowledge my Family. My parents and my brother who have always given everything for my education and my future. They will always be an inspiration of hard work, dedication and love. Also, my wife, Luisa, who has brought me joy and support for many years, and specially in the past year when working on this thesis. Thanks for being there in every up and down.

I gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. Additionally, 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. This work is also supported by the National Science Foundation, under award SHF-1763654, as well as other CAPSL grants.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

ABSTRACT

In sequential computers, the Instruction Set Architecture provides a clear division between software and hardware. Separation of software and hardware through a well defined contract enabled decades long of seamless evolution of computer systems. The end of Dennard's scaling and slow down of Moore's law has forced architects to abandon purely sequential architectures in favor of parallel/distributed and heterogeneous systems. The new era represents a new spring of computer architectures. However, the ISA contract has been broken. It is mandatory to reconcile the abstraction between hardware and software in order to recover performance, portability, and programmability.

Sequential architectures take advantage of Instruction level parallelism to overlap the execution of instructions. These techniques use dataflow to implicitly perform side-effect free parallel execution of code. On the other hand, parallel programming often requires explicit reasoning of workload distribution, communication, memory synchronization and worker management. This thesis proposes the Sequential Codelet Model, a program execution model for parallel, heterogeneous and distributed execution of programs. It defines a machine abstraction (namely hierarchical Von Neumann machine), that recognizes the natural hierarchical structure of computer systems. Programming of the machine uses a hierarchical imperative programming model reassembling an Instruction Set Architecture at each level. A Codelet is the name given to an "instruction" of a level, as expressed in terms of instructions of the level below. By means of Instruction Level Parallelism inspired techniques, parallel/distributed execution of programs is achieved. The final system leverages the vast progress made for sequential computers. Finally, We present a the Super Codelet Architecture, a possible realization of the Sequential Codelet Model.

Chapter 1
INTRODUCTION

Computer systems are highly complex machines composed of a large number of parts interacting with each other. Throughout history, hardware and software have constantly evolved aiming to push the limits of what computers can do. Computer architects have competed to define high performance computing organizations with faster processing capabilities. On the other hand, software developers have design a large infrastructures that eases programming and utilization of computer systems. In general, computers are expected to deliver increased performance in every new generation, and software is expected to be easy to understand, create and extend while utilizing most of the available resources. Finally, rapid evolution requires re-usability across generations of systems, enabling a seamless evolution. These three aspects: Performance, Portability, and Programmability (or just 3P) have continuously driven computer innovation. For years, sequential computing provided a system abstraction and execution model that fostered these three properties. However, the arrival of parallelism has proven difficult to maintain them.

Following the advice of Hennessy's and Patterson's Turing Award lecture [1], we shall revisit some of the history of computer architecture. There are three major creations that lead to the sequential computing success. First, the Turing Machine introduced by Allan Turing in 1936 [2] which provided the necessary mathematical model. Second, the definition of the Von Neumann architecture in 1945 [3] that narrowed down the components of the computer system and their interaction into a specific system organization. And third, the introduction of a standardized Instruction Set Architecture as a single interface between hardware and software in the 1960's [4] by Frederick Brooks and its IBM/360 design team.

The standardization of the Instruction Set Architecture (ISAs) worked as a long lasting contract between hardware and software. As long as this contract was respected, each part could independently evolve its capabilities. On one hand, hardware designers focused on improving processor's frequency of operation and scaling down feature size of transistors. Higher frequencies and more transistors meant increased throughput and complexity. Software developers, on the other hand, were relieved of the need to constantly adapt their programs to new systems. As long as both hardware and software used the same ISA and respected the architecture's execution model provided by the hardware (i.e. Turing Machine and Von Neumann model), they could evolve programming models and compilers continuously.

ISA is almost a synonym of the system's architecture. There are multiple possible implementations of the same ISA, each with different characteristics. However, given an architecture, it is possible to execute the same program in every implementation that follows exactly the same ISA (i.e. Portability). Some of the most advanced implementations that are available today parallelize and re-arrange the execution of instructions without interfering with the running software or its correct result. A parallel execution of instructions is achieved through dataflow inspired techniques in a process that is completely transparent to the user. For a subset of the instructions in the program stream, dependencies are discovered and maintained at runtime, allowing for the order of execution of instructions to be changed. Furthermore, as long as instructions are committed in the order they are described (i.e. program state changes are performed in-order), the user is not aware (and does not need to be aware) of the runtime execution order. It is still possible to increase performance for a given implementation of an ISA. An avid programmer may change their code to exploit these execution mechanisms, but even so, the developer is not required to think of the potential hazards of the parallel execution.

The end of Dennard's scaling [5] and slowing of Moore's law during the past 2 decades [1] has limited the ability to improve sequential architectures through the same mechanisms used by the industry for over 30 years. Consequently, computer architects were forced to adopt parallelism, at first, by means of increasing the number of cores present in the same system (i.e. multicore architectures). Most recently, architects have shifted towards heterogeneous systems composed of multiple units each specialized in different computational patterns or application specific operations. Although this seems like an reasonable step, maintaining the Von Neumann abstraction over multiple cores has presented difficulties and challenges, specially when aiming to maintain the aforementioned 3P properties. We will discuss this further in the next subsection. The new architectures have revived interest in parallel computer architectures that already existed in the 70s and 80s, albeit its lack of success in the past. Nowadays, it is an accepted fact that computer systems must be parallel, heterogeneous and distributed machines.

Unlike sequential computing, trending parallel processing lacks of a broadly acceptable common abstraction that allows a division between software and hardware (similar to an ISA). Hardware evolution is usually slower than software due to its complexity. In order to utilize the newly introduced parallel hardware (i.e. multicore and heterogeneous systems) software (and developers) have been burdened with the task of creating this abstraction. Three big changes arrived to software. First, the concept of thread was introduced in the operating system. Second, programming models and languages were leveraged to explicitly express parallelism. Third, runtime systems were introduced to emulate machine abstractions and organizations. Nowadays, parallel programmers are responsible of workers creation, workers synchronization, and memory management across workers.

This thesis aims to provide a system and program organization for highly parallel, heterogeneous and distributed systems. This work intends to leverage the sequential computation abstractions (i.e. Turing Machine, Von Neumann model and Instruction set architectures) and extend it beyond the core unit. It defines a model of computation for parallel program execution that spans across the whole computer system infrastructure. The aspiration is to take advantage of the original sequential abstraction to exploit parallelism through ILP techniques, but for the execution of tasks that define more complex mathematical operations.

1.1 The problem with trending parallelism

The problem of parallelism, in general, is one of creation, communication and coordination of workers, tasks and resources. In addition, there are three requirements for general purpose parallel computing: 1) The program execution must use the available resources efficiently (i.e. high performance), 2) performance must be carried through multiple system generations and system architectures (i.e. performance portability), and 3) it is easy for a programmer to describe such program, reducing time-to-solution (i.e. programmability). Current trending parallelism relies on two major architectures: Shared memory multi-core systems and accelerator-based systems (e.g. GPGPUs) that use a SPMD execution model. Furthermore, there are currently two trending communication mechanisms across workers: 1) Message Passing (e.g. MPI) and Shared memory. I focus here on these four elements.

The problem with trending parallelism, in particular, is twofold. First, the lack of a common abstraction, namely Program Execution Model (PXM) as previously defined [6][7][8]. Second, current abstractions aim to maintain a Von Neumann view of the system. A PXM describes the execution of a program in a parallel system as a whole, and it should serve as a common view for hardware designers, system software developers, and programmers. A PXM allows for a single strategy to be used in order to represent and execute programs in the system. Such abstraction is similar to the role that the Universal Turing Machine, the Von Neumann model and the Instruction Set Architecture had in sequential system. All of which allowed an smooth evolution of hardware and software in sequential computer systems. The following paragraphs aim to understand the evolution of parallelism in the recent years and explain why these two problems are critical for the evolution of computer systems.

1.1.1 The evolution

Parallelism is not a new idea in computer architectures. It dates back to the early times of computer architectures, but contrary to the early years of parallel computing, mainstream parallel systems are now easily available. Modern parallelism arrived to an already existing infrastructure (as seen in figure 1.1) that relies on general purpose computation and urges backwards compatibility. With the decay of Dennard's Scaling and slow down of Moore's law, architects needed a solution to continue growing computational power of systems. Single core performance advancements hindered, and systems started featuring multiple cores (or independent threads in the case of SMT) side by side in a single die. This approach was simple for hardware development and allowed to keep backwards compatibility with existing software infrastructure. Furthermore, it was easy for the operating system to trade concurrency for parallelism when applications were running independently.

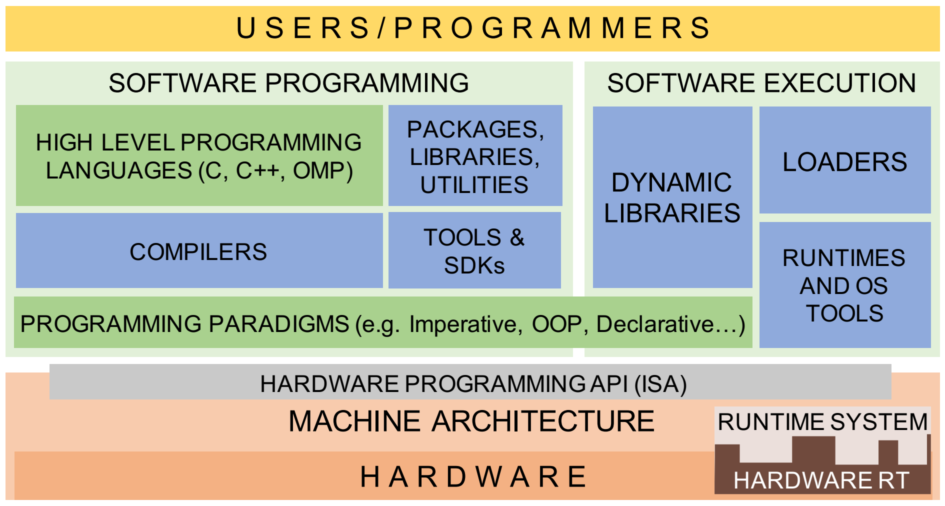


Figure 1.1: Conceptual view of computer system infrastructure nowadays

Multicore architectures are the simplest form of parallelism. They comprehend a number of computational units (namely CPUs or hardware threads) that are independently programmable workers. That is, the inner operation of a single worker does not inherently affect the operations of another worker. To coordinate execution of parallel programs in multicore systems additional communication mechanisms are necessary.

Given the already existing influence of Von-Neumann architectures, memory is the preferred method of communication in multi-core systems. Cores use reads and writes to memory to share information and influence program execution of other workers. Consequently, the order of memory operations have an impact on the communication across cores, and therefore on the overall execution of programs. The most common multicore abstraction is composed of multiple homogeneous cores all connected to a monolithic memory. Although memory is often seen as a single unit, memory organizations are not monolithic. Due to performance benefits, memory is divided into different physical locations, allowing a single memory address to be potentially located in different physical locations. The most common example is cache hierarchies.

A more recent system design trend are accelerators (or co-processors) featuring specialized hardware that optimize execution of specific sections of the program for a sub set of applications. Accelerators have proven that a heterogeneous system could provide great benefits towards computational performance. Perhaps the two most common type of accelerators right now are GPGPUs, and brain-inspired neuromorphic accelerators such as Google's Tensor Processing Unit [9] or IBM's TrueNorth project [10]. The latter have shown interesting results for brain-inspired applications. Accelerators are architectures that are created to exploit performance given certain characteristics such as optimizing application specific operations, or enabling data parallelism in hardware. When a program fits the execution model of the accelerator, its performance is improved.

Systems featuring GPGPUs have been successful when it comes to data parallelism and are the current trend in the fastest supercomputers of the world according to the Top500 list [11]. These systems heavily borrow from vector machines [12] and use Single Program Multiple Data (SPMD) and Single Instruction Multiple Data (SIMD) execution models. In these models all the threads receive the same stream of instructions, but a unique identifier of the thread allows for accessing and processing different data in each thread. GPGPUs contain hundreds or even thousands of simplified cores that are relatively simpler and slower than the CPU counterparts. Cores are also grouped into set of cores that usually share memory and other resources. Multiple groups of cores are placed into the same device, and it is possible to use multiple devices. The structure of GPGPUs is a hierarchical and programs usually adapt to this hierarchy in order to utilize the whole system.

Accelerators often feature non-traditional ISAs, therefore, they rely on commodity hardware, usually referred to as the host, to start and coordinate the execution of the program. Under this model, certain segments of computation are offloaded to the accelerator, while the creation and coordination of program is left to the host.

Parallelism in software has evolved with parallelism in hardware. Prior to the arrival of mainstream parallel systems, operating systems (OS) already supported concurrency of different processes. The use of virtual memory and time-slicing scheduling techniques allows independent programs to share system resources transparent to the user. Communication between processes requires the intervention of the OS. Additionally, due to the isolation of processes, messages must make use of intermediate storage such as reserved kernel address space or I/O. Processes are software representation of the von Neumann machine, while the OS guarantees its mapping to the underlying hardware. In multi-core systems it is easy for an operating system to trade concurrency for parallelism. The independent behavior of processes maps to that of CPU cores in multicore systems. Adaptation between single-core and multi-core OS runtimes featured a parallel scheduler with restrictions on shared resources coordinated by the single kernel. In the absence of synchronization between instruction streams of different processes and their respective memory spaces, the OS is the only mediator for the communication between processes and resource sharing, reducing potential side effects to specific interactions that obey a strict sequential order.

Message passing interfaces (MPI) was created to enable explicit communication across processes, reducing the intervention of the operating system. Thanks to process isolation, MPI can easily spawn across different compute nodes by using already existing networks to distribute messages. Since memory is independent for each process, MPI requires interaction between the program's user space and the operating system's kernel space for communication. Furthermore, when spawning across nodes, this communication also involves different drivers and hardware. All these makes cooperation between processes expensive, resulting in a penalty to performance [13, 14], that must be accounted for when parallelizing software.

Multi-threading programming allows creation of multiple threads on the same process, sharing the same resources such as memory address space. Multi-threading removes the shield across processes placed by virtual memory, while at the same time reducing the operating system's role and interaction during program execution, potentially reducing overhead and providing more performance. The number of threads in a process do not have to map the number of physical cores in the system. Instead the operating system can concurrently execute threads with the means previously described. However, it has been shown that over-subscription of threads that compete for the same resources hampers performance [15], and concurrent execution of threads may leave to potential deadlocks, if not handled correctly.

Multi-threading still relies on a Von Neumann view of the system with a monolithic perspective of memory. CPU core's performance heavily relies on the use of cache-like memories (among other pre-fetching mechanisms) that reduce latency of memory access. Caches are transparent to the user and are not part of the software thread abstraction, but a given unique memory address in the thread's memory space may map to different physical locations in hardware. Shared memory systems on multi-core architectures introduce a problem: Out of all potential memory locations, and their respective values, what is (are) the correct value (values) a worker can observe at a certain moment of time? Memory models are used to determine the set of admitted values in the presence of multiple workers (observers and producers) [16].

Multi-treading requires programmers to reason about all potential side effects that could occur in memory during the execution. Order of events is critical for synchronization of workers. Relaxed memory models with large set of correct values increase potential side effects. Therefore, most of the commodity multi-core systems available nowadays have been design to respect the sequential consistency memory model [17].

Cache coherence protocols are built around the memory hierarchy to maintain the illusion of an ordered sequence of memory operations that allows a single value to be observed by every worker at a given time. Additionally, different flavors of atomic memory operations are often provided by these systems to allow for a more strict definition of the order of memory operations.

With the arrival of commodity parallel systems, it was still necessary to adapt previously existing high level programming models and programming languages to be able to support parallelism. Software threads are managed by an OS libraries and kernel functions (e.g. POSIX Threads). An application can request and manage thread resources through different calls exposed as a system library. Software and hardware threads are flexible and allow to be building blocks of more complex models. Furthermore, they enable backward compatibility with an already existing software infrastructure. Therefore, high level programming languages would only require a library or module that connects the OS API with the particular semantics of the programming language.

Consequently, mainstream programming models have been extended with semantics to express parallelism. Extensions have been proposed to define creation, communication and coordination of workers, tasks and resources. Currently there is no widely acceptable parallel programming model, and in order to take advantage of the parallel resources there is a myriad of frameworks that implement different programming models for parallelism. Multi-threading, for example, provides full flexibility, but the burden lays completely on the programmer [18]. Another example is the Fork-join models. The program starts sequentially and, at a certain point in computation a number of workers are spawned. Once they have finished executing they join into a single barrier and sequential execution resumes. Workers may have local private memory and there may be global shared memory. Depending on the implementation, communication is allowed across workers, but it must be coordinated by the programmer. Yet another example of a programming model is tasking. This model is heavily inspired by dataflow models of computation. tasks are segments of code that are executed when their dependencies are satisfied and the necessary resources are available. There are data dependencies defined as a producer-consumer relationship, and control dependencies, defined as the order in which tasks need to be executed. Tasking usually defines a graph of dependencies either at runtime or statically at compile time.

Accelerators have also been considered. Accelerators often come with low level languages that extend from already existing languages (e.g. Cuda, HIP, or SYCL). Additionally, other programming languages and frameworks (e.g. OpenMP, and OpenACC) have been extended to account for heterogeneity, allowing the user to express computation targeting the given architecture. Finally, there are libraries which provide application specific operataions which ultimately use the accelerator as part of its implementation (e.g. TPU Tensorflow, cuCNN, and Intel's MKL).

1.2 The problem

Sequential computing built up upon two models: The Turing Machine and the Von Neumann architecture. These models allowed a common view of the system the between hardware and software development, provided a solid ground for the evolution of computer systems. However, the introduction of parallelism has shaken these abstractions.

Multicore architectures enable backwards compatibility by using already existing ISAs in each core. However, these system struggle to maintain the 3P properties. Programmability suffers from two angles. First, independent worker creation and synchronization is tedious. Second, as memory operations affect the result of the computation, it is often necessary to enforce an order in memory operations to easily understand program behavior. Consequently, multicore systems use mechanisms to guarantee this order operations (e.g. cache coherency protocols) which ultimately degrade performance as the number of cores increases. Performance is also heavily dependent on other system's elements which are often left outside of the multi-core abstraction (e.g. cache organization, memory bandwidth, interconnect architecture, and memory access times). The lack of accountability for these concepts in the software abstractions ultimately degrade portability when these elements change from system to system.

Some accelerators have proven to improve performance by reducing the overhead, increasing the number of workers and allowing different execution models to be used. In addition to performance, programmability also improves with accelerators as they ease worker creation and coordination. However, the host-accelerator machine abstraction centers its execution into the accelerator and usually leaves behind other compute capabilities such as the host. This problem is exacerbated by the distance between host and device, leaving the host as a simple scheduler of computation. Finally, there host-accelerator machine abstraction has not been formalized to allow portability across generation of systems.

Regardless of the programming model, there needs to be a mapping in between the abstract machine of the hardware, and the abstract machine of the programming model. With the lack of a common abstraction at the hardware level, parallel programming models have been tasked to come up with abstractions on how to organize computation to take advantage of parallelism. Due to the aforementioned evolution of hardware and parallelism, most of the programming models rely on runtimes that are software implemented and linked against the OS libraries that provide thread creation and management. These implementations use hardware threads, software threads and processes as building blocks. More often than not, these mappings assume their runtimes are the only running element in the system, and if placed next to other runtimes they tend to compete for resources. Moreover, attempting to mix these models in the same program tend to have similar complication (e.g. MPI+X). Therefore, current parallel programming models tend to break programming generality as defined by Dennis in [7]. Coming up with a common abstraction which can be implemented in hardware is crucial to solve most of these issues that come from the excessive freedom given through current parallel systems.

Parallelism is hampered by its increased difficulty in programming. As previously mentioned, programmers are required to think of the interaction between workers through the use of shared memory or message passing. Message passing is tampered by its overhead and limitations to express complex programs that are also portable. In shared memory there is no implicit synchronization or coordination between threads, or hardware computational units. The programmer must build and use mechanisms (e.g. mutex primitives, critical regions and atomic operations) for synchronization of workers, coordination of tasks and assignment of workloads. An excellent explanation of this issue is presented by Edward Lee's paper "The problem with threads" [19]. These issues become more complex and prominent as the number of cores and nodes increases.

In hardware, scalability is hurt by the need to maintain the von-Neumann representation and the underlying sequential memory model needed to ease programmability. Aiming to relax the memory model becomes prohibitively expensive for the programmer as it would increase side effects during execution of programs. This could overwhelm the most avid developer, which would have to account for it on top of the requiring thread creation, synchronization and workload assignment. Multi-core systems with relaxed memory models and with non-coherent memory hierarchies have failed to succeed so far.

GPGPUs are a good example of how hardware implemented parallel abstractions can considerably reduce the overhead introduced by operating systems and the different runtimes. Moreover, it shows the need for heterogeneity and parallelism in order to progress computation. It also reduces the developer's burden when it comes to workers and tasks creation and orchestration as well as resource management that is usually present in multi-threading programming. However, and despite considerable advances in their architectural designs, GPGPUs still struggle with unstructured parallelism as well as for applications that cannot easily exploit data parallelism. Not only that, but the offloading paradigm that sees the CPU and GPU as two distinct units that are far from each other has reduced the participation of CPU cores as important elements for low latency computation. It is still desirable to recognize the strength in heterogeneity as a collection of elements with different compute capabilities. An abstraction for general computation should consider these aspects.

Operating System processes and threads abstraction enforce a Von Neumann view of the system even in the presence of accelerators. Likewise, programming models have historically evolved from sequential computation, but they may not be suitable for parallel abstractions [20]. It is often desired to have a unified shared memory to maintain the Von Neumann abstraction in a software infrastructure built for sequential computers. However, parallel programming has already asked software developers to completely re-structure their software to account for heterogeneity and parallelism. For example, while it is possible to use complex classes and data structure abstractions in the GPU, they are often not encouraged in favor of performance. Moreover, these devices have created deep memory hierarchies with different memory regions that are only shared with a subset of the system. These different memory regions also require extra modifications to the code, as well as the program execution strategies that are not trivial. If the system is to remain under the original Von Neumann abstraction, the monolithic view of memory, together with the oversimplification of the system's architecture (e.g. interconnection networks, memory hierarchy and non uniform memory accesses) would make it impossible for performance, portability and programmability to be recovered.

Bottom line, as the contract provided by common ISAs broke with the introduction of parallelism in computer systems, there is a constant struggle to adapt the already existing software ecosystem into the available hardware resources. Both hardware and software have aimed to find a new contract to replace the old one. Software and its ability to adapt fast, has come up with many solutions, but they still rely on heavy software implemented runtimes and a constant push for a Von Neumann view of the system. These runtimes enforce different models that collide, and have not proven to provide the three desired features: Programmability, performance, and portability. As the trend into accelerators rises, they provide valuable opportunities, but they also hardened the creation of the desired program execution model and its abstract machine. This work aims to provide a feasible solution to some of these problems.

1.3 Synopsis

The rest of this thesis is organized as follows. Chapter 2 contains background information.It discusses the basics that influenced previous models of computation as well as the Sequential Codelet Model. Some of the topics include the Turing machine, Von Neumann architecture, dataflow computation, and concepts of parallel computing based on these three. It also introduces the original Codelet Model as the predecessor of the Sequential Codelet Model. Next, Chapter 3 summarizes the objectives of this thesis, as well as provides the problem formulation.

Having introduced the topics and intention of this work, this book proceeds to describe in depth the main contributions of this Thesis. Chapter 4 describes the Sequential Codelet Model of Computation. It starts by introducing the Hierarchical Turing Machine as the underlying mathematical model that inspires the creation of the Sequential Codelet Model. Then it discusses the Hierarchical Von Neumann abstract machine, and the execution model based on it. Parallelism is described as an artifact of the execution of the program, but not inherently part of the original execution mode. An architecture is defined using the Sequential Codelet Model in Chapter 5. This architecture is a possible realization of the Sequential Codelet Model in a real hardware system. To evaluate our approach, Chapter 6 describes the creation of an emulator that allows execution of SCM programs into commodity hardware architectures. This emulator is named SCMUlate (pronounced S.C. Emulate). This Chapter not only describes the structure of the SCMUlate, but also applies the hardware/software co-design principles in its design and implementation. Additionally, this chapter describes how a developer and user can simulate the Sequential Codelet Model with SCMUlate. Based on SCMUlate, Chapter 7 shows evaluation results and analysis on two different benchmarks: Vector Addition and Matrix Multiplication. These benchmarks are well known kernels that are particularly important for High Performance Computing computation, both in data sciences and scientific code. They have been selected thanks to the experience gather when contributing and working with ECP

SOLLVE applications (an important project for the future of HPC and Exascale computing) during the development of the OpenMP offloading Verification and Validation testsuite [21][22]. “Even when standardization and initial implementations of OpenMP features are complete, the SOLLVE project’s efforts continue. With an ever-growing OpenMP validation and verification test suite, SOLLVE monitors how well compilers and runtime systems support OpenMP across pre-exascale and exascale test bed systems. This provides guidance to the implementers and facilities and ensures the portability of OpenMP with regard to compilers and systems” – as recently reported in [23].

Finally, this thesis presents some related work in Chapter 8, it discusses future work in Chapter 9, and provides conclusions in Chapter 10.

Chapter 2
BACKGROUND

In the quest of creating a program execution model, it is necessary to have a holistic view of computer systems from all different layers. Computer systems are a myriad of components and layers that interact with each other. The background needed for each of these layers can comprehend several books and requires extensive knowledge. However, in this chapter I have selected some of the elements that I have found most important for this work.

As the title suggests, this section is divided in two parts. First the Foundation Section 2.1 that contains fundamental concepts of models of computation. We introduce the Turing Machine, Universal Turing Machine, Von Neumann architecture, dataflow models of comptutaion and others. Second, Section 2.2 summarizes the original Codelet Model, its components and elements. This model is the predecessor of the Sequential Codelet Model. These two sections aim to cover all the different areas that one way or another inspired this work.

2.1 Foundation

Following are some of the most important and influential concepts for this work.

2.1.1 The Universal Turing Machine

The most important model of computer systems is the Universal Turing Machine. This mathematical model is the foundation of most computer systems that are used nowadays. Alan Turing (1912 - 1954) was an English mathematician (among other expertise) that is recognized as the founder of theoretical computer science and artificial intelligence. Turing originally described his machine in [2]. This paper's objective was not to define the machine, but the machine was an utensil to demonstrate that the Entscheidungsproblem had no solution. While this conclusion was also reached by Alonso Church [24, 25], Turing's work had a different approach that involved the definition of a theoretical computing machine which was unique of its own.

At a high level, Turing Paper has four major contributions: 1) The definition of the Turing Machine, 2) The definition of computable numbers, 3) the definition of the Universal Turing Machine, and 4) the demonstration that the Entscheidungsproblem has no solution by using the other three contributions. This work mainly focuses on the Turing Machine and the Universal Turing Machine. For a more comprehensive understanding of the whole paper, it is worth visiting [26].

Turing Machine

The Turing Machine is inspired by the process a person calculating a real number with a pen and a piece of paper. When a human is calculating a number, he or she knows the set of steps or rules that need to be applied according to the values that are read from the piece of paper. Then, the person will move along the paper, read the values from the paper and write new values to the paper. The Turing Machine is an automatic process that does not require external intervention. It is used to calculate real numbers by only following a set of steps. The Turing parts from three elements that are parallel to the resources used in the human process described above: 1) A tape that stores information in the form of symbols (Paper). 2) A head that reads and writes into the tape (Pen), and 3) a set of rules (ordered steps), that determine the position in the tape as well as what to write into the tape according to what is read from the tape (The algorithm in the mind of the person). These rules are referred to as configuration states or `m-configurations` of the machine, and a given machine can only have a finite number of states. Therefore these rules reassemble a Finite State Machine. Figure 2.1 shows the different components of the Turing Machine.

The process is simple: The `tape` is easily represented as a one dimensional infinite tape divided into `squares` and capable of storing `symbols`. The `head` is located

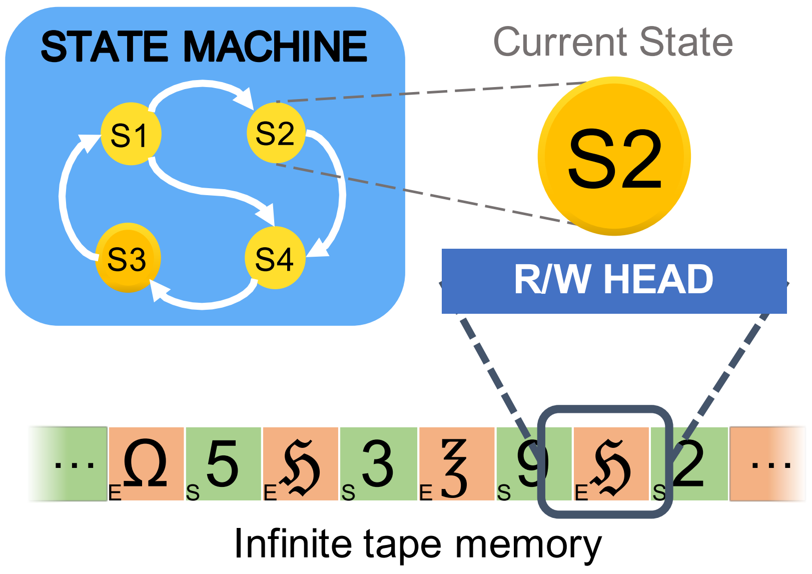


Figure 2.1: Turing Machine. There are four different states S_1 through S_4.The tape shows the two different type of `Squares`: `E` and `S`

in a given position of the tape. A `symbol` (or the lack of) is read from the current position and based on the current `m-configuration` a set of operations are performed in order. These operations are a combination of: Moving the head *Left* or *Right*, *Print* a new value in the current position or *Erase* the value from the current position. After the operations are performed, the machine goes to a different `m-configuration` state. Notice that initially, there is no limit on the number of operations that can be performed for a given state. While this is true for the Turing Machine, this changes in the Universal Turing Machine, as will be explained later in this section.

The `Symbols` written in the machine are of any kind. However, due to the interest in numbers, there are two kind of symbols. Numeric symbols that represent the value being computed, and non-numeric symbols that are used to mark properties of the computation. The latter do not aim to represent values, but to `mark` additional information at given positions of the tape (e.g. positions that have been already visited). Therefore, the tape will contain one type of `squares` called Figures or `F-Squares` that store numeric values containing the number being calculated; and it contains another type of `squares` called Erasable or `E-Squares` that contain the non-numeric symbols

m-config	symbol	s-operations	next m-config
	$None$	$P0$	b
b	0	$R, R, P1$	b
	1	$R, R, P0$	b

Figure 2.2: Example of an `m-configuration` table of the Turing Machine

that represent marking information of the `F-Squares`. In Turing words, these squares are “notes to assist the memory”. The location of `E-Squares` and `F-Squares` is not important, as long as it is possible to pair one to another. Turing decided to alternate one another (e.g. `E F E F` ...) assigning an `S-Square` the `E-Square` to the left. The need for more `E-Squares` to mark `F-Squares` can be solved by introduction more symbols to represent different information. Figure 2.1 shows the two different `Squares` in green and red as well as with a subscript `E` and `S`.

To represent the Finite State Machine, Turing used the concept of `m-configuration` tables. These contain 4 columns. The first column contained the name of the `m-configuration`. The second column contained the symbol read by the tape and which would determine what operations to perform. The third one containing the operations to perform. Four different operations are used here: P_X for Printing the `symbol` X, R/L for moving the head one `Square` to the right or left respectively, and E for erasing the content of the `Square`. Figure 2.2 contains an example of an `m-configuration` table that would print “0 1 0 1 0 1 0 1 0 ...” in the tape. If we consider that this number represents the decimals of a binary number written as $0b0.01010101$... it is equivalent to the number 1/3 in decimal.

Example of implementing copy on a Turing Machine

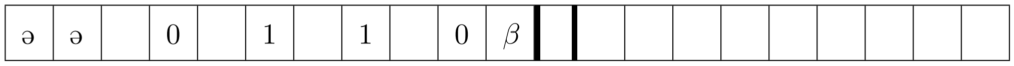
Let us consider the Turing Machine with the `m-configuration` tables of Figure 2.3. This machine will copy a sequence of symbols from the beginning of the sequence to the blank spaces at the end of the sequence. We will call it `copy`. It uses binary numeric symbols 0 and 1 since binary numbers are easier to use than decimal numbers

`m-config`	`symbol`	`s-operations`	`next m-config`
begin	Not ə	L	*begin*
	ə	R	*copy*
copy	BLANK	Px, R	*copy*
	0	R, R	*copy*0
	1	R, R	*copy*1
	β	E	*following*
*copy*0	0 or 1	R, R	*copy*0
	BLANK	$P0, L$	*removex*
*copy*1	0 or 1	R, R	*copy*1
	BLANK	$P1, L$	*removex*
removex	Not x	L, L	*removex*
	x	E, R, R	*copy*

Figure 2.3: Example of the `m-configuration` table of a `copy` Turing Machine

(see [26] for an example of a Turing Machine using decimal symbols). It also uses three non-numeric symbols: ə, β, and x. The purpose of the əsymbol is to represent the beginning of the sequence in the tape, and it was originally used by Turing in his paper. The β symbol represents the end of the sequence to be copied, as well as the beginning of the resulting sequence. The x symbol is used to mark the current numeric symbol being copied.

Let us assume that the current total configuration of the machine contains the initial sequence as well as the β symbol at the end of the sequence as such:



This grid represents a segment of the infinite tape. The head location is represented by the thick lines surrounding a square. Let us start execution at the *begin* state. This state will take the head to the first left-most blank square after the ə, and it will call the *copy* state. The first execution of the copy state will find a $BLANK$ square, therefore printing an x and moving right. After this we end up with this tape configuration:

ə	ə	x	0		1		1		0	β										

The *copy* state is executed again, but this time the read symbol is different. We have already marked the symbol to be copied with an x, allowing us to keep track of which symbol we are currently copying. Following we start performing the copy of the 0 that is currently under the head. Notice that I use the state *copy*0 which allows us to have some sort of memory of the symbol we are currently copying. Finishing this state we are two blocks further right from the symbol we are copying, resulting in the following machine configuration, and our next state is *copy*0.

ə	ə	x	0		1		1		0	β										

The *copy*0 `m-configuration` is now being executed. This state will first find the left most numeric `F-square` that do not have a symbol in it. To do this, the state will continuously call itself while the value is not $BLANK$. Then, when the empty square is found, the head will print the value 0 in it followed by moving the head one position to the left and changing to the next state *removex*. Before changing states, and after several iterations of the *copy*0 configuration we end up with the following tape:

ə	ə	x	0		1		1		0	β	0									

We now move to the *removex* `m-configuration`. Notice that our head is currently located in an `E-Square`. We first need to find the x to be removed by iterating over the `E-Squares` to the left (i.e. L, L) until we find the x symbol again. This is achieved by constantly calling the *removex* state until the value under the head is an

x. At this point we can remove the current x and move two blocks to the right (i.e. E,R,R). We have finished copying a whole symbol, and we are ready to start the next copying process.

ə	ə		0		1		1		0	β	0									

Next, we have the *copy* state again. The value under the tape is a $BLANK$ once more, therefore printing an x symbol, marking the second digit of our sequence. We will start a new cycle. The configuration of the tape right before the new cycle is as follows.

ə	ə		0	x	1		1		0	β	0									

After going over the whole sequence and having copied all the digits, we are back again to the *copy* configuration. However, this time we will not find a $BLANK$, instead we read a β symbol. As a result we now have concluded the *copy* process, and we are ready to move on to the next state, namely $following$ in the `m-configuration` table. To do this, we first make sure to remove the β symbol. This is the final snapshot of the machine right before moving to the $following$ state. The head is currently where the β symbol used to be.

ə	ə		0		1		1		0		0		1		1		0			

This concludes the execution of the copy operation on a Turing Machine.

Skeleton m-configuration Tables

When representing `m-configuration` tables, one will easily encounter repeating patterns of `m-configurations` that contain similar operations (instructions), but different symbols or following `m-configuration`. Turing defined in his paper the concept of "`Skeleton Table`" that is somehow similar (but not completely equivalent) to a subroutine in current programming languages. The idea is to represent an `m-configuration` that has parameters that work as variables in the configuration table. The parameters will be either numeric and non-numeric symbols, or other `m-configurations` that.

Going back to the example in Figure 2.3 we notice that the *copy*0 and *copy*1 `m-configurations` are both exactly the same, but they differ in the the symbol it is printed as well as the following state to be called. By using `skeleton tables` it is possible to re-write this `m-configuration` table as seen in Figure 2.4.

`m-config`	`symbol`	`s-operations`	`next m-config`
$copyS(\boldsymbol{S})$	0 or 1	R, R	$copyS(\boldsymbol{S})$
	BLANK	$P\boldsymbol{S}, L$	$removex$

Figure 2.4: Using `skeleton Tables` for representing *copy*0 and *copy*1 `m-configurations` into a single table

`m-config`	`symbol`	`s-operations`	`next m-config`
copy(**following**)	BLANK	Px, R	*copy*
	0	R, R	*copy*0
	1	R, R	*copy*1
	β	E	**following**

Figure 2.5: Using `skeleton Tables` for representing the *copy* `m-configuration` table allowing different $following$ states.

The $\boldsymbol{S}$ can be seen as a numeric symbol that is a parameter of the newly created *copyS* table. The $P\boldsymbol{S}$ instructions will actually print the value that was sent to the *copyS* configuration. Likewise, *copyS* is recursively called with the same symbol until the $BLANK$space is found.

Furthermore, if imagine multiple uses of the *copy* `m-configuration`, where the $following$ state corresponds to different continuation states. Then, without using `Skeleton tables` we would have to duplicate the *copy* state over and over, using multiple names, and only changing the state that replaces the $following$ placeholder.

Instead it is better to have a single version of the *copy* configuration, that receives as part of the parameters the continuation state as in Figure 2.5.

By putting together the two elements in Figures 2.4 and 2.5, we can re-write the `m-configuration` table of example 2.3 as seen in Figure 2.6.

`m-config`	`symbol`	`s-operations`	`next m-config`
$copy(\textbf{following})$	BLANK	Px, R	$copy(\textbf{following})$
	0	R, R	$copyS(0, \textbf{following})$
	1	R, R	$copyS(1, \textbf{following})$
	β	E	$\textbf{following}$
$copyS(\boldsymbol{S}, \textbf{following})$	0 or 1	R, R	$copyS(\boldsymbol{S}, \textbf{following})$
	BLANK	$P\boldsymbol{S}, L$	$removex(\textbf{following})$
$removex(\textbf{following})$	Not x	L, L	$removex(\textbf{following})$
	x	E, R, R	$copy(\textbf{following})$

Figure 2.6: Using `skeleton Tables` to re-write example 2.3.

It is necessary to pass along the **following** state to be able to connect the inner calls of the *copy* configuration with the original *copy* configuration. On the other hand, even though it looks like an infinite recursion, it is important to understand that `skeleton tables` are not meant to be resolved at "runtime" by the Turing Machine. Instead they are meant to be for easier readability of the whole Finite State Machine. As a consequence, all the different states (`m-configurations`) that exist as a result of unrolling `skeleton tables` must be known. Furthermore, the amount of states must still remain finite.

The Universal Turing Machine

Currently it is known the importance of binary numbers. At the end of the day, computer programs are just long strings of ones and zeros that encode instructions in the system. This means that every program is a (fairly large) number. Such realization is also a result of Turing's work.

The examples presented so far do not limit the number of operations that are performed in each `m-configuration`. The Turing Machine was originally limited to a print or delete operation, followed by moving the head of the right or left or not at all. But even Turing himself realized how long `m-configuration` tables would be if you limit just to these operations. However, going back to the original restrictions, Turing could demonstrate that it is possible to build a machine that is able to execute other machines that are represented as numbers written in part of the tape. However, before it is possible to do this, you must be able to encode these machines into such numbers. Similar to Instruction Set Architectures, where bits are used to encode operations, Turing created a set of "instructions" built as standardized `m-configurations`.

`m-config`	`Symbol`	`s-operations`	`next m-config`	
q_i	S_j	PS_k, L	q_m	(N_1)
q_i	S_j	PS_k, R	q_m	(N_2)
q_i	S_j	PS_k	q_m	(N_3)

Figure 2.7: Standard forms of states in the Turing Machine

Figure 2.7 shows the three different forms that any state could use in standardized form. The symbols S_j, S_k can be any of the form 0, 1, $BLANK$, or any other predetermined non-numeric symbol. The use of $BLANK$ will replace the Erase operation used in previous examples. Finally, a single instruction can be concatenated together into a single string. For example it is possible to write N_1 as $q_1S_jPS_kLq_m$;. If each part of this string is encoded as a number (or collection of numbers), then it is possible to write any machine as a number encoding the operation.

Programs of the Universal Turing Machine are therefore encoded in numeric representations of the standard form (i.e. N_1, N_2 or N_3). Following, it is possible to write a second machine that interprets this standard form in order to execute the different states. A details explanation of this process is available in [26].

Computable numbers

As shown before, a Turing Machine has the potential to output different numbers. Each number represented by the sequence that results from the execution of the machine. Turing limited his study to the machines (and therefore set of `m-configurations`) that do not stall during execution, resulting in machines that yield to valid numbers (e.g. $0b0.010101000...$). It is possible to observe that a given valid machine will output a numeric value that is the representation of a real value. Furthermore, the set of possible states to be used in a Turing Machine must be finite. The set of resources that are needed for the machine to work in order to output the number is also finite. It is possible to show that all possible Turing Machines that could be built based on these restrictions are enumerable. Each machine will output a single number. The set of all numbers that can be a result of all possible Turing Machine are then defined as the computable numbers. Since the possible Turing Machines can be enumerable (see Turing's paper for details), the computable numbers is a countable set. The set of computable numbers does not contain all the real numbers, but it is a subset of the real numbers. Furthermore, this set is larger than the set of rational numbers, including some irrational numbers such as π and e. However, most of the irrational numbers are non-computable numbers.

This finding is of great importance and lead to the creation of the theory of computable numbers. However, the deep mathematical theory behind these numbers is outside the scope of this work.

2.1.2 Von Neumann Architecture

Von Neumann and Turing met each other in at least two opportunities [26]. One when Von Neumann visited Cambridge in 1935, and the second time when Turing visited Princeton from 1936 through 1938. Their interests in the field of mathematical logic and computation matched, and it is certain that Von Neumann was aware of Turing's work at the moment of working on this architectural model. The so-called Von Neumann architecture was initially described in a draft of the design of the EDVAC system in 1945 written by John Von Neumann. Although Von Neumann is the sole author of this draft, the document intended to be a design summary of the EDVAC project which involved more people. The manuscript was never officially finished. Von Neumann sent a handwritten draft to Herman Goldstine, and he distributed a typed version among the different members of the EDVAC team. However, due to the large interest it attracted, the document quickly spread across different countries before the final version was concluded.

Other than the architecture itself, the document talks about design trade-offs that were made in favor of a more general purpose computer. Among them there are the use of binary numbers, the selection of vacuum tubes over mechanical means, and the description of the basic mathematical operations needed for general purpose computing as well as the target different applications the system could perform. While these decisions were not original to his time, they were an important part of the design and to this day they are an essential part of computer systems.

First, binary numbers are used because they allow for simpler implementation of arithmetical circuits with higher switching speeds. Although using binary numbers requires larger execution times (in comparison to other numerical systems), it ushers simpler circuits capable of achieving faster execution times. Consequently, components used for building the system must be able to operate at high frequencies as well. [1] Second, instead of supporting complex arithmetic operations that require larger

[1] At the time of the EDVAC system, mechanical switches were available and widely used, but they were too slow. The newly available vacuum tubes had faster response

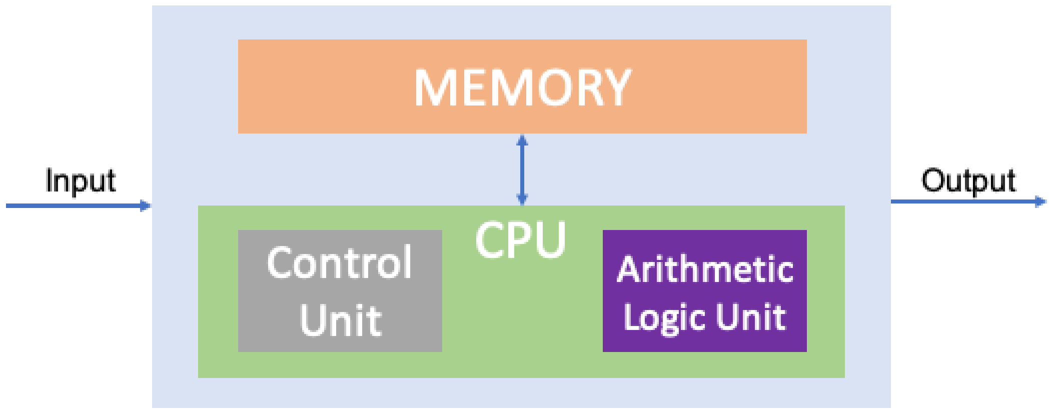


Figure 2.8: Von Neuman Machine Architecture

circuits, systems often expose only simple operations (e.g. add, sub, and mul). It is still possible to implement complex operations by means of simple arithmetic (or at least fair numerical approximations). After all, Turing's paper had already proven this with the set of computable numbers.

The Von Neumann Machine describes an organization of a system that implements the Universal Turing Machine. As can be seen in Figure 2.8, the Von Neumann machine is comprised of three major parts: A Central Processing Unit (CPU), a memory storage, and an input/output interfaces to the external world. Memory represents the tape of the Universal Turing Machine, and CPU represents the head. Program instructions are usually stored as part of memory in what is known as stored-program computer, which pre-dates the EDVAC system, and assimilates the Universal Turing Machine. Memory also stores program data. On the other hand, the CPU has two parts: 1) the Control Unit, which decides what instruction to execute next (e.g. resolving branches) and schedule it, And 2) the Arithmetic Logic Unit (ALU), which contains all the circuits that perform arithmetic operations. Therefore, the Control Unit schedule the instructions for execution in the ALU.

With the exception of loading the initial tape configuration, and reading the

times making them ideal. Later on, tubes were replaced with silicon transistors with higher frequencies, smaller sizes and better power performance.

result from the tape, the Universal Turing Machine is a closed system with no interaction with the outer world. Differently, the Von Neumann system I/O interfaces have an important role for the interaction with users and other peripheral systems outside of the machine even during the execution of the program. The I/O allows the system to communicate with the exterior world in different ways: like in the UTM, I/O interfaces allows the system to provide *input* configurations (programs and data), and obtain *output* results. But in current systems based on Von Neumann, I/O also has an important role throughout the execution of the program. For example fetching information from the outside world (e.g. internet), as well as communicating with many other hardware systems and peripherals that change the execution of the program. It is important to coordinate how I/O signals interface (or avoid interfering) with the current program execution, specially for the sake of deterministic behaviors.

The Von Neumann architecture benefits from sequential execution of program instructions. The program counter determines the current executing instruction. Some instructions, referred to as control flow instructions, explicitly determines the next instruction to be executed at a different location of the program by modifying the program counter. For other instruction types, the program always follows the next instruction in the stream increasing the program counter by one. For the original model, an instruction should not start before the previous instruction finishes. Thanks to this set of rules, there is no need to explicitly determine continuation instructions in the encoding, nor to have a matching mechanism between instructions. Producer-consumer data dependencies between instructions are respected by the order of execution.

An important observation is that memory is monolithic in the Von Neumann architecture. However, architects are required to create a hierarchical memory structures to support larger size and limit memory access latency. At the first level, next to the CPU and inside of the core's architecture, a register file contains several memory locations with a static naming convention. Each register is assigned a name and a predefined length and role (e.g. general purpose or configuration registers). Register are close enough to the ALU, therefore, register access time from the ALU is low in comparison with the ALU's operation latency. Nevertheless, register file size is limited by the die area and other architecture design tread-offs. Two or more levels, each with a larger memory capacity, can be built outside of the core. The most common memory architecture features a large and slow main memory (i.e. DRAM) directly connected to the CPU, and a hierarchy of memory caches in between that exploit temporal and spatial locality of memory accesses. Outside memory is often organized in fixed size chunks known as "words" that are independently distinguished by some addressing mode. Although there are many different possibilities, flat memory spaces are often used to respect the original concept of a monolithic memory space in the Von Neumann architecture.

The aforementioned memory structure is a key aspect to achieve high performance in Von Neumann based systems. As explained by Turing himself [27]:

> (...) I believe that the provision of proper storage is the key to the problem of the digital computer, and certainly if they are to be persuaded to show any sort of genuine intelligence much larger capacities than are yet available must be provided. In my opinion this problem of making a large memory available at reasonably short notice is much more important than that of doing operations such as multiplication at high speed. (...)

However, most of the properties that are essential for the Von Neumann architecture, present difficulties when used in parallel systems. The following section will aim to cover parallel designs that are based on Von Neumann.

2.1.3 Multithreading Computation and Multi Core systems

In a nutshell a thread is a sequence of instructions that can be scheduled into hardware. Threads are often described as their current execution point also known as the Program Counter (or just PC). From the perspective of the Von Neumann machine, a thread corresponds to the sequence of instructions in memory that conform the executing program as well as the current program counter (i.e. the current instruction) inside the Control Unit.

In a computer, the Operating System (OS) allows multiple programs to concurrently use the same hardware. In order to achieve this, the OS uses multiple processes, each containing software implemented threads. A process can be seen as an independent virtual Von Neumann machine, therefore each process has a CPU (i.e. a software thread), and a memory that is private. The OS uses virtual memory [28][29][30] to allow for each process to completely own memory, further encapsulating the idea of a virtual Von Neumann machine. Additionally, I/O is solely managed by the OS and its drivers, allowing each process to communicate with the hardware resources independently without requiring much coordination with other processes. The OS implements an scheduler that grants processes access to the CPU (and other hardware resources) at different times. When a process is scheduled, the software thread of the process is "connected" to the hardware thread of the CPU processor, and thus progressing the execution of that program. Figure 2.9 depicts this organization, and also shows an execution trace of the concurrent use of the CPU by all the the processes: P1, P2, and P3. Notice the similarities between the process structure and the Von Neumann machine.

Multithreading is the ability to have multiple threads running at the same time. From the perspective of the OS, software multithreading allows a process to create and manage multiple threads. From the perspective of the CPU, hardware multithreading allows the hardware to have multiple threads sharing the same memory and resources. Hardware multithreading is achieved either by having multiple independent CPUs (i.e. multi-core systems) connected through some network, or by allowing a single core to have multiple control units (i.e. Simultaneous Multi Threading or SMT) while sharing the ALU and core other resources. Note that it is not necessary to have hardware multithreading to support software multithreading. However, it is necessary to have OS support for software multithreading to allow a process to request multiple threads. Moreover, it is possible to request more software threads than hardware threads. This is usually referred to as oversubscription of threads.

POSIX Threads (or just PThreads), the most common implementation of software threads, is a language independent standard that defines software threads. The standard determines the different components of a thread (e.g. Memory segments,

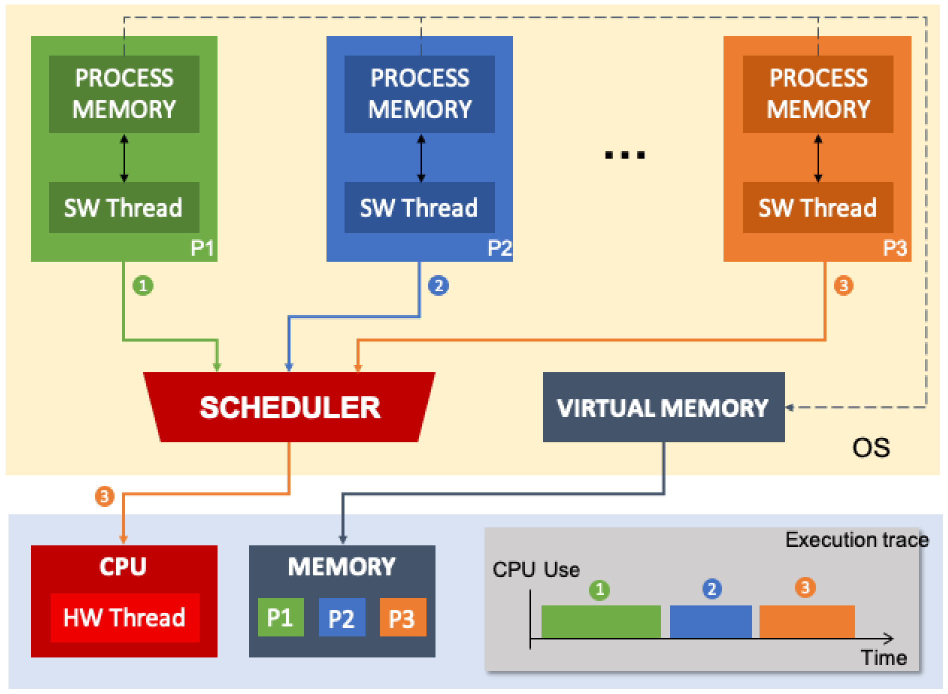


Figure 2.9: Role of the Operating System in concurrency: Each process is a virtual Von Neumann machine. The OS maps each process memory to physical memory through the use of virtual memory. The OS Scheduler concurrently maps the SW Threads to HW threads through context switching.

stack pointer, and program counter to mention some) as well as the API for creation and interaction with threads (e.g. `pthread_create()` and `pthread_exit()`). Software multithreading usually allows threads to have local private memory associate to each threads. Communication between local memory and shared process memory requires explicit memory copies.

Multithreading modifies the original Von Neumann architecture. Figure 2.10 depicts the most common multithreaded Von Neumann architecture. The most important aspect are the multiple CPUs located side by side around memory. Von Neumann based multithreaded systems consider memory to be a shared resource. Other systems will also include private local memory associated to each CPU. It is possible to think of the register file as an special case of private memory.

The flat memory address space used in sequential computers creates a monolithic memory view. However, real systems have a distributed memory system composed of

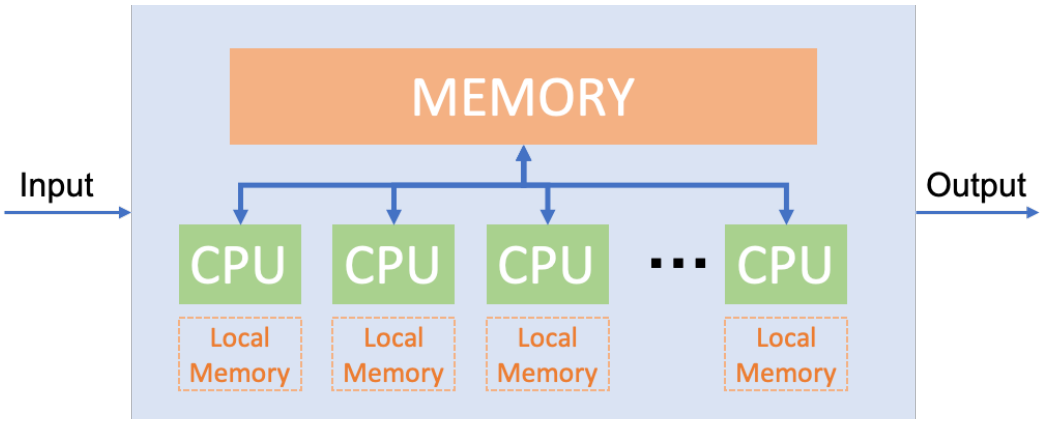


Figure 2.10: Diagram of a multithreaded Von Neumann machine.

cache, registers files and multiple external memory. Consequently, for a single memory reference (i.e. address) in the flat address space, it is possible to have data replication and multiple versions in different physical memory locations. In the case of cache, for example, data replication is essential for performance.

In multithreading systems based on Von Neumann, memory consistency models [31] provide rules that determines the behavior of multiple memory accesses (reads and writes) for a given memory reference (address). This is, for a single address that could map to multiple physical locations, the set of values that are admitted to co-exist as valid in the different locations is determined by the consistency model. For example, the value of a variable X may be stored in the register file of core 1 and the register file of core 2. A memory consistency model may determine this as valid behavior, or it may restrict that only a single element is valid.

Sequential consistency [17] is perhaps the most important consistency model of all. It determines that at any point in time there is only a single valid value per memory reference. Furthermore, it enforces the idea of a monolithic memory with a single port that can only process one transaction at a time. Thus, changes to memory occur in a sequential order. Cache coherency is a mechanism that maintains the sequential consistency when using caches. There are many cache coherency protocols [32][33].

There are two major issues with multithreading (von Neumann based) parallelism, as presented by Arvid et al in [34]:

- Memory references often require long idle times in the control flow of instructions, resulting in loss of overall performance.
- required synchronization between workers results in necessary context switching between threads, which are expensive and could potentially hurt performance considerably.

In addition to performance issues, multithreading parallelism requires the programmer to think of all possible side effects that may rise from parallel execution of threads, as described by Edward Lee in "The problems with threads" [19]. The lack of explicit synchronization between instructions executed in different CPUs means the user must explicitly define synchronization operations. In order to do this, a programmer must think of all possible interleave of instructions in the different threads, and how they change the memory state of the program simultaneously. The difficulty of this process increases as the number of threads increases. Furthermore, if system does not enforce sequential consistency, programming becomes a prohibitively complex process. Therefore, programmability in multithreading is a major issue.

2.1.4 Dataflow Computation

The dataflow model of computation was originally defined by Karp and Miller in 1966 [35]. The first version of a dataflow architecture was described by Jack Dennis in 1974 [36] and its corresponding programming language [37]. Following these principles, the 1970's and 1980's brought many computer architectures using these principles [38][39][40][41][42][43].

The dataflow computation model describes a program in terms of a directed graph. Each node of the graph is an operation (e.g. +, -, and *) also known as an actor. Operations are described in terms of its mathematical function as well as the number of inputs, the size of their inputs, the number of outputs and the size of their outputs. Inputs and outputs of a node are connections of the graph, and they represent data that travels from node to node as it gets transformed by the operations. Arcs are the equivalent to memory, and they carry tokens from one actor to the next one. Thanks to the lack of a single locus of control, and the explicit synchronization defined by the arcs of the graph, dataflow representations have the potential of fully exposing parallelism in programs.

In comparison with Von Neumann architectures. dataflow models of computation do not consider a program counter or a unique execution point in the program. Instead, actors (nodes in the graph) have a state that determines its execution. The state of the actor is only dependent on its inputs and firing rules. An actor is inactive or `disabled` if its input tokens are not available according to its firing rules. Once the firing conditions are met, the actor becomes `enabled` and it is ready for execution. Upon scheduling for execution, the actor is considered `fired`, all input tokens are consumed, and new output tokens are produced for the downstream actors.

Another important difference with the Von Neumann architecture is the lack of a globally accessible centralized memory. In the original dataflow model of computation, actors are only aware of its internal state and the input tokens. Therefore, memory locations for tokens are an essential part of the design of dataflow systems.

Dataflow Tokens Memory

In dataflow programs, connection between actors require a physical memory location to store tokens. Therefore, a mechanism to match new tokens to instructions (actors) is required. Some architectures encode token's memory in the instruction itself, while others have a separate memory with a token matching mechanism.

On the other hand, re-entrant dataflow code is a dataflow graph that may execute multiple times. Examples of re-entrant code are loops, and programs with multiple activations. Depending on how the computation is mapped into the physical hardware, special care needs to be placed in order to avoid deadlocks, or overwrite already existing tokens. These mechanisms may include changes in the token description.

The first dataflow architectural model proposed by Dennis uses a static approach. Each arc is a static memory location assigned to each connection between two actors and which could only store a single token. Therefore, firing rules of the actors are modified. The output arc in a static dataflow model needs to be empty at the moment of firing, so an output token can be generated. One solution to this problem is using FIFO queues to represent arcs. As long as there is enough space in the queue, and the graph is well-behaved, it is possible to solve some of the aforementioned issues.

Later dataflow architectures used a more dynamic approach. An arc is allowed to have multiple activations, as long as each activation has a different identifier, or is stored in a different memory location (executing environment). Coloring tokens is a common approach where a color is assigned to each activation of the graph. Firing rules are modified so tokens of the same color must be available in the inputs of the actors in order for it to be fired. An special actor is specified to create new colors before a new activation starts. In comparison to FIFO queues, color tokens allow exploiting more parallelism in the application, as the execution does not have to satisfy a given order between activations.

Well formed dataflow graphs

Determinism, determinacy and repeatibility are important aspects of program execution [44]. Although not every program needs to be determinate, parallel computing and parallel programming often brings new challenges to program repeatibility and correctness. In Von Neumann systems, the shared state of memory and the lack of an explicit synchronization of workers easily allows for unexpected behavior in program executions. On the other hand, dataflow computation has an explicit synchronization scheme. However, not every graph forming a dataflow program yields to well behaved dataflow programs.

A well behaved dataflow graph guarantees that in every execution results are always the same. Although evaluation of the nodes in a dataflow graph may be in different order, when considering the graph as a whole and the computation it performs,

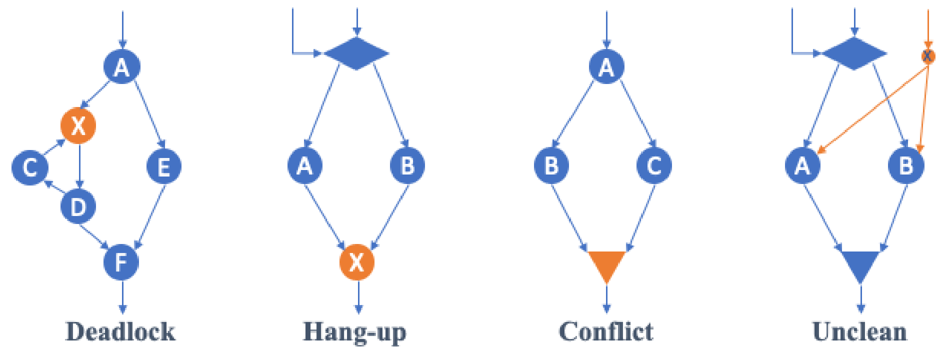


Figure 2.11: Example of non well-behaved graphs ("sick" graphs). Issues highlighted in orange.

well behaved dataflow graphs always yields to the same outcome. A well behaved dataflow graph is determinate [45][46][47][48].

Creation of dataflow graphs also depends on the definition of the firing rules and properties of the actors. Some models have opted to use non-deterministic switch and merge operands that, under re-entry operations, may result in non-determinate graphs. Furthermore, the structure of the graph is also important to create well-behave dataflow schemes. Some examples of "sick" dataflow graphs are presented in figure 2.11. The major issue is highlighted in orange. `Deadlock` and `hang-up` happens when actors will never resolve its dependencies, as is the case for the actors **X**. A `conflict` occurs in undeterministic merge causes a conflict between tokens of actors **B** and **C**. Finally, `Unclean` graphs will have dangling tokens in arcs. In the example, depending on the evaluation of the switch, either A or B will have a dangling token after execution.

2.1.5 Hybrid Von Neumann/Dataflow architecture

Dataflow and Von Neumann are two distinctive models of computation. They can be seen as two extremes of a continuous spectrum of possible models of computation. On one end, Von Neumann is driven by Control Flow operations and is inherently sequential. On the other end dataflow relies on the explicit synchronization in each operation and it is heavily parallel. In between the two there exists a whole variety of hybrid models that combine properties of both worlds.

Several surveys have been created in Dataflow/Von Neumann hybrids [49] [50][51][52]. There are multiple approaches to combine dataflow and Von Neumann. I describe three major approaches that are the most critical to this work.

2.1.5.1 Instruction Level Parallelism

To this day, Von Neumann based sequential systems have dominated the market. The use of Instruction Level Parallelism techniques, together with Moore's law [53] and Dennard's scaling [54], are major reasons for a successful and long reign of sequential architectures.

Instruction level parallelism (ILP) allows for a sequential program to implicitly exploit parallelism, without user intervention or required modification to the code. Neither the Von Neumann abstraction of the system, nor the semantics of the code is altered.

ILP relies on dataflow properties of program execution. Like in dataflow architectures, ILP uses data dependencies between instructions to allow for overlapping of independent instructions amid hardware availability. In Von Neumann data dependencies are maintained through memory references. When two instructions use the same memory in any of its operands, they are dependent. For architectures that rely on registers, dependencies can be easily discovered by following the register names and external memory references.

There are three type of data dependencies, but only one that is unavoidable. For a pair of instructions I_1 and I_2, where I_1 is evaluated before I_2 in the program stream, there is a data dependency if both instructions use the same memory reference. Output dependencies (i.e. Write after write or just WAW) occur when both I_1 and I_2 write results to the same memory location (e.g. register). Anti-dependencies (i.e. Write after read or just WAR) occur when I_1 reads from a memory location (e.g. register) that then I_2 writes to. Finally, true-dependencies (i.e. Read after write or just RAW) occur when I_1 writes to a memory location (e.g. register) that then I_2 reads from. In this case, instruction I_1 is producing a value for I_2, therefore, it is not possible to remove this dependency. In addition to data dependencies, control dependencies arise for two instructions I_1 and I_2 require the same hardware to perform its operation, regardless of the operands. For example Two multiply instructions in the presence of a single multiply in the ALU.

To achieve ILP, the hardware system access a window of instructions around the program counter at runtime. Specialized circuitry is used to discover dependencies within those instructions. Important care needs to be placed when communicating with memory. Regardless of the order in which instructions are executed in the hardware, outside communication must remain in the original order in which the program is described.

There are several techniques, but here I focus on three: Superscalar architectures, register renaiming and out of order execution. These techniques are not mutually exclusive, and are often used together.

Superscalar

Superscalar architectures focus on eliminating control dependencies, while being aware of data dependencies but not resolving them. The basic idea behind superscalar architectures is to increase the number of arithmetic logic units within the core [4][55] to allow for multiple instructions to be issued with the same operational unit. The order in which instructions are fetched and issued is respected, if a dependency cannot be satisfied, the pipeline is stalled.

```
MUL R1, R1, R1
MUL R2, R2, R2
MUL R3, R3, R3
MUL R3, R1, R2
MUL R4, R3, R1
```

Listing 2.1: Example code for Superscalar ILP

As an example, Listing 2.1 shows an instruction stream that can exploit ILP in superscalar architectures. Multiplication operations of lines 1, 2 and 3 are not data independent to each other, but the are control dependent by using the same ALU operation **MUL**. A superscalar architecture allows all three instructions to be issued for execution at the same time in different ALU units. On the other hand, instruction 4 is dependent on instructions 1, 2 and 3. Therefore, a superscalar system will stall execution until the first three instructions are committed. Likewise, instruction 5 will need to wait for instruction 4.

Register Renaming

Register renaming aims to suppress anti- and output dependencies. Register renaming is often used with memory locations in the register file, therefore its name. In a nutshell, register renaming uses a second register file hidden to the programmer. When an anti- or output dependencies are encountered, the register renaming mechanism swaps the conflicting operand's register name for a free register in the hidden file. The mechanism stores the mapping between the original register name and the assigned register name, allowing future true dependencies to be assigned the right register name.

```
1  MUL R1, R2, R2
2  ADD R2, R3, R3
3  SUB R1, R4, R4
4  MUL R5, R1, R2
5  DIV R6, R7, R8
```

Listing 2.2: Example code for Register Renaming ILP

Code in Listing 2.2 shows an example code that can exploit register renaming. Instruction in line 2 has an anti-dependency with line 1. Likewise, line 3 has an output dependency with line 1. Under register renaming, all three instructions 1-3 are able to execute at the same time. Register $R2$ of line 2 will be rename $R'2$, and register $R1$ of line 3 will be renamed $R'1$. When the instruction in line 4 is issued, the register renaming mechanism will replace the references $R1$ and $R2$ for $R'1$ and $R'2$ respectively, resulting in the equivalent line `MUL` $R5$, $R'1$, $R'2$. Register renaming will stall the execution on line 4 until instructions 2 and 3 have committed their results. Therefore, register renaming will not allow execution of instruction in line 5 until 4 has finished, despite having no dependencies whatsoever with any of the other instructions.

Runtime register renaming is equivalent to eliminating anti and output dependencies during code generation. There are many different implementations of register renaming. For example Tag-indexed register file [56], reorder buffer, and reservation stations [57] to mention some.

Out of order

Out of order (OoO) execution aims to avoid stalls in the execution pipeline while exploiting instruction parallelism. Out of order execution heavily relies on register renaming and superscalar architectures. However, contrary to the other two methods, instructions are not necessarily scheduled in order. Consequently, the execution pipeline is not stalled if an instruction is not ready for execution. Two important examples are the scoreboard algorithm, as used in the CDC6600 [55], and Tomasulo's algorithm [57].

OoO engines use a window of instructions around the program counter for which the execution follows a dataflow model as previously described. In contrast with the original dataflow model of execution, OoO only enforces dataflow execution within the window of instructions, while the overall program still relies on the Von Neumann model abstraction. Within the window, instructions are scheduled as their dependencies are satisfied, while techniques similar to register renaming and superscalar increase the available parallelism. OoO relies on two actions to retain the Von Neumann abstraction at the program level while using dataflow: first, data dependencies between instructions are discovered and respected throughout the execution. And second, any instruction that communicates with the outer world of the CPU (including memory) is committed in the order they appear on the original instruction stream.

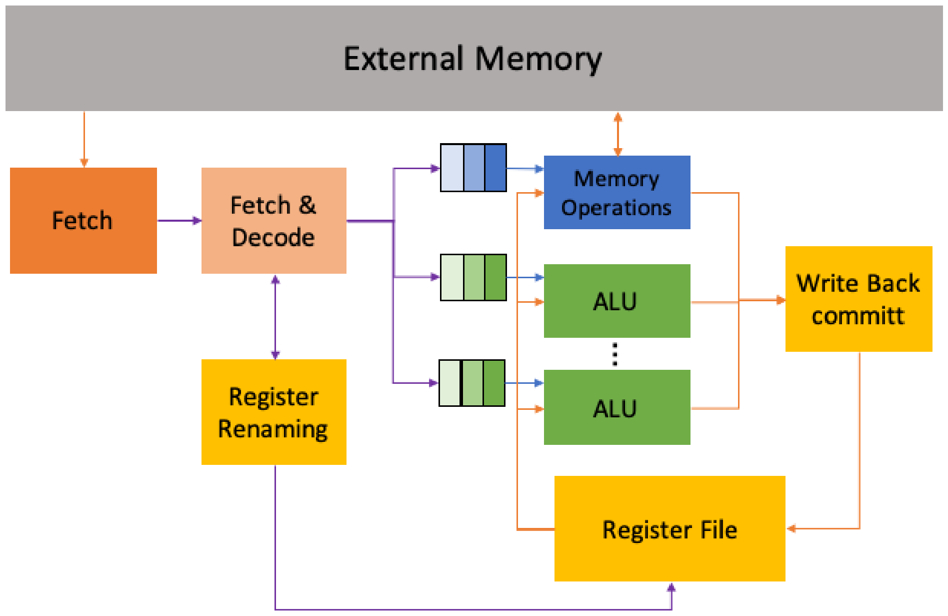


Figure 2.12: Pipeline of an Out of Order architecture with Superscalar and Register renaming.

Referring to the example in Listing 2.2, an OoO execution engine will allow instructions 1, 2, 3 and 5 to execute in parallel, while instruction 4 waits for 1, 2, and 3. A system that uses all three ILP techniques can be seen in figure 2.12.

2.1.5.2 Tasking

ILP aims to maintain the Von Neumann abstraction and use dataflow techniques throughout the execution of the instruction stream inside the core. On the other hand, tasking brings a reversed approach. Let us define tasking as the representation of programs with an structure similar to dataflow model of computation. That is, programs are represented as directed graphs. Nodes of the programs are operations and arcs between programs are data and control dependencies. Contrary to nodes of a dataflow architecture, tasking nodes (i.e. tasks) represent coarser grain sets of operations with multiple low level instructions per task. Tasking may be implemented in software (e.g. [58][59][60][61]) or hardware (e.g. [62][63][64][65]).

There are different possible ways to mix the Von Neumann and dataflow models in the form of tasks. More often than not, tasking models use a Von Neumann execution model inside of the task description. This is, tasks are a set of instructions that execute sequentially. Under this model, tasks are scheduled by some runtime across the different cores of a multiprocessors. Instructions inside the task are issued in sequence by a program counter. A dependency mechanism in the runtime handles the communication across tasks. A memory model describes how memory is seen inside and across tasks. In order to map the dataflow execution to a Von Neumann machine, the tokens that communicate tasks need to be stored in the shared memory system of the multithreaded systems 2.1.3. A memory model must be described that determines what memory locations may be access inside tasks and across tasks, as well as who has access to those elements, and how shared accesses are coordinated.

Other approaches use Von Neumann description of groups of instructions that are executed in a dataflow style. Therefore, a dataflow graph represented inside a procedure. A program uses procedures in an stream of instructions that is evaluated sequentially. The execution of each procedure occurs by means of dataflow models.

Regardless of the mapping between Von Nuumann and dataflow, memory models are a critical aspect of the design of hybrid approaches. Some models stick to the shared memory Von Neumann model inside and across tasks. The user is in charge of guaranteeing that under parallel execution of tasks, memory references that are accessed do not overlap across tasks with no dependencies. For example, if A, and B are two tasks that do not have any dependencies and may be executed in parallel. User must guarantee that all memory references inside A and B do not overlap. Otherwise, data races may occur. Some programming models distinguish data that is private to the task, from data that is shared among tasks. Tasks are often described as functions (or procedures). Therefore, private data for a task is often the stack of the executing function. A challenge that comes from the use of procedures is that the Von Neumann based programming models commonly distinguish between stack and heap memory across the different threads. Hybrid models using shared memory often struggle with memory management, requiring to delimit context for which tasks exist [58]. An example of this issue is when a new task B is created out of task A, and task B references private memory from task A that is stored in the stack of the executing thread. When task A cease execution before task B, locations referenced by task B will likely not be valid anymore.

Other models use specialized hardware or software implemented schemes to provide synchronization mechanisms between producers and consumers at the memory access level. Futures [66] and split-phase transactions [67][68][69] allows for reads and writes of memory to occur in any order, and it uses the write operation to signal the consumer. Under these models, memory is consider to be write once only, or a full empty bit mechanism is needed to allow re-writing to memory.

Yet another technique groups together multiple tasks into execution frames (e.g. threaded procedures [38], or object memory in object oriented programming [63]). The grouping of instructions allows for exploitation of locality. To organize communication between context, it is required to guarantee allocation of resources. In the context of parallel execution of context, a common approach is to use asynchronous threaded procedures called from within other frameworks. They rely on a cactus-like stack memory structure [70]. When a new frame is spawned, a new stack is created. Since multiple frames can be spawned in parallel, multiple stacks will reside at the same logical level, forming the cactus structure. To synchronize between frames, split-phase like transactions can be used as futures to continue computation in the spawning frame.

2.1.6 Instruction Set Architectures and Program Execution Models

One additional key element that was introduced with the evolution of general purpose computation is the Instruction Set Architecture (ISA). The ISA not only describes the operations supported by a machine's computational unit, but also provides a well defined description and overview of the operation of a machine that implements such ISA. These operations are defined in terms of behavior and language interface regardless of how they are implemented. An ISA also describes other elements of the machine such as registers available to the user and their purpose (i.e. general or specific purpose). From the hardware perspective, the Instruction Set Architecture (ISA) formalizes the software-hardware interface. The ISA is a contract between the user of the hardware, and the hardware implementer.

The introduction of ISAs was imperative for the evolution of computer systems. It resulted in two disjointed (and yet related) evolution processes. First, on the hardware side, architecture developers were able to continuously improve their designs without requiring a whole re-structuring of the executing programs. An example of this is Intel's Tick Tock model [71]. Second, on the software side, a rapid evolution of software infrastructure and algorithms. Software has continuously grown to define multiple programming models, programming languages and programming paradigms that built upon each other into different layers of complexity. Therefore, software has isolated the programmer (i.e. user) from the actual low level execution model and ISA of the hardware. Growth of software technology has also been accompanied by the evolution of compiler technology, libraries and runtime systems that has allowed users to focus on modular application development.

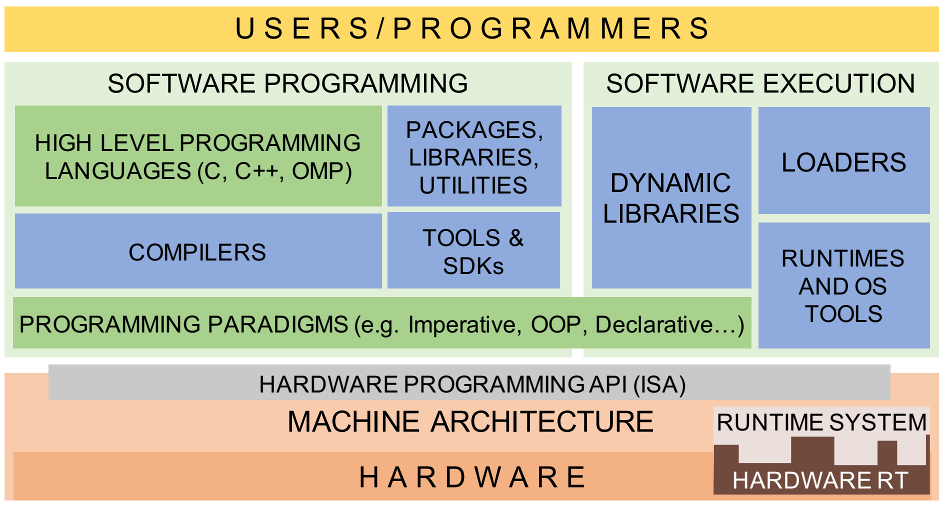


Figure 2.13: Conceptual view of computer system infrastructure

To illustrate this evolution figure 2.13 shows a conceptual view of what current computer systems are in terms of the models and the different interacting parts. At the top level we have the users and programmers, and at the bottom we have the executing hardware. Hardware usually implements a runtime system, usually transparent to the user, that glues together the conceptual view of the hardware programming API and ISA with the physical parts of the machine. In the middle of users and hardware, we have the software infrastructure. An advanced and complex interaction of moving parts between the programming realm and the execution realm, glued together by many levels of abstractions and models. On the programming side, high level programming languages allows the user to define programs based on programming paradigms exposed by the semantic of the programming language. Furthermore, a collection of libraries, packages, tools and compilers allows translating these programs into lower level abstractions that can be interpreted by the hardware and given to system for execution. Software execution is aided by software implemented runtime systems, operating system features, loaders and dynamic libraries that in many cases allows the implementation of the different programming paradigms or language execution models. A brief example is the C runtime library that is linked with the user program, or the OS APIs that allows the connection between virtual memory and physical memory. Computer system infrastructure is so entangled, that innovation is limited by the inertia of all the moving parts. As Hennessy and Patterson pointed out in their Turing award lecture [1].

> *As seen repeatedly, although the marketplace is an imperfect judge of technological issues, given the close ties between architecture and commercial computers, it eventually determines the success of architecture innovations that often require significant engineering investment.*

On the other hand, Parallel computing does not count with a widely accepted contract akin the ISA. There exists a large number of execution and programming models that expose parallelism in both hardware and software. However, each model presents a different system abstraction. Even within the same programming model, its evolutionary process across versions leads to a modification of the abstract machine in ways that are not trivial for the programmer. A clear example is the evolution of OpenMP throughout the inclusion of different parallel models across versions (e.g. fork-join, tasking and offloading models).

The Program Execution Model (PXM) is an abstraction that shares similarities with the concept of Instruction Set Architectures. Unlike ISAs that only account for a single core processor, Program Execution Models accounts for the computation system as a whole. A program execution model is defined around a machine abstraction that describes the organization and interconnection of components. The PXM defines the behavior of the system under program execution, as well as the required language to allow a user to program the machine. A PXM is a formal definition of the Application (or even better Architecture) Programming Interface API of the computer system.

A PXM often describes three elements: a) an activity model which defines the work to be performed (similar to ALU operations and ISA arithmetic instructions); b) the memory model which specifies the addressing model of memory as well as the results of memory operations and the corresponding memory state transition; and c) a synchronization model that deals with interactions between activities. If system architects and programmers are able to reach an agreement on a program execution model, it would be possible to achieve a seamless evolution of parallel machines. One goal of this work is to be able to provide a well defined program execution model for parallel machines.

2.2 The Codelet Model of Computation

A more thorough description of the Codelet Model can be found in [72][59]. Following is a summary of some of the most important and influential elements of the Codelet model for this work.

The Codelet Model is a Program Execution Model that uses a Von Neumann/-dataflow hybrid approach to describe parallel programs. A Program is represented by several direct acyclic graphs (DAG). Each node of the DAG is a task (called Codelet) and the edges are data or control dependencies between tasks. Codelet DAGs are grouped together into asynchronous procedures called Threaded Procedures (or just TP for short). Starting from an initial TP, Codelets can invoke new procedures creating an storage structure similar to a cactus stack memory model [70]. Due to the asynchronous nature of TPs, continuation Codelets are used to manage signals across TPs similar to a split-phase transaction.

The Codelet Abstract Machine (CAM) describes the organization of the system as viewed from the perspective of the Codelet program. Starting at the bottom, there are two different type of units: Compute Units (CU) and Synchronization (or scheduling) units (SU). Multiple computational units (e.g. cores) and one synchronization unit are grouped to form a clusters. Multiple interconnected clusters form a node, and multiple interconnected nodes for computation system. Memory can be hierarchical and distributed as well. It is mapped to the system as follows. Each core may have its own local memory to serve as stack for execution of Codelets. A cluster may contain memory that is shared by multiple cores. Additionally, shared memory at the node level may allow multiple clusters to share information. Synchronization Units (SU) are specialized programmable hardware designed to make resource management and scheduling decisions within the cluster. Compute Units (CU) are general purpose hardware cores in charge of executing the code that describe a Codelet. Inside a cluster, there may be different kinds of CUs, allowing for heterogeneous computation. A Threaded Procedure may only map to a single Cluster. Figure 2.14 shows a depiction of the Codelet Abstract Machine.

Firing rules determines the order of execution of a Codelet program, as well as the need for synchronization. As is the case for dataflow inspired models, Codelets have three different states that depend on data dependencies, control dependencies and availability of hardware resources. A Codelet is in `waiting` state if one or more dependencies have not been satisfied. Once all dependencies are satisfied, the Codelet moves to `ready` state and it is waiting to be executed. As CUs become available, Codelets are scheduled, moving to `fire` state. An important distinction with pure dataflow models is that Codelets are also event-driven. Some Codelets have dependencies on certain events in the system. Events allow for long latency operations to occur without stalling compute resources. Events may come from an external source (I/O events) or when operations within the system cannot guarantee a bounded latency (e.g. far memory

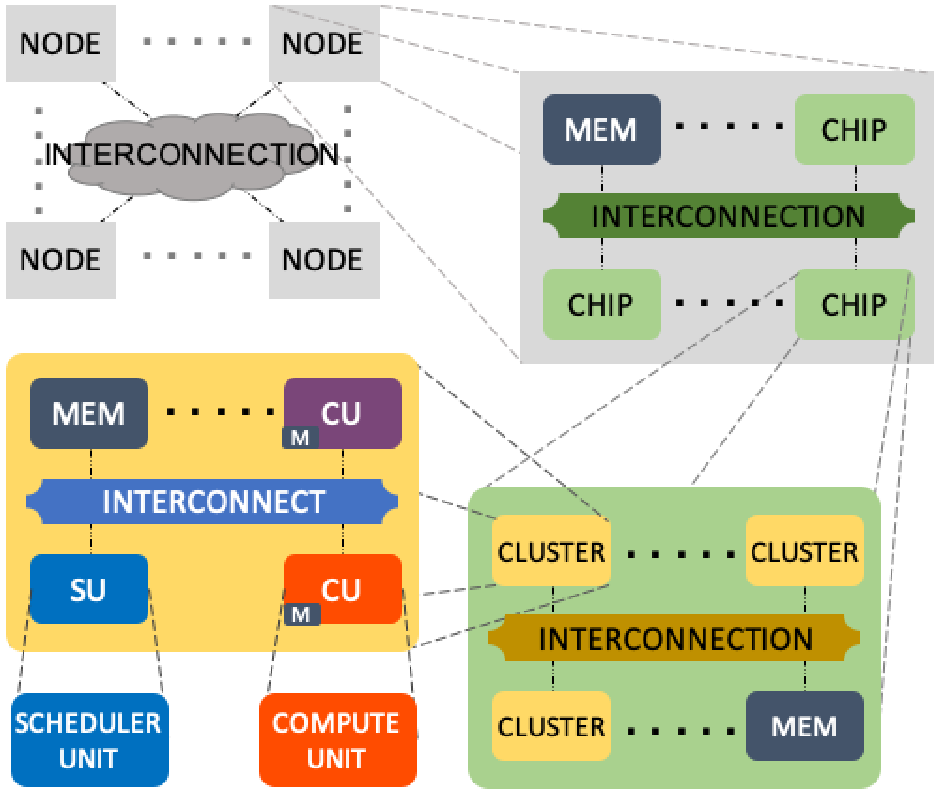


Figure 2.14: Codelet Abstract Machine Depiction. Hierarchical organization of the system and its memory.

accesses and results from other threaded procedures).

Codelets are also atomically scheduled and non-preemptive. A Codelet is executed until completion and cannot be stopped. Non-preemption allows saving resources on context switching. To enable non-preemptive execution, any resource that a Codelet needs at runtime must be guaranteed to be at a bounded access latency from the scheduling CU at the moment of Codelet execution.

The Codelet Model aims to solve some of the issues of dataflow based architectures [73]. In particular, traditional dataflow systems have a flat memory abstraction that does not necessarily map to the hierarchical structures of memories in current systems. Second, local scheduling policies of dataflow operations may result in non-optimal scheduling of tasks. As shown in the Codelet Abstract Machine, the Codelet Model considers a memory hierarchy on the system. Furthermore, limitations imposed to the mapping of Threaded procedures into a single cluster allows to exploit locality. Likewise, the multiple compute units that conform the cluster are in close distance reducing the communication cost. At the system level, the distribution of Synchronization units enables the creation of different local and global scheduling policies. As a result, implementations of the Codelet Model [74] has already used work stealing and other scheduling techniques.

However, there are some problems that have not been explored in the original Codelet Model. First, while scheduling is distributed, it lacks of a hierarchical view of the system. What happens beyond the cluster?. At the system, node and cluster levels, there are no scheduling artifacts that allows for better workload managing and better resource utilization. Second, Threaded Procedures are forced to remain at the cluster level. The size of threaded procedures is limited to the size of a cluster's memory, unless upper levels of memory are utilized. However, coordination and locality of memory beyond the cluster is not trivial. The question remains: how to appropriately manage Codelet size, Threaded Procedure size, and the mapping of larger memory into the system's memory hierarchy?. Third, Imperative programming models based on sequential execution of code are easy to understand for a programmer. However, description and debugging of graphs is still a difficult task. The original Codelet model depends on creation of well-behaved dataflow graphs [44], thus complicating system programmability. This work aims to solve some of these issues by extending the Codelet Model of Program Execution.

Chapter 3
OBJECTIVES AND PROBLEM FORMULATION

This thesis starts by defining the Hierarchical Turing Machine and the Hierarchical Von Neumann architecture [75]. Based on these two models, this work then defines the Sequential Codelet program execution model and the SuperCodelet computer system architecture. The Sequential Codelet Model is hierarchically defined. At each level, programs are written with sequential semantics, similar to assembly code. Unlike current ISA assembly, the supported operations of this program are tasks, called Codelets, and can be user defined. At a certain level of the machine a program describes Codelets as supported operations of that level. Codelets of one level are defined in terms of the Codelets and instructions of the level below. Parallelism is achieved through techniques similar to the optimizations used by Instruction Level Parallelism (e.g. Superscalar architectures, Out-of-Order execution engines, and Register renaming techniques). Memory is hierarchically organized and structure to map to the memory hierarchy in real systems.

3.1 Objectives

The overall objective of this thesis is to define a program execution model that comprehends the computation system as a whole, allowing for programs to execute in parallel while considering programmability, performance and portability for future improvements of the system. In particular the objectives of this thesis are the following:

- Definition of the mathematical model that works as ground base for computation across large systems. It shall be called the Hierarchical Turing Machine.
- Definition of an abstract machine that is based on the mathematical model and which describes the organization of a system and the interaction of the different components. It shall be called the Hierarchical Von Neumann Model.

- Definition of a program execution model of a system that is organized in the structure defined by the abstract machine. It shall be called the Sequential Codelet Model.
- Definition of a possible architecture that implements the Program Execution Model. It shall be called the SuperCodelet Architecture.
- Create an evaluation methodology that allows to assess the benefits of the Sequential Codelet Model through the implementation of the Super Codelet Machine by the use of current commodity parallel/distributed hardware and its extensions.

In a nutshell the major contribution of this work is the creation of the Sequential Codelet Model that, by means of a hierarchical structure of the system, it describes programs sequentially while allowing lowering parallel/distributed execution of work.

3.2 Problem Formulation

This thesis aims to give an answer to the important question:

> A general purpose parallel/distributed computation model should have desirable properties of performance efficiency, portability and programmability. **Is it possible to define such models, and their corresponding abstract machines, on commodity hardware and extensions?**

In order to give answer to this important question, we ought to first find a solution to the following problems:

- **Problem 1:** Following the path of Turing and Von Neumann, What are the models that enable the aforementioned properties across a parallel/distributed and hierarchical system?. This question is developed in Chapter 4 Section 4.2 through the Hierarchical Turing Machine and the Hierarchical Von Neumann model.
- **Problem 2:** Based on the foundation of problem 1, what is the program execution model and abstract machine that enable the aforementioned properties across the parallel/distributed and heterogeneous system?. Chapter 4 Section 4.4 describes the Sequential Codelet Model.
- **Problem 3:** How to describe an implementable architecture that uses the Sequential Codelet Model?. I call this the Super Codelet Architecture, and it is described in Chapter 5.

- **Problem 5:** What is an appropriate strategy to evaluate the Sequential Codelet Model and determine system design parameters in commodity hardware and extensions? Chapter 6 shows the design and implementation of the SCMUlate system. Chapter 7 show early evaluation results of the SCMUlate system.

Chapter 4
THE SEQUENTIAL CODELET MODEL

The Sequential Codelet Model (SCM) is a Program Execution Model that heavily borrows from the success of sequential computation. The SCM is not yet-another-attempt to parallelize already existing sequential code. Instead, the SCM defines a program execution model (as described in section 2.1.6) that uses sequential semantics, and it leaves parallelization as an execution optimization. On the other hand, current auto-parallelization techniques of sequential code aims to bridge a Von Neumann based sequential programming model and a parallel model of computation. Attempts to auto-parallelize sequential code into multithreaded versions often fail to be general purpose enough due to the difficulty of understanding side effects of the interaction between threads, memory access patterns and problem dimensions in the context of a complex system architecture. The Sequential Codelet Model starts by providing a well defined abstract machine and execution model that are different to the Von Neumann machine. That is, the SCM is based on the Hierarchical Turing Machine and the Hierarchical Von Neumann architecture, and it is still required to write programs for the proposed abstraction. Parallelism occurs transparently to the programmer by means of techniques based on principles of dataflow similar to those used in Instruction Level Parallelism. Contrary to a single core ISA, the SCM uses a hierarchical abstraction that covers the whole computer system. Contrary to other out of order execution of tasks, the hierarchical organization of memory allows to break away from the original Von Neumann abstraction, allowing for a more appropriate mapping of this model and current parallel/distributed and heterogeneous systems.

Perhaps the most valuable aspects of the Sequential Codelet Model are: 1) it presents a hierarchical organization of Turing Complete machines that enables computation at any level of the machine. 2) Instructions are defined sequentially in an imperative style programming model. The SCM machine may use ILP-inspired techniques to achieve program parallelization, therefore removing the burden of the programmer to think in parallel. 3) the memory organization of the abstract machine presented in the sequential Codelet Model recognizes the hierarchical nature of memory organization (e.g. registers, L1 cache, L2 cache, L_N cache, DRAM, storage devices, network file systems, cloud storage, and so on). And 4) the Sequential Codelet Model allows for a weaker memory model to be implemented system-wide, yet within each level it relies on sequential consistency models at each level [17]. The Sequential Codelet Model considers the system as a whole, allowing to span beyond the single core into multi-core, multi-sockets, multi-node and cloud computing systems.

The Sequential Codelet Model, is an extension of the Codelet Model defined by Sutterlein et al. and the CAPSL research group in [59] and [72]. We envision a realization of the SCM model as a hardware/software co-design strategy that includes machine organization, machine programming interface, compilation technology, and definition of appropriate higher level programming abstractions and programming languages. This chapter starts with a motivation example that inspires the Sequential Codelet Model. Following by the definitions of the Hierarchical Turing Machine and the Hierarchical Von Neumann Model. Based on these models, the Sequential Codelet Model is introduced. Finally, a hardware realization of the sequential Codelet Model, called the Super Codelet model, is presented.

4.1 Motivation example

Operations supported by a computer are described through the Instruction Set Architecture and its extensions. Among the different supported instructions on a particular ISA, arithmetic and logic instructions provide the building blocks to represent more complex mathematical functions in a program. However, for a given architecture that targets a particular domain the set of supported arithmetic operations is part of the architect's tradeoffs on how to use the die area. While general purpose CPUs usually support basic arithmetic operations (e.g. ADD, SUB, DIV, and MUL), other architectures targeting domain specific applications (e.g. DSP-like systems), are equipped with extra instructions that compute more advanced math operations (e.g. trigonometric operations, advanced FMAs, or FFT operations).

ALU operations often follow these characteristics:

- The latency of that given operation is bound to a number of cycles. This value is known and does not change considerably. A good operation design would guarantee that this latency is as low as possible for the given technology, and within the same order of other operations.
- The latency to access to the operands stored in registers is bounded to a certain value and negligible. The operation's execution time would be driven by the execution time of the ALU, and not by the access time to registers.
- The execution of the operation has no side effects other than the output operands, and depends only on its input operands.

Property 1 allows a compiler to optimize the scheduling of instructions, maintaining the pipeline busy. Instructions within an ALU are often in the same order of magnitude in terms of number of cycles, allowing the system to use other parts of the pipeline, while instructions finish. Furthermore, property 2 allows property 1 to hold. When an operation is scheduled in the ALU, all the operands that are needed for its execution are already available in the register file. The ALU does not waste time waiting for data to arrive from memory or waiting for other resources. Finally property 3 ensures a deterministic behavior of the instruction execution, and it does not limit the creation of more complex programs to be described by means of these ALU operations.

Let us imagine that, as long as we guarantee the aforementioned properties, we could write our own ALU operations at runtime by using a special assembly code that changes the behavior of the ALU. We also assume that we have a register file large enough to maintain our operands. For this example, we wrote an special instruction in the ALU that counts the number of matches of a 4-byte integer value within an array of 100 4-byte integers. The 100 element array is stored in a single register of size 400 bytes (i.e. `SourceReg`, while the result is stored in a single register of size 4 bytes `ResultReg`. The following signature of our new ALU operation is as follows:

COUNT SourceReg, ResultReg, #value

By using this new instruction, it is possible to write the sequential program in listing 4.1 that counts the number of occurrences of the value 55 in an array (named Arr) of size 300 stored at location $0x50$ in a 4-byte addressable memory. R_L are large registers storing the array, and R_S are small registers storing the result.

```
LD R_L1, 0x50
COUNT R_L1, R_S1, 55

LD R_L1, 0xB4
COUNT R_L1, R_S2, 55

LD R_L1, 0x104
COUNT R_L1, R_S3, 55

;;Add the three results together
ADD R_S1, R_S1, R_S2 #R_s1 += R_s2
ADD R_S1, R_S1, R_S3 #R_s1 += R_s3
```

Listing 4.1: Motivation Example

If the processor that executes this instruction uses multiple ILP optimizations (e.g. register renaming, Out of Order execution and superscalar), the execution of these instructions are likely to occur in parallel thanks to the lack of dependencies across the different instructions Therefore, as long as the operations associated to a particular application are available in the ALU to be executed, and we respect the aforementioned observations, it should be possible to parallelize a sequential code by using ILP optimizations that have already been studied for several decades. Furthermore, the ability to modify the ALU allows a programmer to have functional instructions that help solving more complex problems. If we had the chance to modify our ALU at runtime, it could be possible to optimize certain operations critical to our application.

However, die are is a limited resource, and register files usually have a limited size, how could we create a system that allows us to modify the ALU while also providing us with larger registers. The important takeaway of this example is that, if we had the possibility of modifying the ALU operations, we might be able to create sequential programs that execute in parallel thanks to Out of Order execution techniques available in current architectures. The Sequential Codelet Model aims to define an architecture that allows the ALU to be programmed into more complex operations, while Instruction Level Parallelism is used to obtain a parallel execution of code.

4.2 Hierarchical Turing Machine

The Hierarchical Turing Machine (HTM) is a mathematical model that works as a foundation of the Sequential Codelet Model. The Hierarchical Turing Machine was first introduced in [75]. Due to the generality of the HTM model, it could lead to different organizations of a machine. Ones possible organization is described in this thesis. We focus our attention into the Sequential Codelet Model which resembles a hierarchical Von Neumann Architecture. The Hierarchical Turing Machine is a variation of a multi-tape Turing machine, therefore it does not extend the computational numbers set defined by the Turing Machine.

Alan Turing first described the Universal Turing Machine in his paper in 1936 [2]. The Turing Machine has three major components: 1) An infinite storage tape divided into sections containing symbols; 2) a reading head that slides along the tape accessing a single section at a time; and 3) a state machine (i.e. `m-configuration`) that corresponds to the program executed by the machine. Each state performs certain

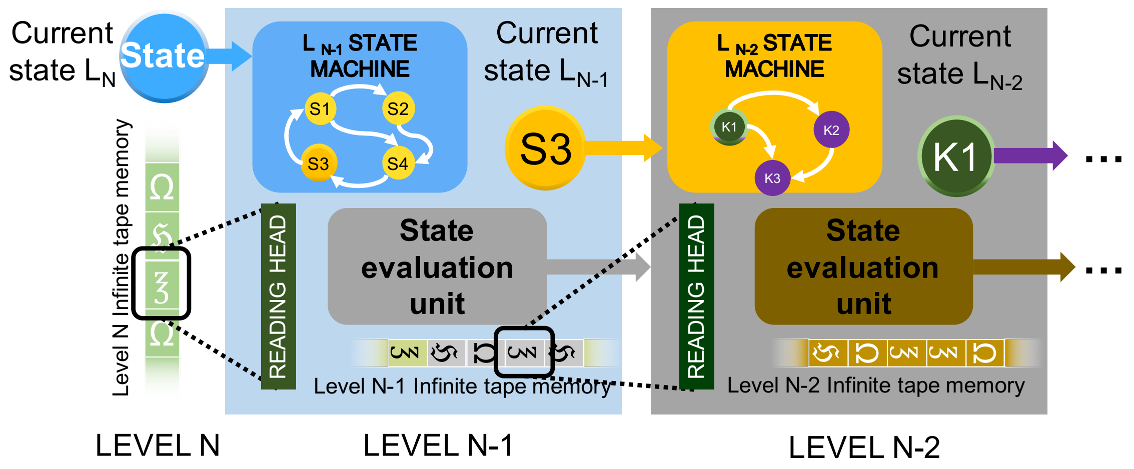


Figure 4.1: A 3 levels Hierarchical Turing Machine. An state of level N is expressed as an state machine of level N-1 and evaluated by level N-1. Each level has an infinite memory tape used for the computation performed at that level. Symbols depicted in tapes are just to account for any possible symbol that is known to the machine.

operations before transitioning to the next state. Transition is defined based on the symbol read in the current section of the tape. A state's operation is determined by a combination of three possible actions: Move right or left across the tape, print into the current tape section, or erase the current tape section. Based on these operations, Turing created the Universal Turing Machine, which is capable of executing any Turing machine based on a description language encoded in the tape.

In the Turing Machine, the head contains the state machine and it has no storage of its own. The operations are only guided by the values that are read from the current tape square, and the current state. However, there are many operations that require to keep temporary results or marks during computation. The distinction between the `F-Squares` and `E-Squares` (See section 2.1) represents an example of what forms temporary values or marks, and what forms the final number computed by the machine. Therefore, in order to maintain partial results the machine uses part of the tape to work as a scratchpad for computation. Since the tape is infinite, this is not necessarily a problem. The location of the scratchpad memory and the result memory in the tape can be changed according to different Turing machines.

Another aspect anticipated by Alan Turing was the definition of parametric instruction tables that unwrap into regular instruction tables. He called them "skeleton tables". Turing showed that it is possible to describe a state in terms of other states and basic operations, as long as at some point the recursion of definition resolved to a single finite state machine.

Using the aforementioned observations, and following a similar strategy to that used by Turing to describe the Universal Turing Machine, it is possible to modify the structure of the original Turing Machine into a hierarchical organization without loosing generality. The Hierarchical Turing Machine (HTM) can be seen as multiple Turing machines stacked one on top of the other. Similarly to the original Turing Machine, each level contains three basic elements: A tape, a head and a state machine. However, states in the Turing machine can also be described as superstates, similar to the "skeleton tables", which are defined by a state machine. For a Superstate of level N, the Turing Machine of level $N-1$ is used. The final state machine is a hierarchical state machine [76] that maps to the hierarchical organization of the HTM.

The structure of the HTM is depicted in Figure 4.1. Level N is represented partially, while the two levels of the hierarchy ($N-1$ and $N-2$) are completely depicted. As can be seen, level N's current state is evaluated by using level $N-1$. That is, the state machine of level N is a combination of states and superstates. States are evaluated in level N, and superstates are mapped to level $N-1$. The evaluation unit (i.e. the level below $N-1$) is therefore used to evaluate some of the states of Level N. Likewise, $N-1$'s superstates are evaluated by use of $N-2$ as an evaluation unit. This hierarchical pattern occurs multiple times, until a bottom level N_0 is reached. N_0 is a level for which no state is expressed in terms of the lower state N_0-1, therefore, it corresponds to the lower level and it reassembles a standard Turing Machine.

Not every state that is described for an inner levels must be a superstate evaluated by the level below. That is, for a state machine expressed for an inner level N (different to N_0), there may be states that do not require a lower level $N-1$ to be evaluated. Instead, these states are expressed in terms of the regular operations of the original Turing Machine: Move right/left, print a symbol or erase a symbol. On the other hand, states of a level N state machine that do map to level $N-1$ are described similarly to “skeleton tables”, as originally defined by Alan Turing. “Skeleton tables” allow for grouping of state descriptions assimilating a routine or procedure. It is possible to create a recursive “Skeleton table” or use other tables to describe an “skeleton table”. Nevertheless, it is not allowed to have a description of a state that results into an infinite recursion. The resulting state machine after flattening the Hierarchical State Machine must be a valid finite state machine.

4.3 Hierarchical Von Neumann Architecture

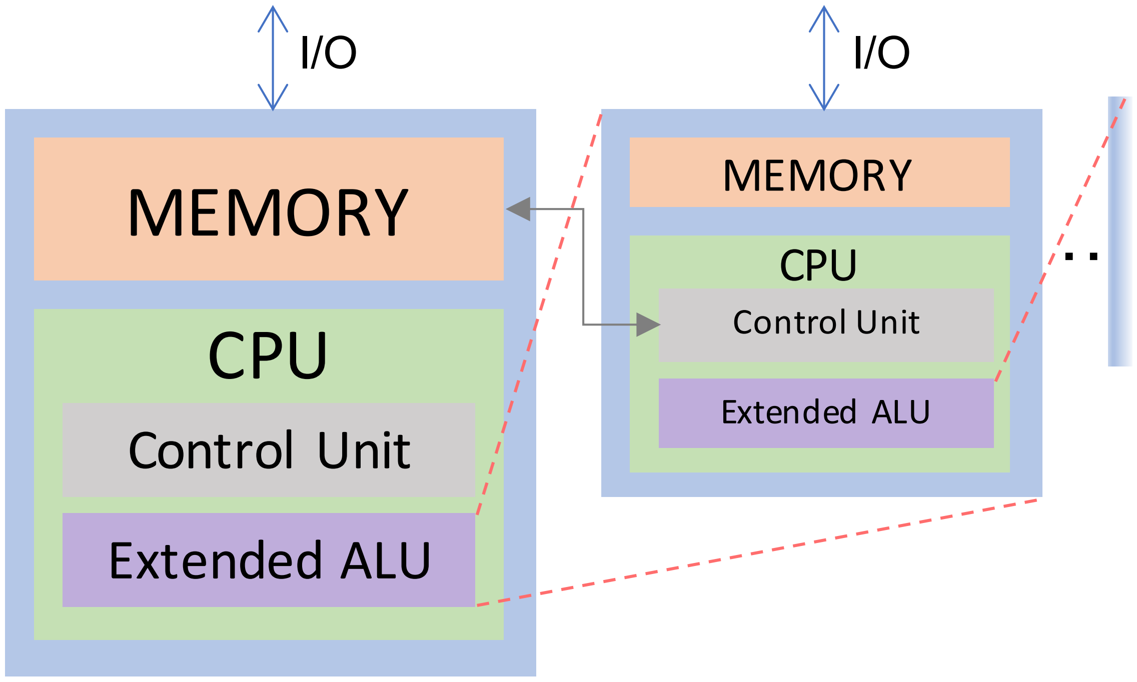


Figure 4.2: Structure of the Hierarchical Von Neumann Architecture. A Sequential Codelet Model Abstract machine

The original Von Neumann system organization contains four basic elements: a central processing unit, an operations unit, a memory unit, and the I/O. Inside the CPU there are at least two elements: The control unit and the Arithmetic Logic Unit.

Furthermore, memory contains instructions and data. Based on the Hierarchical Turing Machine, we define the Hierarchical Von Neumann Model. The difference between the original Von Neumann Model and the hierarchical Von Neumann Model relies in the arithmetic logic unit. In the former model, the ALU contains a set of predefined arithmetic operations, while in the latter the ALU is extended to contain a whole programmable Von Neumann machine (i.e. a Turing complete ALU). Hence the similarities between the hierarchical Turing Machine and the Hierachical Von Neumann Model.

Figure 4.2 shows a simplified diagram of the Hierarchical Von Neumann model structure. The machine reassembles multiple conventional Von Neumann machines stack on top of each other. However, similarly to the HTM, a level N of the machine may use the level below $N-1$ to execute some of the instructions. For a given level N, all the supported instructions are seen as single units of operation, atomically scheduled and with a well defined behavior. Some instructions of this level N are expressed as a sequentially described program of level below $N-1$. Likewise, from the perspective of level $N-1$, all its instructions are seen as atomically scheduled units of operations, some of which may be expressed in terms of level $N-2$. So forth until a level $N=0$ is reached. This last level corresponds to a regular Von Neumann machine where all operations map to a regular ALU that is not programmable.

In the Hierarchical Von Neumann model, the ALU is extended by including a whole programmable Von Neumann system. However, this does not mean that it loses its ability to perform basic arithmetic operations. Similarly to the states of the different levels of the HTM (see section 4.2), some operations of a the inner levels $N! = 0$ the hierarchical Von Neuman model will still be executed by an ALU-like system at that particular level. The benefits are twofold, it allows to resolve simple arithmetic operations needed by the control flow operations, but it also enables compute capabilities at higher levels of the hierarchy (e.g. on-memory compute)

Additionally, the hierarchical organization of this abstract machine also results into a memory hierarchy. Memory for a level N corresponds to a subset of the memory in level $N+1$. The evaluation of an instruction of level $N+1$ described as instructions of level N, uses part of the memory of level $N+1$. This area of memory is used to store the input and output operands as well as any internal state needed for the execution of that instruction (e.g. stack). Memory of level N is then a subset of memory of the whole level $N+1$. As a result of this hierarchical organization, memory may be mapped to the hierarchical memory of real hardware systems. By means of well defined operations at each level, it is possible to use different memory consistency models across the levels, as we will see in the definition of the SuperCodelet architecture in Chapter 5.

On the other hand, important attention needs to be placed in the I/O. While the diagram of Figure 4.2 does not restrict the location of I/O at a given level, system and operation designers have to be careful as it will have a critical effect on the deterministic behavior of the system, as well as the instructions of a given level. A safe mode of operation is that I/O can only impact the level at which the computation is initiated, or that I/O corresponds to the stimuli that initiates the computation. Ideas outside of this safe mode of operation need to be explored further, and are outside of the current scope of this work. For now we assume I/O is only present at the upper most level of the hierarchy, that is, the level where computation starts.

Finally, the Hierarchical Von Neumann model is not the only system that can be envisioned with the Hierarchical Turing Machine. Furthermore, a hierarchical Von Neuman machine can use a combination of other architectural models that could be used in this hierarchical fashion. This is particularly useful for the support of heterogeneous computation, where application-specific architectures become the ALU of a given level. An example of this would be to use FPGAs or Neuromorphic chips [10][9][77] as the ALU of a level of the hierarchy. A Codelet at this level would be expressed in terms of the RTL instructions that maps into the given architecture.

4.4 The Sequential Codelet Model

Based on the Hierarchical Turing Machine, it is possible to create new organizations of the computer system. In the following section we present the Sequential Codelet Program Execution Model and its corresponding abstract machine.

4.4.1 The SCM Abstract Machine

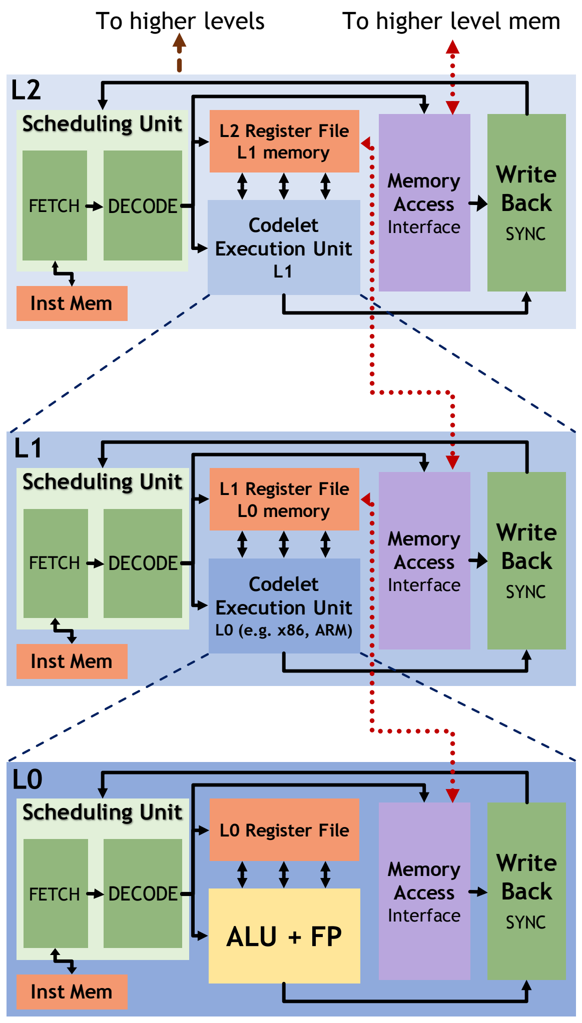


Figure 4.3: A 3 level abstract machine of the Sequential Codelet Model that implements the Hierarchical Von Neuman Model.

Figure 4.3 shows a more complex abstract machine that implements the Hierarchical Von Neumann architecture. Each level is organized as a 5 stages pipeline: Fetch, Decode, Execute, Memory and Write Back. At the bottom, level 0 (marked L0) corresponds to any single core architecture that is commonly found today (e.g. a core implementing any of the commodity architectures: RISC-V, ARM, x86, or POWER PC). The L0 ISA is extended to include a `commit` instruction that works as a return signal for the level above. Other than the `commit` instruction, a program expressed for L0 has no difference to a program written in assembly level on current architectures.

As can be seen in the Figure, on top of L0 there is level 1 (marked L1 in the figure). The execution stage (Extended ALU) of the L1 pipeline corresponds to the whole L0. Likewise, the execution state of L2 corresponds to L1. This organization continues to other levels above. At each level N, the execution state corresponds to the level below $N-1$. Each level may also be accompanied by an ALU, not depicted here for the sake of simplicity.

In addition to the 5 stages, a level has an instruction memory where operations (Codelets) are stored, and a register file that contains memory for operands. The Memory Access unit issues memory operations to the register file of the level above (see doted arrow line in Figure 4.3). Therefore, the label of the L1 register file is also L0 memory, and L2 register file is L1 memory. In the original abstraction, a level can only see the memory in the level above.

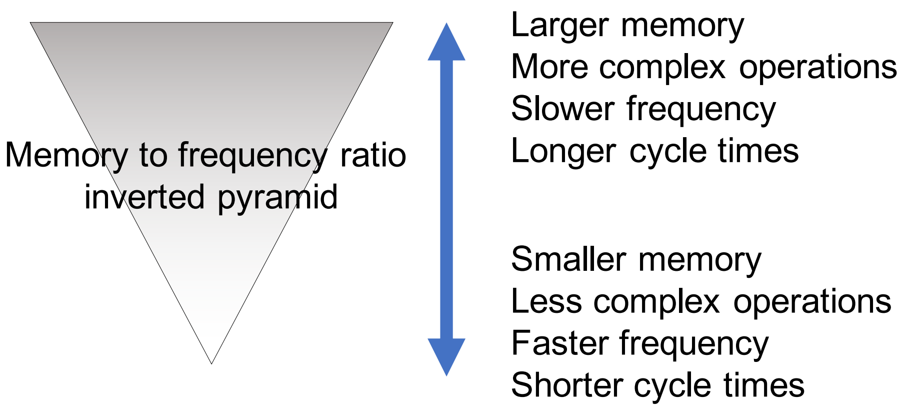


Figure 4.4: The Memory size to Frequency ratio of the SCM

An important observation is that the higher the level of the hierarchy, the larger the memory capacity is with respect to the execution units of the level below. Furthermore, as we move up the hierarchy, the complexity of the operation is expected to increases and the operation's latency in the execution stage also increases. This can be summarized as the inverted pyramid of Figure 4.4, and it will be important when deciding the size of the operations, and where to execute them. Additionally, the approach described here uses memory that is inclusive. Values that are needed in the lower levels from the upper levels have to be copied in the levels in between. In a more advanced architecture it should be possible to describe memory and commit operations that jump across levels bypassing memory accesses across levels, and breaking this inclusion rule. A possible trade-off for energy for complex memory hierarchies was explored by Livingston et. al. [78] and it could be extended to this approach, however this is outside of the scope of this work.

4.4.2 The SCM Program Execution Model

An execution model contains three properties. Tasking model, memory model and synchronization model.

4.4.2.1 Tasking model

In Figure **??** we observe three different levels. L0 has a commodity ISA, as previously described. However, the operations supported by the L1 instructions (i.e. the L1 ISA) are not limited to simple arithmetic operations. Instead, L1 has more complex, user implemented functions as part of its instruction set. These instructions are referred to as Codelets. An instruction Codelet is atomically scheduled in L1, and it is represented by a stream of instructions evaluated in L0 and which implements the L1 function. This stream of instructions finishes with a `commit` instruction that informs L1 when the Codelet has finish execution.

In the Sequential Codelet Model a task or instruction is represented as a Codelet. The term Codelet is taken from the definition proposed in the original Codelet Model of program execution [59] as explain in Section 2.2. In the Sequential Codelet Model, Codelet is the name given to an instruction that is used in a level N (where $N! = 0$) and executes at a level $N-1$. Codelets share similar properties to ALU operations available in current ISAs. Codelets are atomically scheduled and executed non-preemptively. Execution of Codelets is side-effect free, and there is no internal state stored across multiple execution of the same Codelet, hence, a Codelet output only depends on its inputs.

A Codelet maps to a particular level of the hierarchy, and its operation is described as a sequence of instructions (Codelets or native ISA instructions) of the level below, guided by a set of control flow operations (i.e. a similar structure to assembly code). Inputs and Outputs of the Codelet are assigned to memory locations (i.e. registers) of that particular level in the hierarchy (as described in section 4.3). A Codelet program assigned to the level N uses sequential semantics. Codelets are issued in the order they appear in the program. A Codelet corresponds, for example to the `COUNT` instruction described in the motivation example at the beginning of this chapter.

Referring again to Figure 4.3, L1 Codelet operands are stored in the L1 register file. This L1 register file corresponds to L0 Memory. Hence, Arithmetic and control operations of the L1 Codelet are described in terms of L0's ISA. Memory instruction inside the Codelet will access data from L1's registers. This limits the latency of memory operations to the distance between L0 and L1, allowing to have a bounded execution time for Codelet memory accesses. The total size of the register file as well as the size of each register is expected to be higher in L1 than in L0. Therefore, for a single register in L1, multiple load and store operations may be needed to access its data. It is still possible to have registers of different sizes in L1, including registers of common sizes (e.g. 8, 16, 32, and 64 bits registers).

We refer to the L_1 Codelet `COUNT R1, R2, val` described in the motivation example. This Codelet sets register `R1` to the number of times the value `val` is present in `R2`. R1 is an integer an it may be stored in a 32 bit register, while R2 corresponds to an array of 100 integers and it will use a 3200 bit register (each of size 32 bits). The value `val` is an immediate value. Both R1 and R2 are registers in L1. The implementation of `COUNT` is assembly code in L0. Since L0 registers are smaller than 3200 bits, there will be multiple load operations for each of the integer values of R2. Each load operation will be a 32 bit register in L0, requiring 100 loads in order to compute the whole COUNT operation.

The relationship between L2 and L1 is similar to the one of L1 and L0. Instruction Codelets that are used in L2, are described as stream of instructions and Codelets in L1. Additionally, L2's register file memory space has a larger capacity than L1's register file. Therefore, registers available in L2's register can be larger than those of L1, resulting in multiple L1's memory accesses to cover a whole single register of L2's register file. Operands of the instruction Codelet in L2 use registers in L2, memory instructions of L1 map directly to L2. Consequently, the hierarchical instruction spans over multiple levels, even beyond L2.

To summarize the execution of a Codelet in the L2 offloads to L1, and Codelets of the L1 instruction set offloads to L0. Load and store operations of L0 will fall into L1's register file, and those of L1 fall into L2's register file. Since the registers of L1 can be larger, multiple load and store operations of L0 would be required to access the whole operands of the operation in L1. Consequently, L1's register file acts as L0's memory. It is expected that L1's register file is physically close to L0 (e.g. L1's memory could be located where level 2 or 3 of cache is located in current architectures), hence memory latency of memory operations in L0 should be bounded to this short distance between L1's memory and L0's registers.

4.4.2.2 Synchronization Model

Codelets are synchronize by means of data dependencies expressed as registers used in its operands. A Codelet is described with a number of operands. Each operand can be a constant value (immediate value), or a register value. Additionally, each operand has a direction that could be either `READ`, `WRITE`, or `READWRITE`.

The implementation of the Sequential Codelet Model must guarantee that real dependencies (i.e. `READ` after `WRITE`) are always satisfied during the execution of the Codelet program. Synchronization is determined by the relative position of two Codelets in the program stream. Two Codelets A and B have a real dependency if 1) Codelet A comes before Codelet B in the instruction stream, 2) Codelet A has an register operand `R` with a direction `WRITE` or `READWRITE`, and 3) Codelet B has an operand that uses the same register `R` with a direction `READ` or `READWRITE`. Synchronization occurs when the order between A and B is guaranteed in a given implementation of the Sequential Program Execution Model.

4.4.2.3 Memory Model

Memory is hierarchically organized. However, level N can only interact with level $N+1$ through Load/Store operations, and Level $N-1$ by means of operations that affect the register file. This restriction allows for a memory consistency model to be define for the interaction within a single level with respect to the level above. It is not necessary to maintain consistency across other parts of the system, reducing the need for a monolithic view of memory.

For two Codelets within the same level, memory operations must be seen in the order the Codelets appear in the instruction stream. This applies to both the register file of the given level, and the memory of the level above. In general memory instructions must be ”committed” in order in which the program is described. There is a lot of freedom in how these are guaranteed, including the use of Gao Sarkar Location Consistency Model [79]. For now, we focus on enforcing sequential consistency across two levels of the machine. In the case of the register file, the order of memory operations is guaranteed by the synchronization model. While for memory operations, the Memory stage in the pipeline must guarantee that instructions are committed to the level above as they appear in the instruction stream.

On the other hand, support for data structures is an important aspect of programming models. For the aforementioned imperative representation of code, memory locations in registers are seen as memory locations with a given size. However, the hierarchical structure of memory used in the SCM allows for arguments of a Codelet to be interpreted in the form of any data structure. For example, the execution of a level N Codelet $COD\_EXAMPLE$ with parameters $COD\_EXAMPLEP1, P2, P3$ uses three registers, one for each parameter. From the perspective of N registers represent memory locations in the register file, with no given interpretation. However, at level $N-1$, each operand P of the $COD\_EXAMPLE$ is seen a regular memory in the sense of the Von Neumann abstraction. Therefore, Level for $N-1$ the interpretation of the values inside of the registers P may be organized in the form of different data structures. Further research on higher level programming models would be needed to design the appropriate strategies to span data structures at the whole system level, as well as any operation to transform data structures.

4.4.3 Programming model

As a result of the structure of the Sequential Codelet Model, code is independently written for each level. There exist a basic instruction set that applies to all the levels. This basic instruction set contains some simple arithmetic operations (e.g. ADD and SUB) and some control flow operations (e.g. Jump, and branching). These instructions are used to determine the control flow of the Codelet program at the given level. A Codelet program ultimately represents a dataflow graph where arcs are represented in terms of data dependencies between instructions. The basic instruction set allows the definition of such dataflow graph at runtime.

To write a program for a given level N, code is written by use of a set of Codelets and instructions in the basic instruction set. Codelets of level N are defined in terms of the level below $N-1$. The description of a Codelet in level $N-1$ also uses the basic instruction set architecture, and Codelets of level $N-2$. At level 0, a program will described in terms of currently available ISAs (e.g. RISC-V, ARM or x86).

Thanks to the influence of sequential execution models, we expect to be able to extend already existing sequential programming languages to adapt to the hierarchical behavior of the system. It is important to mention that it is possible to fix the signature of a Codelet at a given level, while work on different possible implementations for that same instruction, with different objectives (e.g. performance, energy or other optimization goals). Such property improves software composability [7][80], allowing also for the creation of libraries that describe Codelets of a given level.

4.5 Codelet Level Parallelism: Parallelism and performance

This section describes different possible ways to achieve parallelism through the SCM. We call these approaches Codelet Level Parallelism (CLP) due to their similarities with Instruction Level Parallelism (ILP). In this section I first discuss the use of ILP optimizations used at each of the levels of the hierarchical abstract machine to achieve CLP. Second, we describe how it is possible to use other architectures to build a heterogeneous system to improve performance of the execution of code. Finally, we briefly touch on how compiler optimizations for code generation could be used.

4.5.1 Codelet Level Parallelism

Choosing the five stages pipeline to represent the Hierarchical Von Neumann Model in figure 4.3 was not an arbitrary decision. This machine model allow us to easily introduce the use of ILP optimizations at each level of the hierarchy to achieve implicit parallel execution of code.

Starting with superscalar architectures, it is possible to organize a multi-core system as a collection of execution units at any level. By means of dataflow techniques studied in chapter 2, it is possible to execute multiple Codelets at once, as long as data dependencies are respected. At each level, there exists a register file that is shared among the different execution units, while the latency of communication between these execution unit remains bounded to a common value (e.g. memory access time of L3 cache memory for a multicore architecture). Finding the right number of execution units at each level requires extensive research, but it assimilates to the tradeoffs of single core architectures design. Several specifications could influence this decision: register file size, memory latency, energy and silicon area are just a few examples.

Extending the number of execution units in each level is what allows the Sequential Codelet Model to account for parallelism. Let us imagine a Level L1 composed of multiple RISC-V cores into a single chip, followed by a level L2 with multiple sockets, a level L3 with multiple nodes, and a level L4 with multiple clusters.

In addition to SuperScalar techniques, it is possible to use register renaming to eliminate false dependencies that could potentially limit the parallelism found at a certain level. By using a hidden register file, it should be possible to increase parallelism. As an example, code in listing 4.1 could benefit from register renaming techniques. By renaming registers R_{L1}, multiple versions of the $COUNT$ Codelet may execute in parallel. The direction of a given operand in the Codelet (i.e. `READ`, `WRITE` or `READWRITE`) is an important property in the Codelet to enable register renaming.

Finally, the most important technique to be used is an Out of Order (OoO) execution, that allows out of order issuing of instructions within a window at each level. This ILP technique uses dataflow concepts, by detecting data dependencies at runtime for a window of instructions. Dataflow is a major component of the Original Codelet Model, therefore, using OoO execution engines would allow implementing this property. Through the use of reservation tables, and by adapting commonly known algorithms for OoO execution (e.g. Scoreboarding [55] and Tomasulo's [57] algorithm), it should be possible to achieve higher levels of parallelism during execution of programs at each of the levels.

Likewise, further parallelism could be obtained through the use of forwarding techniques of values between execution units at any level, as well as adaptations of speculative execution engines. Moreover, using streaming techniques such as the ones described by Raskar et. al. [81] could also benefit performance. In general, it is necessary to study the different techniques to be able to find the proper trade-offs between them, and understand their impact during execution of a program.

Finally, there is Codelet Level Parallelism that could bridge current fork-join models of computation to the Sequential Codelet Model. Current instruction set architectures feature SIMD extensions that allows to apply a single instruction to a register that holds multiple data values. Such extensions use special architectural features that increase the throughput of instructions per cycle the system can perform. A Single Codelet Multiple Data (SCMD) analogy is possible. A Codelet in SCMD mode is applied to multiple data elements at the same time.

A difference between SIMD and SCMD, is that SCMD could potentially map to multiple levels of the hierarchy. When extended to multiple levels, an SCMD execution model assimilates to hierarchical parallelism available in current GPGPUs. These systems feature a group of threads referred to as warp (usually 32 or 63 threads). Multiple warps compose a block, and multiple blocks compose a grid. An indexing scheme allows for each thread to have an independent identification value that can be used to offset data access in the execution. For an SCMD mode a similar mechanism needs to be put in place. Exploration of SCMD execution models will be left as future work of this thesis.

4.5.2 Heterogeneity and distributed systems in the Hierarchical Abstract Machine

Recent interest in heterogeneous systems have shown the importance of heavily parallel architectures that exploit different execution models (e.g. GPUs and SPMD execution). Additionally, there are many architectures that could be used for specific applications yielding to improved performance. For example, recent introduction of Tensor Core units in GPUs [82], as well as google TPUs [9] for AI related applications. These architectures could be part of the abstract machine as well. By using them as an execution unit of any of the levels, it could be possible to create specific Codelets that map to these architectures. For example applications could use GPU's Streaming Multiprocessors as an execution unit that takes over Codelets that are SPMD friendly. Equally, a neuromorphic execution unit could be used to execute AI related Codelets, or an FPGA implementation of a given Codelet could be offloaded to an re-configurable FPGA to improve performance of the application. Figure 4.5 shows a multilevel heterogeneous SCM abstract machine.

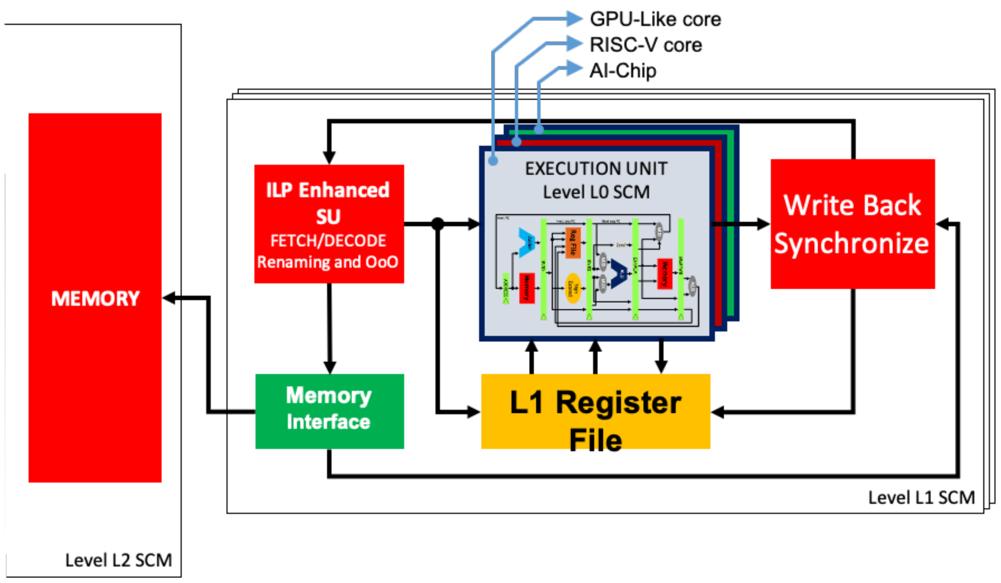


Figure 4.5: Heterogeneous Sequential Codelet Model abstract machine

Furthermore, the hierarchy could be extended through multiple levels. This is depicted in figure 4.6. While nodes in orange represent CPU cores with a behavior similar to the one described so far, green nodes represent the aforementioned heterogeneous architecture. It would be beneficial to have a highly heterogeneous system in a single chip. The hierarchy could be extended through multiple nodes of a cluster, or even connect multiple clusters together to for a single computation system.

In general, it is expected that the higher in the hierarchy a level is, the larger the available memory, but also the larger the memory latency. However, latency of operations in a higher level would also be larger. The different trade-offs need to be studied to be able to balance out these elements: memory size, memory latency and latency of operations. In the same way, bringing cores from the bottom level of the hierarchy to higher levels (i.e. in-memory processing) is also a feasible solution that could influence given trade-offs.

4.5.3 Compiler techniques

Perhaps another advantage of the Sequential Codelet Model is that there are already many compiler optimizations that could potentially improve the scheduling of

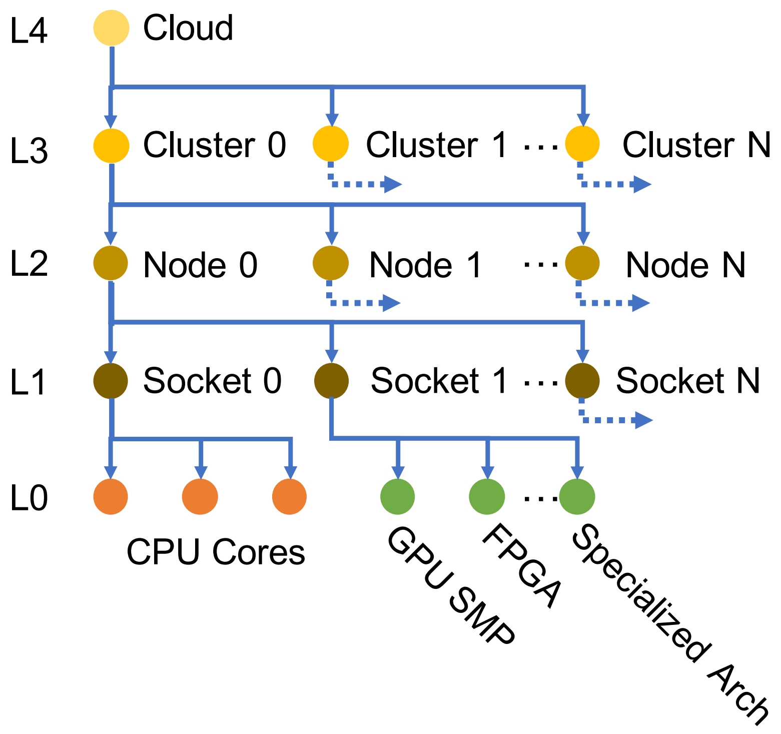


Figure 4.6: Extending the hierarchy beyond L2.

instructions at runtime for a particular level. A program written for any level could be independently analyzed for compiler optimizations that would treat Codelets as ISA operations to further exploit hardware optimization and increase ILP-based optimization opportunities. It might be that additional information about the Codelets would need to be provided or discovered. As an example, Codelets could be merged together given a particular context, similar to how compilers change independent multiplication and addition operations into a single fused multiply-add operation (FMA). While compiler techniques would have to be further studied, we recognize the potential benefits that applying compiler optimizations could bring to the performance of the executing code, for a given realization of this machine.

Chapter 5
THE SUPERCODELET ARCHITECTURE

Having laid down all the theoretical background in Chapter 4, I now proceed to present the SuperCodelet architecture. This architecture uses all the elements presented in the SCM, and it allows a construction of a parallel system. In terms of system organization, I use Tomasulo's original architecture [57] as basis, and I extend it for the upper levels of the hierarchical abstract machine of Figure 4.3. In particular I focus on level 1 (L1) of this architecture, and I use both CPU and GPU cores as the execution unit. My intention is to show how to achieve parallelism on heterogeneous systems using the SuperCodelet architecture, demonstrating that my intention is not to go against the progress made up until now in heterogeneous architectures, but instead build upon it.

The key element for parallel execution is to guarantee in-order fetch and commit in the presence of out of order execution. Between instructions that do not communicate to the outer world (all but memory instructions), it is important to maintain the order of true dependencies (i.e. Read after write), while anti and output dependencies can be eliminated through Tomasulo's algorithm and register renaming techniques. For instructions that communicate with the outer world, the order in which these instructions are performed must be maintained. Having a unique memory controller guarantees sequential consistency, while there may be some re-ordering opportunities on certain operations (e.g. read after read).

5.1 The Architecture organization

Figure 5.1 shows the diagram of the SuperCodelet architecture for a 3 level system organization. Level 2 is partially shown in light gray background surrounding

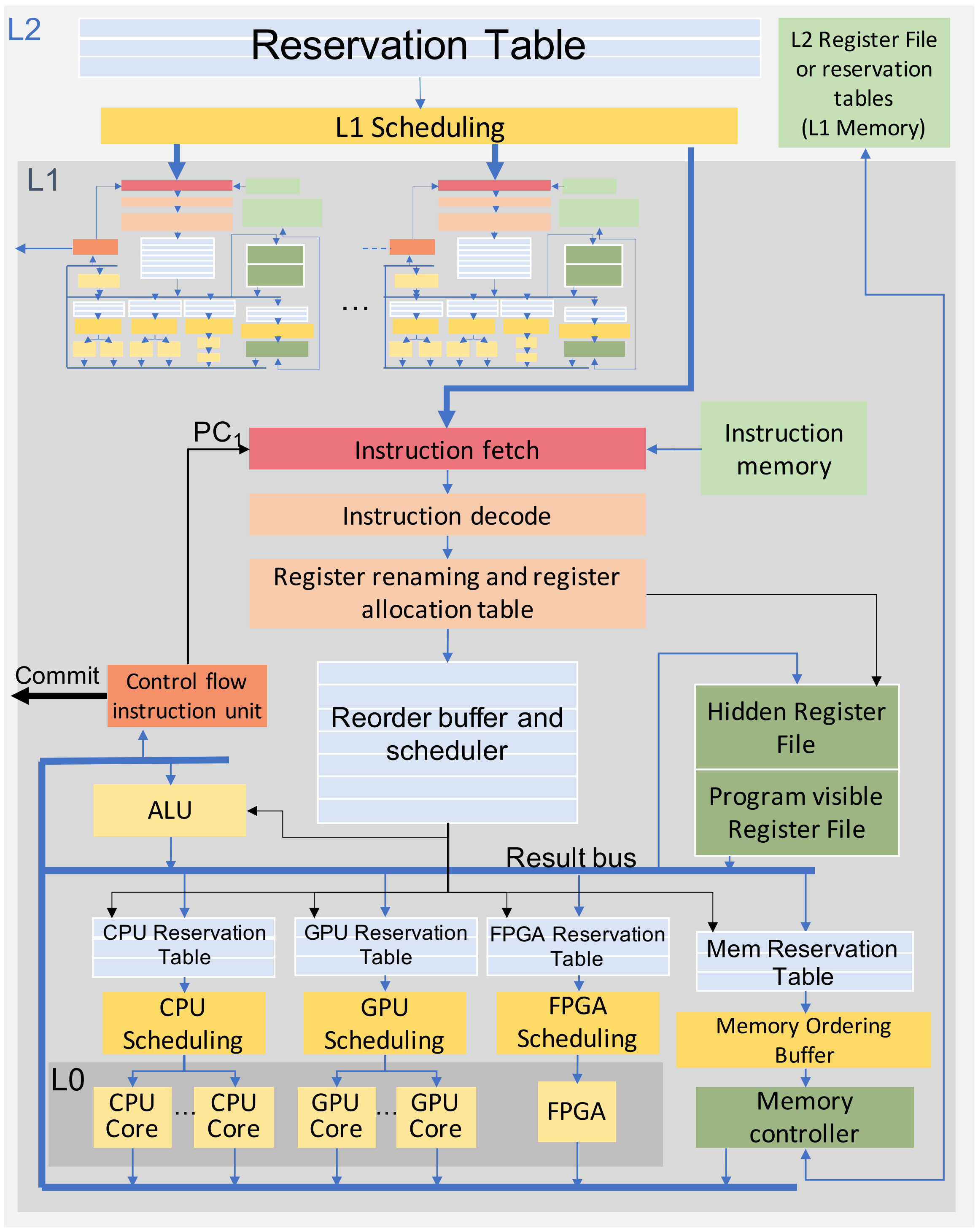


Figure 5.1: Diagram of the SuperCodelet architecture

the whole figure and label L2 on the top right. L2 only pictures the reservation table, L1 Scheduling logic and the L2 register file (i.e. L1 memory). Following, Level 0, in dark gray background near the bottom of the figure, is composed of several commodity architectures seen as black boxes. Finally Level L1, in mid-range gray, is shown completely, with three different L1 compute units displayed in the figure (2 of which are smaller on the top right of the L1 block). Let us focus our attention in Level 1.

5.1.1 Scheduling units

The Scheduling unit is in charge of fetching instructions and decoding them for the given computational unit of the level below. Our pipeline starts in the red block named "*Instruction fetch*" which takes the current program counter of level 1, and reads the next L1 instruction. This instruction is given to the "*instruction decode*" that determines the instruction's destination and its operands in the register file. Following, instructions are analyzed for potential "*register renaming*" opportunities, and these registers are allocated for the given instruction. As is the case in original Out of Order execution engines, some sort of look up tables are needed to maintain the information of which translation corresponds to what original register. This is the "*register allocation table*". Next in the pipeline there is a "*reorder buffer and scheduler.* The purpose of this stage is to seek hazard free execution opportunities within a window of instructions. As the instructions are ready to execute, they are assigned to different "*reservation tables.* An instruction may be ready even if a missing value has not been produced yet, but it will be produced by another core. Once an instruction is determined to be ready it is allocated into one of the "*reservation tables*", depending on which computation unit in L0 the Codelet maps to (i.e. *GPU*, *CPU* or *FPGA*).

5.1.2 Computation Units

There are four *reservation tables* in the diagram [1], one for each of the type of execution units available: CPU, GPU, FPGA and memory. For each reservation

[1] this is not necessarily a hard limit, other architectures can be envisioned

table there exists a scheduling logic that checks if the operation has all the required operands available to start execution. If so, one of the computational units for that particular reservation table is selected (i.e. GPU-like cores, CPU-like cores or Memory operations). In the diagram of Figure 5.1, I use four types of computation units: CPU-like cores that use commodity architectures (e.g. RISC-V, MIPS or ARM). GPU-like cores that are similar to those used inside the recent GPGPUs and which have architectures based on vector machines such as those used by the Cray-1 computer [12]. A regular ALU unit used for basic arithmetic operations that are still needed for loops, offsets and address calculations, including other control flow operations. Finally, an FPGA computational unit that could be in the form of bit-streams, allowing reconfiguration of the FPGA bit-streams to occur at runtime.

5.1.3 Memory Units

Memory operations require their own infrastructure. Communication with the upper level L2 needs to be in the program order in which they are defined. However, there is still some opportunity for re-ordering instructions. Under certain circumstances that do not affect the output of the result, two memory operations may be executed in an order different to the original program order. These circumstances are determined by the region of memory location that is being accessed by the memory instruction, and the operation to be performed (i.e. read or write). The memory ordering buffer can attempt to take care of these situations. Depending on the memory model implemented at the given level, the restrictions across memory operations may be strictly enforced or relaxed.

As we progress in the hierarchy, more complex memory operations may be required. While it is not possible to envision all possible memory access patters, it is still possible to create 1D, 2D and 3D memory instructions, similar to what other ISAs have done [83]. Furthermore, an application may decide to create a special Codelet that performs memory operations. An special interface for this Codelet requires the calculation of memory addresses that are access by this Codelet in order to calculate potential hazards in memory accesses across different instructions.

If the same diagram is used to describe L2 and beyond, memory operations will result in reads to the Reservation Table, and writes will be sent through the result bus and stored in the subscriptions of the given result.

5.1.4 Register File

There are two "*register files*". One that is accessible by the user, that corresponds to all the different general purpose registers the user can use in the description of the program. A second register file is used for register renaming opportunities. This register file is hidden from the user, and it is accessible only by the architectural runtime.

The size of the register file, as well as the size of the registers in the register file is still a subject of research. I have explored some options based on the current size of L3 caches in commodity architectures. This will be explored further in the evaluation section 7. What is known for sure is that some of the register must be of small sizes such as the ones used nowadays. These registers are needed to store loop variables, memory addresses and offsets. Otherwise, a lot of space would be wasted in large registers.

5.2 Programming model

In this section I will discuss the low level software interface designed for users to program the system. As previously mentioned, a program written for the SuperCodelet model maps to the hierarchical organization of the machine. I leverage the work previously presented in [75]. Therefore, the program is written in levels. For each level of the program, the user must define the collection of Codelets that can be executed at that particular level, but which will be used as part of the instruction streams in the level above. This is, if a Codelet is to be used in the code for L2, then that Codelet needs to be defined in level L1 using instructions of level L0.

There is a subset of operations that is called SuperCodelet ISA. These operations can be used in any level and will be specified in this section. Therefore, a Codelet for a certain level uses any of the supported basic SuperCodelet ISA operations, plus the Codelets of the level below. Native arithmetic operations may also be supported as extension to the SuperCodelet ISA. An example are the Codelets of L1 which are described in terms of L0, and which will use the ISA of the particular CPU architecture of L0 (See section 5.1.2).

There is a top level Codelet that I referred to as the Main Codelet. Its job is to work as entry point of the program. For now, for the sake of simplicity let us restrict that only the Main Codelet can manage I/O operations.

5.2.1 SuperCodelet ISA

Codelets are defined as sequential programs similar to assembly code. In order to describe the control flow of instructions inside a Codelet, I provide the basic set of instructions that conform the bare minimum for the description of Codelets across all levels. I call this basic set the SuperCodelet ISA and it is summarized in table 5.1. In the table, the reader can observe two type of operands: Registers (Reg), and Immediate values (Imm). The first one corresponds to a register name, while the second one is an immediate value. Let us limit the immediate values to a number of bits, in order to limit the size of the instruction encoding in memory. I assume a size of 32 bits for now, but the answer to what is the appropriate size to be use may require extra research. There are four families of operations: Arithmetic operations, control flow operations, memory operations, and Codelet control operations. The first three families are self explanatory. The Codelet control operations coordinate the communication across levels. These instructions allow instantiating Codelets from one level to the level below, and returning values after execution of the Codelet to let the upper level continue execution. This is, these operations allows communication between the levels. The `COD <name>` operation spawns a new Codelet for execution on the level below. The `COMMIT` operation informs the upper level that the Codelet has finished execution, and that the results are ready to be used.

Family	Instruction	Op1	Op2	Op3	Op4	Operation
Arithmetic	ADD/SUB	Reg	Reg	Reg/Imm	-	Op1 = Op2 +/- Op3
	MULT	Reg	Reg	Reg/Imm	-	Op1 = Op2 * Op3
	SHFL/SHFR	Reg	Reg/Imm	-	-	Op1 <<Op2 (or >>)
Control	JMPLBL	label	-	-	-	Goto label
	JMPPC	Reg/Imm	-	-	-	PC = PC + Op1
	BREQ	Reg	Reg	Reg/Imm	-	PC += (Op1 == Op2)? Op3 : 1
	BGT/BGET	Reg	Reg	Reg/Imm	-	PC += (Op1 >or >= Op2)? Op3 : 1
	BLT/BLET	Reg	Reg	Reg/Imm	-	PC += (Op1 <or <= Op2)? Op3 : 1
Memory	LDIMM	Reg	Imm	-	-	Op1 = Op2
	LDADR	Reg	Reg/Imm	-	-	Op1 = [Op2]
	LDOFF	Reg	Reg/Imm	Reg/Imm	-	Op1 = [Op2 + Op3]
	STADR	Reg	Reg/Imm	-	-	[Op2] = Op1
	STOFF	Reg	Reg/Imm	Reg/Imm	-	[Op2 + Op3] = Op1
Codelet Control	COMMIT	-	-	-	-	Finish Codelet execution.
	COD <name>	Reg/Imm	Reg/Imm	Reg/Imm	Reg/Imm	Call codelet <name>with params

Table 5.1: tab:instSuperCodelet

I do not expect the current state of the SuperCodelet ISA design to be final. It is possible to extend some of the work done by ISA definitions of other architectures (e.g. RISC-V) to provide that basic set of instructions, but also allow for upper level computation through the means of a regular ALU. For now the SuperCodelet ISA works as a proof of concept.

Another important aspect to mention is that at Level L0, legacy hardware architectures is used. Therefore the Codelets written for the lowest level use the native ISAs supported by the respective architectures.

5.2.2 Codelet Definition

Codelets are user defined. Therefore, some flexibility is needed in terms of the number of operands, the type of operands (Reg or Imm), and the direction of these operands (Read or write). The number of parameters in a Codelet, as well as the length of encoding the Codelet instruction is equivalent to the distinction between CISC and RISC architectures. It is possible to consider a RISC-like architecture, but for simplicity of the programming model I focus on a CISC-like design. Following, the direction of an operand is important to be able to discover dependencies during out of order execution. It needs to be somehow encoded in the definition of the Codelet and accessible to the runtime.

For now, I propose the following semantics to be used in conjunction with the SuperCodelet ISA defined in section 5.2.1.

```
def<L> codname(type opname:dir, type opname:dir, ...):
  ;;Codelet Body
  COMMIT;
enddef
```

There must be an assembler or compiler that translate these into actual executable code. The keyword `def` marks the definition of a Codelet, while `enddef` marks the end of a Codelet. To determine the level for which the Codelet is written, an integer is used inside of the delimiter `<L>` (e.g. `def<1>` for L1). At some point during the Codelet execution there must be a `COMMIT` instruction to finish the execution of the Codelet. Otherwise, there is an implicit `COMMIT` at the end of the Codelet definition.

The number of operands is determined by the number of elements inside the parenthesis. An operand has a `type` that is either a register or immediate value. When `type` is register, a size needs to be determined. For registers, the keyword `reg<size>` is reserved, where `<size>` is the size of the register in bytes that the given operand uses. For immediate values, the keyword `Imm<type>` is used. Immediate values are of type signed, unsigned, float, or double. And they are at most 32 bits long.

The direction of an operand is a 2 bit mask, the LSB bit for Read, and MSB bit for Write. For example a direction of $0b11$ determines an operand that is Read and Write, while $0b01$ is a read only operand.

5.2.3 An example program

I present a really simple program written for an architecture featuring 2 levels. First, I provide a pseudo-code of the intended equivalent of execution, but no translation strategy is provided. Furthermore, I do not intend this example to be an ideal use case of this architecture. This is for the sake of syntax demonstration.

```
void mainCod(int &a, int &b, int &c){
    if (a > b)
        c = a + b; // L0 Codelet Sum
    else
        c = a - b; // L0 Codelet Sub
}
```

Observe that the code written in C has no concept of levels. I have added comments to guide the reader in the translation process.

The above pseudocode can be described as follows in the SuperCodelet Programming Model. Listing 5.1 corresponds to the definition of the main Codelet of level L1. Notice that the operand names `a`, `b`, and `c` are seen as addresses inside of the Codelet definition, and therefore are used as arguments of the memory operations. This is because for L2 (who will be calling the Codelet), the operands are registers, but for L1 (who will be executing the Codelet) the operands are memory locations. In the case of immediate values, the value is propagated inside the Codelet at compile time. Registers used inside of this definition correspond to registers of level L1. The convention for register naming inside the definition of a Codelet is `R<size>_regNum`.

```
def<1> main(reg<8> a:01, reg<8> b:01,
            reg<8> c:10):
   ;; a, b and c, are the address in L2
   ;; of these arguments.
   LDADR   R8b_1, a;
   LDADR   R8b_2, b;
   BRLET   R8b_1, R8b_2, R8b_3;
   COD sum R8b_1, R8b_2, R8b_3;
   JMPPC   2;
   COD sub R8b_1, R8b_2, R8b_3;
   STADR   R8b_3, c;
   COMMIT;
enddef
```

Listing 5.1: Main Codelet definition at level L1

Following with the example, I show the definitions of the `sum` and `sub` in Listings 5.2 and 5.3 respectively. These two Codelets are created for Level L0. The most important aspect of these two listings is that the code is written in RISC-V assembly code in addition to Codelet Control operation COMMIT. Again, `a`, `b`, and `c` are addresses of L1. In fact registers `a4` and `a5` are both 4 bytes. If we were to use the whole 8 bytes, we would have to perform 2 `lw` operation. This behavior is expected and encouraged.

```
def<0> sum(reg<8> a:01, reg<8> b:01,
            reg<8> c:10):
   ;; Using RISC-V architecture
   ;; a, b, and c are the address in L1
   ;; of these arguments
   lw       a4, -0(a)
   lw       a5, -0(b)
   addw     a5, a4, a5
   sw       a5, -0(c)
   COMMIT;
enddef;
```

Listing 5.2: Sum Codelet definition at level L0

```
def<0> sub(reg<4> a:01, reg<4> b:01,
            reg<4> c:10):
   ;; Using RISC-V architecture
   ;; a, b, and c are the address in L1
   ;; of these arguments
   lw       a4, -0(a)
   lw       a5, -0(b)
   subw     a5, a4, a5
   sw       a5, -0(c)
   COMMIT;
enddef;
```

Listing 5.3: Sub Codelet definition at level L0

In order to create an evaluation strategy of this architecture, it is necessary to define simulation and emulation of the hardware and runtime. Following chapters will provide an initial evaluation of the runtime.

Chapter 6

SCMULATE. AN EMULATION RUNTIME FOR THE SEQUENTIAL CODELET MODEL

The definition of the Sequential Codelet Model and its application in the SuperCodelet architecture (As explain in Chapters 4 and 5 respectively), are meant as basis for the design of computer systems. These concepts allow to design a Hardware/Software co-design strategy that may be applied to general purpose compute architectures as well as high performance parallel systems. However, success of hardware/software co-design heavily relies on both simulation and emulation of the desired system.

Emulation imitates the behavior of a system while not necessarily performing the same steps. On the other hand, simulation aims to replicate the behavior of the system as faithful as possible. Emulation provides us with early results expected from the construction of the system. Due to the complexity of computer systems, it is often necessary to mix and match both emulation and simulation approaches.

To this end, I have created a runtime, namely SCMUlate (pronounced S.C.Emulate), that emulates the behavior of the Sequential Codelet Model with an organization similar to the one presented in the SuperCodelet architecture. The objective of this runtime is twofold: First, perform an early evaluation of the SCM design when implemented with already existing architectures. Through the use of micro benchmarks, the runtime provides an assessment of the expected performance of sequential Codelet programs on commodity hardware architectures. Furthermore, it evaluates the expected performance gains in programs optimized through Codelet Level Parallelism techniques, as seen in section 4.5. Second, create an strategy for exploring the large search space of the system design that implements the Sequential Codelet Model. There exist an extensive set of design parameters such as: 1) appropriate register dimensions for different levels of the architecture, 2) efficient Codelet sizing in terms of memory and compute complexity, 3) relationship between compute capacity, memory sizes, and memory bandwidth, and 4) other parameters that are required in the construction of an actual system.

This Chapter summarizes the implementation of the SCMUlate emulation platform. It introduces the software infrastructure that consist of an interpreter, a programming API for Codelets, and a runtime. It shows the strategy for creating Codelets, as well as use them in a program. Furthermore, it shows different Codelet Level Parallelism techniques that are implemented. SCMUlate is Open Source and publicly available at [84].

6.1 SCMUlate software design

The SCMUlate system is a parallel software written in C++ that uses an Object Oriented Programming approach to emulate the different parts of the SCM model. SCMUlate is meant to run on commodity multicore hardware. SCMUlate creates a class for each of the elements of the SCM system. Figure 6.1 shows a simplified UML diagram of all the classes that are used by SCMUlate. The project is maintained in GitHub and uses a CMakeFile build system for easy compilation and deployment.

6.1.1 The role of OpenMP

SCMUlate uses OpenMP for the implementation of the runtime, but not for the description of parallelism. The use of OpenMP is dispensable, and it could be replaced by a lower level PThreads implementation. Specifically, OpenMP is limited to an initial runtime thread creation, synchronization of runtime signals, runtime begin/end barriers, and, in the case of GPU Codelets, to take advantage of GPU code generation in the compiler. An OpenMP parallel region is spawned after creation of the runtime. Each thread is assign a different role (i.e.. SU, CU, or memory) depending on the OpenMP thread ID. A configuration file statically determines the mapping

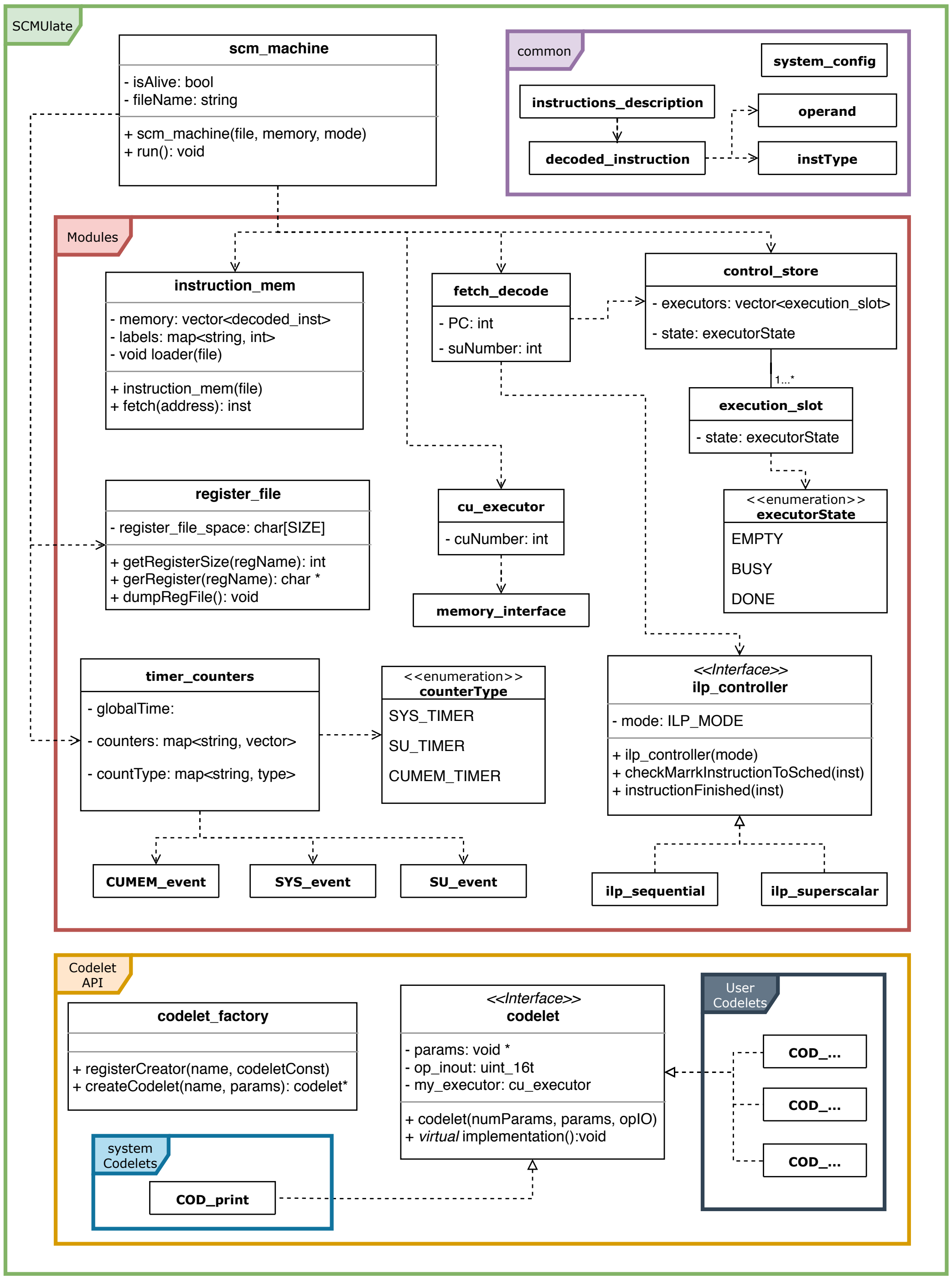


Figure 6.1: SCMUlate UML Classes Diagram

between roles and threads. Additionally, after runtime initialization, it is necessary to synchronize the elements before starting the SCM computation. Likewise, once the SCM program has completed, it is necessary to synchronize the threads for runtime finalization and proper garbage collection. For these two moments an OpenMP barrier is used. Finally, atomic operations on control variables of the runtime (e.g. full empty bits) are achieved by using OpenMP atomic directives.

It is crucial to understand that I am not relying on OpenMP to perform parallel computation, but instead to manage and create the runtime of the SCMUlate emulator. Thus, once SCMUlate starts executing parallel execution of compute code is driven only by the emulation of the SCM execution model.

6.1.2 Folder infrastructure

The project is organized in a `src/include/test` folder infrastructure, for source code, header files and tests respectively. The Source code is divided in subfolders for the different parts of the program.

A `common` folder contains header files with system configuration variables (e.g. size of queues, and number of operands) and thread configuration variables (i.e. CU and SU thread mapping). Additionally, this folder contains the semantic description of the SCM basic ISA (as described in section 6.2.3) in the form of regular expressions for the different ISA instructions, as well as the format used for the instruction after it has been decoded by the interpreter. Finally, this folder contains helper functions for Info/Error/Warning messages formatting, and string manipulation tools.

Following, there is a `modules` folder that contain different files, one for each part of the SCM machine. An SCM module maps one to one to a physical block of the SCM machine, working as building block of the overall system. Each module is represented as one or more classes in C++. Currently, the following modules have been created:

- **Fetch/Decode:** SU logic, control flow and basic arithmetic instruction evaluation.

- **Executor and Memory Interface:** CU logic and memory interface logic. It contains the runtime function that checks for available work and executes it. Currently, the CU acts as both Memory and Compute units.
- **Control Store:** Similar to the reservation tables described in the SuperCodelet architecture. This module contains the execution slots (or queues) that connect the SU with the CU. Allowing the SU to schedule work.
- **instruction memory:** Storage for the program instructions, as well as entry point for the interpretation of the SCM assembly program.
- **Register File and Registers Configuration:** Containing the data structure that represents the register file with the different register sizes. It also allows for dumping the register file, as well as definition of the available register sizes.
- **Instruction buffer:** Buffer for the window of instructions that are currently inside the pipeline.
- **ILP controller:** different implementations of Instruction Level Parallelism techniques. It modifies the ability of the system to discover dependencies and eliminate false dependencies. Currently there are three modes of execution: Sequential, Superscalar, and Out of Order.
- **Timers and Counters:** Representation of hardware counters, it is in charge of profiling as well as real hardware counters interface.

Following the folder infrastructure, there is a `Codelet model` folder that contains the API for the description and definition of Codelets. This is the programming API for defining Codelets of level L1 in terms of code of level L0. We take advantage of C++ compiler code translation for generating the Codelets in L0. Additionally, there is a folder for `system Codelets` that contains pre-defined Codelets that can be used in any application. Currently, the only System Codelet that exists is `cod_print` which prints the content of a register.

The last folder that composes the source code of the emulator is the `machines` folder. It contains the top level class of the SCM Machine emulation. This class works as entry point for the whole emulator as well as a container for all the different modules that compose the machine.

Finally, there is an `apps` folder in the top level of the project that contains the different microbenchmarks used.

6.1.3 The SCM Machine emulation

There are three different roles a thread can be assigned in SCMUlate: Scheduler (SU), Compute (CU), and Memory (MEM) (as described in sections 5.1.1, 5.1.2, and 5.1.3 respectively). Threads that are assigned the compute unit role evaluate (fire) Codelets by calling its implementation function. Threads that are assigned the memory role are in charged of the interaction with memory of the level above. This is, they perform load and store operations. Emulation of the memory hierarchy and memory operations are explain later on in this document together with the construction of the register file. Finally, A single thread is assigned the Scheduler role. This thread helps emulating the Sequential pipeline of the SCM machine. That is: fetch, decode, and scheduling of units, basic ALU operations, and Codelet Level Parallelism book keeping. This thread is not considered to be part of the computation of the program, instead it emulates the behavior of the SCM machine pipeline.

The Sequential Codelet Abstract Machine (see 4.3) associates a single SU to multiple CUs in each level. The current implementation only supports emulation of a single L1 level. Hence, there is only a single SU that manages all other CUs. Future implementations may include multiple L1 components and emulation of other levels of the SCM hierarchy, therefore, requiring multiple SUs.

There is a class for each element of the SCM model. Following I discuss all the different classes, as described in Figure 6.1 that compose the machine emulation.

6.1.3.1 Instructions and Instruction Memory

Instructions are encoded in a format that assimilates binary encoding of commodity ISAs. An interpreter takes the assembly-like program in string format, and translate them to a list of objects of class `decoded_instructions_t`. The output of the interpreter is an STL vector of instructions representing the instruction memory, where the offset in the vector corresponds to the instruction address, as fetch by a program counter of level L1.

A decoded instruction has an instruction type (e.g. Control Instruction, Memory instruction, and Codelet), an opcode, a list of operands, a bit mask that contains the direction of all the operands (read, write or readwrite), and other instruction type specific information. The decoded instruction also contains the strings for the instruction and operands for debugging information.

There are five type of instructions defined by the enumerator class `instType`: 1) a `CONTROL` instruction type that refers to an operation that directly modifies the program counter (PC) through jumps or conditional jumps. 2) A `BASIC_ARITH` instruction type corresponding to basic arithmetic operations (e.g. ADD, SUB, and MULT). 3) The `EXECUTE` instruction type which corresponds to Codelets. 4) A `MEMORY` instruction type that denotes a memory load/store operation. And, 5) the `COMMIT` instruction type which marks the end of the execution of a Codelet.

All but the `EXECUTE` instructions are determined by the SuperCodelet Codelet Set Architecture explained in section 5.2.1. On the other hand, `EXECUTE` instructions are user defined and require creation of Codelet object as will be explained later in this chapter. Codelets are created and stored in the `decoded_instruction_t` class. Memory operations require careful handling due to their ability to communicate with the upper level memory and the need to keep memory operation's order. When an instruction is of type `MEMORY`, the `decoded_instruction_t` also stores range of memory addresses that it uses. These ranges allows the runtime to discover memory dependencies and allows to maintain their order. Memory ranges are a pair of memory sets, one for read operations and one for write operations.

Operands are encoded in its own class as well. There needs to be a differentiation between immediate values, labels and register names in an operand. Also, each operand has a direction that will be used during data dependency analysis. The `operand_t` class stores operand information. This class contains an operand type (i.e. `REGISTER`, `IMMEDIATE` or `LABEL`), an operand value is stored in an `union`. There are also boolean flags that indicate if the instruction reads and/or writes the register in the operand. A full/empty bit is also included and it indicates if the value is available at runtime or not (for certain Codelet Level Parallel execution modes as explain later in this chapter).

A union value has been used since it allows to store either an immediate value, the PC for the location of a label, or the address of a register within the register file. Registers are represented as pointers to the location of the specific register. The `register_file` module provides the necessary methods to translate a register name into its corresponding pointer. The address is determined by the register size and number. I remind the reader that these registers refer to registers of the level L1 of the architecture and not the architecture registers of Level L0 registers (see 4 chapter for more information).

There is support for using labels in the L1 assembly code to name specific locations in the instruction memory and jump to them through control flow instructions. When a label is encountered during interpretation of the assembly code, the location is stored in a map that associates an label string to a memory location. Then, during operand decoding, the immediate value of the `operand_t` is set to the program location of the specific label.

6.1.3.2 Fetch, decode and execute

The `fetch_decode` class is in charge of scheduling instructions according to the current value of the program counter, and the instruction's state. This module uses the `instruction_buffer` module to represent the window of instructions that are currently inside of the SCM execution pipeline. This buffer has an STL Deque container that stores (`decoded_instruction_t, instruction_state`) pairs. The `fetch_decode` uses the `fetch()` method in the `instruction_memory` to obtain the instruction in the current program counter (PC) and insert a copy of the `decoded_instruction_t` into the `instruction_buffer`. The `fetch_decode` also contains an `ILP_controller` that manages the execution mode of the architecture. The ILP controller is in charge of data dependency management and bookkeeping of the different execution modes (e.g. sequential, superscalar or Out of Order).

An instruction inside the `instruction_buffer` can be in one out of six states. The `instruction_state` represents the current status of the instruction in the pipeline. However, the use and interpretation of each state depends on the execution mode of the Codelet Level Parallelism, as we will discuss in section 6.1.3.4. The 6 available states are: `WAITING`, `READY`, `EXECUTING`, `EXECUTION_DONE`, `DECOMMISSION`, and `STALL`.

The decode process uses the instruction's type to determine the unit in which the instruction should be scheduled. If the instruction is of type `CONTROL`, `BASIC_ARITH`, or `COMMIT`, the `fetch_decode` class will execute it. `CONTROL` instructions will directly modify the value of the PC. `BASIC_ARITH` corresponds to basic Arithmetic Logic Unit operations (ADD, SUB, and MULT). And the `COMMIT` instruction will finish the execution of the current Codelet. Given that SCMUlate currently only evaluates one unit of level L1, this instruction will finish the execution of the whole program.

The `EXECUTE` and `MEMORY` instruction types need to be assigned into a Compute Unit (CU), represented by the `cu_executor` class. The communication between `fetch_decode` and the `cu_executor` occurs through the `control_store` class. The `control_store` class uses `execution_slots`. These slots are queues that represent the reservation tables of the SuperCodelet architecture. These queues contain pointers to the pairs stored in the `instruction_buffer`. Each CU has its own `execution_slot` and the CU thread constantly iterates over it waiting for work to be assigned.

The structure of the `Codelet` class, and how new Codelets are defined will be explained later in this section. When the `cu_executor` receives an instruction of type `EXECUTE`, it evaluates the Codelet by calling its `implementation()` method. The Codelet object and its arguments are first created during interpretation of the user's Codelet assembly program, and stored in the `decoded_instruction_t` class. When the `decoded_instruction_t` is copied, the Codelet is also copied.

The `cu_executor` uses the `memory_interface` to execute instructions of `MEMORY` type. There are some instructions as part of the basic Codelet ISA that allow loading and storing registers from and to the upper level memory. There are also some special kind of Memory Codelets that also require a memory region that shows what regions the Codelet accesses. The user is in charge of providing the function to calculate the required regions.

6.1.3.3 The L1 register file on Cache based systems

The memory hierarchy of the Sequential Codelet Model is a key property that distinguishes it from other models. However, when emulating SCM in commodity hardware that are based on Von Neumann architectures, the control over the memory hierarchy is limited. In particular, systems that rely on cache memory, does not allow for fine grain control of memory allocation in the on-chip memory. To that end, the following mapping approach is proposed.

Cache is a mechanism that sits between the core and main memory, and it aims to reduce memory access time. For each location in main memory, there is a set of possible locations in cache that temporarily store the values of the main memory location. Cache takes advantage of temporary locality and spacial locality. The former happens when consecutive memory accesses to the same location occur. The latter is achieved by fetching a larger memory region every time a location is accessed, therefore, fetching more values around the original memory location. The term cache line is used to refer to the size of memory accessed by the cache memory. The cache line is bigger than the main memory access word, and it is usually equal to 64 bytes.

Cache is arranged in a hierarchical organization. Level 1 and level 2 caches are usually private to the each core. On the other hand, level 3 cache is usually shared across the different cores of the chip. Often, the higher the cache level, the larger the memory space. In order to emulate the register file there are two assumptions:

1. For a consecutive memory region allocated in physical memory. If the size of that consecutive memory regions is the same as the size of cache. Then, every memory location in this region will be assigned a different location of that cache if accessed consecutively. Therefore, allowing indirect control of cache allocation. And,

2. the overhead of the runtime is low enough to not cause cache thrashing. Therefore, allowing consecutive memory access in time to always land in cache.

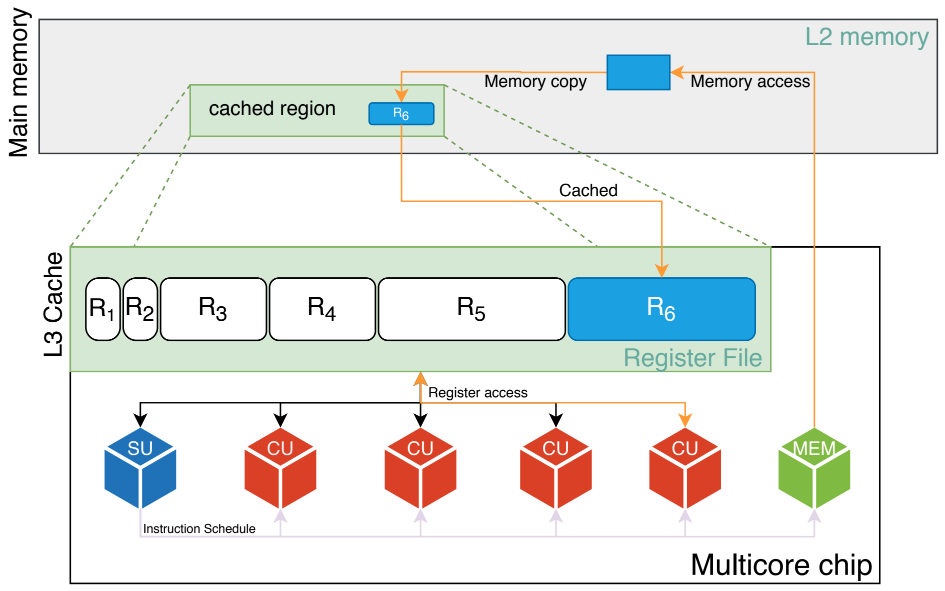


Figure 6.2: Register file emulation mechanism on SCMUlate

While these two assumptions are not necessarily always true, we expect that they hold for the most part of the execution.

Therefore, the register file in the L1 abstraction of the SCM is emulated by allocating a memory region as large as level 3 cache. Level 3 cache is used since the register file is shared among all the cores, allowing Codelet-to-Codelet communication through the last level of cache. Figure 6.2 shows the whole emulation process. The register file region is allocated consecutively, but divided into multiple registers of different sizes. In the figure, all R_x blocks in L3 are registers. We use multiples of the cache line size for alignment. For example, in the registers shown in the figure, length of R_1 and R_2 may be 1 cache line long, R_3 and R_4 100 cache lines long, and R_5 and R_6 1000 cache lines long. When a memory instruction is issued, a memory copy from a region in main memory (that represents a location of L2 memory in the SCM abstraction) is copied to the location of R_6 in main memory. Due to caching, the location will also be fetched into level 3 cache. Therefore, access from the different CUs to this region of memory will (allegedly) have a shorter access time.

The `register_file` class is in charge of allocating a memory region as large as level 3 cache. The datatype is char since this type is usually 1 byte long. However, registers of different sizes need to be accessed independently. Therefore, a `union` in C++ is used to overlap the allocated memory region and the different registers. Furthermore, methods to obtain a translation between strings in the assembly Codelet code, the actual locations in memory and the size in bytes is created. Finally, a debugging method is used to dump the content of the whole register file.

6.1.3.4 Codelet Level Parallelism

So far I have described the different components of the SCM machine. However, in order to achieve parallelism, it is necessary to create the out of order mechanisms. The `ilp_controller` class contains the logic for bookkeeping the dependencies between instructions. There are currently three implementations in place: 1) a sequential version that does not exploit any parallelism whatsoever. 2) A superscalar version that, if there are enough CUs available and there are no dependencies between instructions, they are scheduled in parallel. And 3) an out of order execution that performs register renaming, eliminating false and anti dependencies during the code execution. A fourth version is being developed that aims to optimize memory operations to reduce the number of copies needed for executing loads and stores.

For `ilp_superscalar` and `ilp_ooo` (out of order) it is necessary to keep track of memory locations. A `memory_queue_controller` class serves as memory controller to determine if an instruction can be executed or not. This class emulates the logic in the memory stage of the SCM machine (see the SuperCodelet description), which discovers dependencies between memory operations. This class stores the memory ranges currently under execution, and it provides a method to determine if a given range overlaps any of the other instructions that are currently in the pipeline.

6.1.3.4.1 Sequential execution mode:

The `ilp_sequential` is the simplest version of the ILP controller. The controller has a full/empty bit that determines if there is an instruction on the execution state. Under this mode of operation an instruction is fetch and place in the `WAITING` `instruction_state`. If there are no instructions in the execution stage, as determined by the full/empty bit, the instruction set as `READY`, otherwise the instruction is set as `STALL`. Ready instructions are then assigned to the appropriate execution unit (SU or CU) and placed in the `EXECUTING` state, once the execution is finished, the instruction state is changed to `EXECUTION_DONE`. At this point the full/empty bit of the `ilp_sequential` controller is set to empty, and the instruction is marked for `DECOMMISSION`. The `instruction_buffer` is in charge of removing the instructions that are marked for decommissioning.

6.1.3.4.2 Superscalar execution mode:

The `ilp_superscalar` controller uses register names and operand directions to determine if two instructions do not depend on each other. Read after read dependencies (RAR) and completely independent instructions are allowed to execute in parallel. When a anti-, output- or true-dependency is encountered, the pipeline is stalled. Under this mode of operation, the instructions start in the `WAITING` state. The `ilp_superscalar` controller has a set of busy registers that stores the register name and direction. This set is used to determine if an instruction can be scheduled or not. If there are no dependencies, the instruction state is set to `READY` and the `fetch_decode` unit assigns it to the corresponding unit (SU or CU). If the instruction is of type `MEMORY` or it is a memory Codelet, then the memory controller is used to determine if there are any memory dependencies. If the instruction has a data dependency in registers or memory, the instruction is set to the `STALL` state, and the pipeline is stalled. After an instruction has finished execution, its state is set to `EXECUTION_DONE`. the `FinishInstruction()` method in the ILP controller is called to remove the busy registers and memory ranges (if any). Once the ILP is updated the instruction is marked for decommission.

6.1.3.4.3 Out of Order (OoO) execution mode:

The `ilp_ooo` controller is the most complex controller of all three. There are 6 different containers storing information regarding busy registers, renamed registers, already processed instructions and dependency management for when an instruction is finished. A register state determines the current operation being applied to that register. A register state can be `NONE`, `READ`, `WRITE` or `READWRITE`. When an instruction is fetch the current state of the register is compared to each of the operands of type register in the instruction, and their direction, to determine dependencies or make decisions about register renaming. Additionally, the `ilp_ooo` has a hidden register file of its own used for renaming.

The first container is the `used` map. This map stores the current state of a register inside of the execution pipeline. It associates a register name to a state. If a register is used by any operation in the pipeline, then this container will include the register and the operation being applied. This container is used to determine the relationship between the incoming instruction, and the instructions that are already present in the `instructions_buffer`. Depending on the direction of the instruction, and its state value on the `used` map, an instruction may be marked as ready, left waiting, or stall the pipeline. Likewise a register may be renamed to remove output dependencies across instructions.

Two more containers, `registerRenaming` and `renamedInUse`, associate already renamed registers. The former is a map that associates the original register name to the newly renamed register name. This is used to re-associate any other read reference to this register in future instructions. The latter is a set that stores all the already used registers in the hidden register file, to avoid re-utilizing a renamed register.

Another container is the `reservationTable`. This is a set of pointers to pairs in the `instruction_buffer` previously described. This set stores all the instructions that have already been processed and which should skip the dependency analysis. It is important to run the dependency analysis only once when the instruction is fetch.

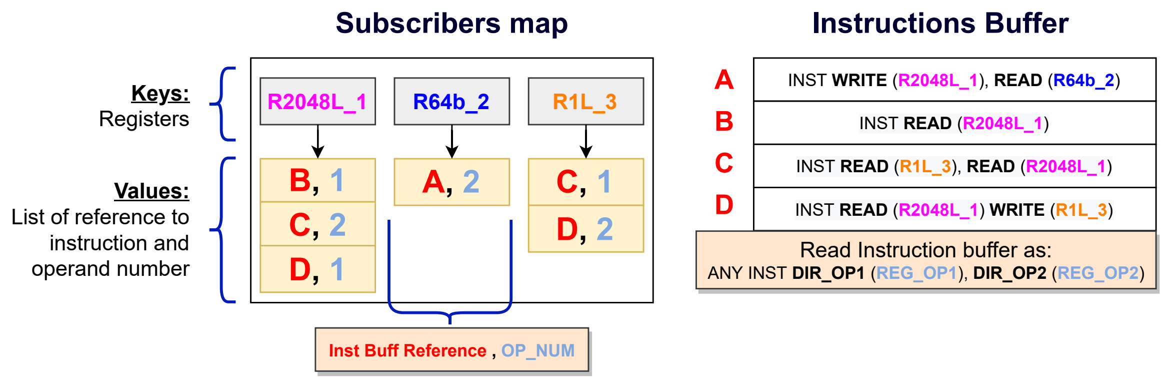

Subscribers map
Instructions Buffer
Keys:
Registers
Values:
List of reference to instruction and operand number
R2048L_1
R64b_2
R1L_3
B, 1
C, 2
D, 1
A, 2
C, 1
D, 2
Inst Buff Reference , OP_NUM
A
B
C
D
INST WRITE (R2048L_1), READ (R64b_2)
INST READ (R2048L_1)
INST READ (R1L_3), READ (R2048L_1)
INST READ (R2048L_1) WRITE (R1L_3)
Read Instruction buffer as:
ANY INST DIR_OP1 (REG_OP1), DIR_OP2 (REG_OP2)

Figure 6.3: Example of the content in the `subscribers` map with respect to the instructions in the `instructions_buffer`

The remaining two containers are meant to emulate the bus in the Tomasulo's algorithm that inspired this design. It connects the production of an operand value from an instruction to the respective consumer operand in another instruction. We define a new type of pair ((`instruction_decoded_t*`, operand number)) called `instruction_operand_ref_t` which associates an operand number to an instruction in the `instructions_buffer`.

A container called `subscribers` is a map that keeps a reference to all the `instruction_operand_ref_t` that reads a given register. An example of the relationship between the `instructions_buffer` and the `subscribers` map is depicted in figure 6.3. The key of this map is the register name, and the value is a vector of `instruction_operand_ref_t`, each containing a reference to a location in the `instructions_buffer` and the corresponding operand number. This container allows a write operation that has finished to enable all the consumers. Furthermore, it allows us to remove a register from the `used` container when there are no other instructions referencing this register.

In the example of Figure 6.3, once Instruction **A** has finished, the ILP controller will enable the operand 1 of instruction **B** and **D**, and the operand 2 of instruction **C**. Furthermore, the register `R64b_2` will have no more references once instruction A has finished, therefore, it may be removed from the `used` container. Register `R1L_3` are currently used in instructions **C** and **D**. Since there is no instruction writing to this register, these two instructions will be ready to execute. Once **A** finishes, all three instructions **A-D** will be marked as ready.

A second map called `broadcasters` have a similar functionality, but it serves a special case when instruction operands feature a `READWRITE` direction. When an operand is `READWRITE`, the content of the register will be modified. If a register has subscribers, then it is necessary to rename the operand and copy the original values of the register into the newly renamed register. However, if an operand `used` state is `WRITE`, the value is not yet available, and the instruction may not proceed. When this happens, a broadcaster subscription is issued instead of a regular `subscription`. When the write finishes, the value of the register will be copied to the renamed register of the READWRITE operand.

For example, let us change operand 1 of instruction **D** of figure 6.3 for a `READWRITE` direction. When instruction `A` finishes, all three instructions **B-D** will require the resulting value. However, if **D** executes in parallel with **B** and **C**, **D** may change the value of register `R2048L_1` before **B** and **C** read its content. To avoid this, instruction D's operand 1 is renamed to another register (say) `R2048L_ren_1`. When **A** finishes execution, the resulting value of `R2048L_1` is "broadcasted" (copy) to `R2048L_ren_1`, allowing all three instructions **B-D** to be executed in parallel.

All three `ilp_controller` implementations use two methods: `checkMarkInstructionToSched()` and `instructionFinished()`. The former is executed every time a new instruction is fetch, the later is executed every time an instruction has finished.

6.1.4 Configuration and common structures

There exist a set of configuration file that contain parameters to define the different components of the SCM emulation. Most of the parameters are defined through C Macros. Starting with register file, configuration parameters include register sizes, total register file size, register names and helper functions Second, thread role assignment for the different OpenMP threads, as well as the number of CUs are determined in another file. Third, the ILP mode of the executing machine (i.e. sequential vs superscalar) is determined through an enumeration variable that is defined in the configuration files. Additionally, the size of the reservation table used by the ILP superscalar class, the maximum number of operands supported by the instructions in the ISA, the size of the `instructions_buffer` and `execution_slots` are all configurable as well.

In addition to configuration parameters. There are a set of string manipulation helper classes that are used through the interpretation of the assembly. Besides, definition of the different supported instructions in the assembly Codelet program requires the use of regular expressions that determine the structure of each instruction, as well as labels and comments in the assembly file. An extensive set of assembler operations are also defined in the common files.

Finally, we have created different helper macros to include debugging information for the runtime. This information has different levels of verbosity and they distinguish between errors, warnings and information messages. These tools are an adaptation of previously defined macros used in a different software project for the creation of OpenMP Validation and Verification tests [21]. Debugging information and level of verbosity can controlled by means of a defined macro that is accessible through CMake.

6.1.5 The Codelet Class

Thanks to the use of inheritance in object oriented programming, it is possible to create an interface class that works as a recipe for other classes. Interfaces also allows to create inherited Codelet classes with user defined behaviors. Therefore, the use of an interface seems natural for the creation of user defined Codelets. However, there is still a challenge to overcome. C++ does not support dynamic type checking. Therefore, information about the class name is usually lost at runtime. On the other hand, the creation of an interpreter for the assembly Codelet code requires runtime matching of class constructors. A factory method pattern is used to solve this issue.

First, the SCMUlate runtime needs to be aware of the name of the of the user defined Codelets in order to connect it during runtime when interpreting the SCM assembly. C++ does not support dynamic type creation at runtime by default. The RTTI is limited to type information, but for the construction of objects from strings, it is necessary to use a factory method pattern that allows runtime creation of any Codelet.

First, a registration process occurs at the beginning of the SCMUlate runtime. Compilers often support attributes that let the user annotate code and modify the behavior of the compiler, linker or runtime. The `__attribute__((constructor))` is an attribute supported by most compiler vendors. It allows for a function to be called before the beginning of a program. Special care needs to be placed when it comes to construction of data structures that have dependencies. This is due to the lack of a guaranteed order of evaluation of the different functions that use the constructor attribute.

Our approach is to create an static class named `codeletFactory`. This class contains a map between a string and a Codelet constructor. Additionally, a registration method called `registerCreator()` is added to this class to allow for Codelet constructors to be added to the map. Additionally, an static method is included as part of the definition of user Codelet classes. This method, called `codeletRegistrer()`, is marked with the `__attribute__((constructor))`, therefore it is executed at the beginning of the program. A second static method, called `codeletCreator()` is used to encapsulate the Codelet constructor to overcome some limitations. Yet another static method namely `registerCodelet()` directly adds the `coodeletCreator()` method to the `codeletFactory`. Finally, the `codeletRegisterer()` calls the `registerCodelet()`.

This complicated process is required to remove the burden to the programmer to have to register each Codelets at the beginning of the program.

A memory Codelet is a Codelet that access memory directly. A memory Codelet must be marked as such at runtime, so the `ilp_controller` can properly identify the memory dependencies. A memory Codelet has an additional initialization process that calculates the memory regions that are read or write during the execution of the Codelet. These regions need to be exposed and calculated to the SCMUlate runtime before starting the execution of the Codelet, to guarantee that no dependencies are violated.

Finally, to support parameters of different type (i.e. register and immediate values), there is a `codelet_params` class that stores a list of Codelet parameters. Each parameter is a `unsigned char *` which can be interpreted as a register, or as an immediate value of up to the size of a pointer in the given architecture (usually 64 bits). In addition, a parameter has a read/write direction and a flag indicating if it corresponds to a memory address or not. A missing memory address parameter must stall the execution of the pipeline, guaranteeing that ranges can be calculated.

The user casts an operand to the appropriate type by using the templated method `getParamAs<T>(int paramNum)`. The user must know what the operand type is, and appropriately handle its. When there are multiple parameters, this pointer is cast to an array of `unsigned char` pointers. In the following section we explain how to use macros to create Codelets as well as the different instructions to be included in the assembly Codelet program.

6.2 Programming API and the assembly Codelet program

So far we have described the design and organization of the SCMUlate software. This section describes the application programming interface (API) that allows for a user to define new Codelets, and create assembly Codelet programs.

6.2.1 Codelets definition

Codelets are first defined in C++ an then used in the assembly Codelet program. C++ allows for compiler code generation and optimizations for the L0 Codelets. Therefore, programmer is not required to write code in L0 assembly (e.g. x86 ISA). Section 6.1.5 explains the behavior of the Codelet class and its interaction with the runtime. As mentioned in this section, in order to define Codelets, it is necessary to create a new class that inherits from the interface class `Codelet` (see Figure 6.1). However, in addition to inheritance from this class, the new class needs to include some static methods to allow registering the class into the runtime.

To ease creation of Codelet types we provide several C Macros: a first macro, `DEFINE_CODELET(name, numParams, OP_IO)`, is used for declaring a new Compute Codelet class, together with the appropriate static methods. a second macro, `DEFINE_MEMORY_CODELET(name, numParams, OP_IO, OP_ADDR)`, is used for declaring a new Memory Codelet class, which will be marked as such, and will contain the methods to calculate the memory ranges. A third macro, `MEMRAGE_CODELET(name, code)` allows the user to specify the function that calculates the memory ranges. Finally, a fourth macro, with signature `IMPLEMENT_CODELET(name,body)`, is used for implementing the Codelet's instruction by providing the `body` of the `implementation()` method.

The `name` parameter in these macros is used to compose the name of the Codelet class (e.g. `myCod` results in the name `_cod_myCod` class). Next, the `numParams` describes how many arguments the Codelet takes. The `OP_IO` parameters contains a bit mask that describes the read and write actions over each Codelet operand. The `OP_ADDR` is another bit mask that uses a single bit per operand to determine if an given operand is an address or not.

The bitmask that determines the `OP_IO` separate read from write for considering the readwrite direction of an operand. Starting from the LSB, and alternating the read and write bits, every two bits represent an operand. For example, the value $0b101101$ represents a read operation on the first operand (01), a read and write operation on the second operand (11), and a write operation on the third operand (10). The bits 00 are reserved for immediate values or unused operands. Bitmasks are defined as a class called `scm::OP_IO` with a collection of `static constexpr std::uint_fast16_t` values, one for each operator read or write.

The bitmask that determines the `OP_ADDR` contains a single bit per operand. Starting from the LSB, each bit is associated to a different operand. For example the value $0b000011$ says that operands 1 and 2 are used to calculate the address range of a memory Codelet. Bitmasks are defined as a class called `scm::OP_ADDRESS` with a collection of `static constexpr std::uint_fast16_t` values that allows marking operands as addresses.

6.2.2 Example of a print Codelet

There is a list of predefined Codelets that I refer to as system Codelets. The print Codelet is an example of them, and it dumps the value of the register that is passed to it. For easy implementation, we have asked the user to specify the length of register as part of the parameters, in an immediate value. However, future implementations should be able to obtain this value from the register name description.

Listing 6.1 shows the definition of the print Codelet. Line 1 uses the `DEFINE_CODELET` macro. The name of the class created under the hood is `_cod_print`, however the user will use the `print` name to refer to the Codelet during the assembly Codelet program. This Codelet receives two parameters: 1) the register to be printed, and 2) the length of the register as an immediate value. The register in the first parameter is read only. Therefore, the `OP_IO` is set to `scm::OP_IO::OP1_RD`. Following, there are lines 3 through 17 contain the implementation code. The `IMPLEMENT_CODELET` macro on line 3 indicates the implementation applies to the `print` Codelet.

```
 1 DEFINE_CODELET(print, 2, scm::OP_IO::OP1_RD);
 2
 3 IMPLEMENT_CODELET(print,
 4   // Obtaining the parameters
 5   unsigned char *reg = this->getParams().getParamValueAs<
        unsigned char *>(0);
 6   uint64_t len_in_bytes = this->getParams().getParamValueAs<
        uint64_t>(1);
 7
 8   std::cout << " = 0x";
 9   for (uint64_t i = 0; i < len_in_bytes; i++)
10     std::cout << std::setfill('0')
11               << std::setw(2)
12               << std::hex
13               << static_cast<unsigned short>(reg[i] & 255)
14               << (i%2 != 0? " ":"");
15   std::cout << std::endl;
16 );
```

Listing 6.1: The Print Codelet implementation

Continuing with the implementation of the `print` Codelet, line 5-6 is used to obtain the parameters. The memory location that points to the given register is obtained in Line 5, and Line 6 obtains the immediate value that contains the size, in bytes, of the register to be printed. Finally, lines 9 through 16 prints the actual value of the register.

Use of the print Codelet is shown in the sample assembly Codelet program in listing 6.2. This code is written for the L1 level of the SCM machine being emulated. As can be seen there are also other instructions that form the assembly Codelet program ISA. Next section will describe these instructions.

Group	Instruction	OP1	OP2	OP3	Description
CONTROL	JMPLBL	LABEL	-	-	PC = LABEL;
	JMPPC	IMM	-	-	PC += IMM;
	BREQ	REG	REG	IMM/LBL	(OP1 == OP2) ? PC = (LBL? LBL : PC + IMM) : PC += 1;
	BGT	REG	REG	IMM/LBL	(OP1 >OP2) ? PC = (LBL? LBL : PC + IMM) : PC += 1;
	BGET	REG	REG	IMM/LBL	(OP1 >= OP2) ? PC = (LBL? LBL : PC + IMM) : PC += 1;
	BLT	REG	REG	IMM/LBL	(OP1 <OP2) ? PC = (LBL? LBL : PC + IMM) : PC += 1;
	BLET	REG	REG	IMM/LBL	(OP1 <= OP2) ? PC = (LBL? LBL : PC + IMM) : PC += 1;
ARITHMETIC	ADD	REG	REG/IMM	REG/IMM	OP1 = OP2 + OP3;
	SUB	REG	REG/IMM	REG/IMM	OP1 = OP2 - OP3;
	MULT	REG	REG/IMM	REG/IMM	OP1 = OP2 * OP3;
	SHFL	REG	REG/IMM	-	OP1 <<OP2;
	SHFR	REG	REG/IMM	-	OP1 >>OP2;
MEMORY	LDIMM	REG	IMM	-	OP1 = OP2;
	LDADR	REG	REG/IMM	-	OP1 = [OP2];
	LDOFF	REG	REG/IMM	REG/IMM	OP1 = [OP2 + OP3];
	STADR	REG	REG/IMM	-	[OP2] = OP1;
	STOFF	REG	REG/IMM	REG/IMM	[OP2 + OP3] = OP1;
CODELET	COD <name>	REG/IMM	REG/IMM	REG/IMM	CALL name(OP1, OP2, OP3);
	COMMIT	-	-	-	Finish Codelet Execution;

Table 6.1: List of SCMUlate supported instructions

```
LDIMM R64B_1, 1; // Load immediate value 1 to R64B_1
LDIMM R64B_2, 2;// Load immediate value 2 to R64B_1
ADD R64B_3, R64B_1, R64B_2; // R64B_3 = R64B_1 + R64B_2
COD print R64B_3, 8; // prints = 0x0000 0000 0000 0003
COMMIT;
```

Listing 6.2: Example of using the Print Codelet.

6.2.3 The assembly Codelet Program ISA

Inside the SCMUlate, there is an assembly-like interpreter that translates a text file with the L1 instructions, into executable binary code organized as a decoded instruction type. There are a set of instructions that are currently supported, summarized in table 6.1. These instructions are similar to the instruction table presented earlier in Section 5.2.1.

Notice that some instructions support both immediate values and register operands. For memory instructions we use square braces in the description to represent a memory access to the address that resolve inside. Immediate loads allow for direct assignment of register values.

It is worth noticing the similarities between commodity ISAs and the SCM ISAs presented in this work. This is intentional as the idea of SCMUlate is to emulate the SuperCodelet Model. Progress made in the design of other ISAs can be applied to the hierarchical design of the SCM ISA. For example, the SCMUlate ISA could be implemented by extending the RISC-V ISA. However, these instructions represent the bare minimum set for a proof of concept. Furthermore, these instructions emphasize the availability of a Turing complete system at each level of the hierarchy of the SCM model, a powerful characteristic. A consequence of this property is the ability to support in-memory compute, where certain instructions of the upper levels of the SCM operate over memory values without having to move them down the hierarchy.

6.2.4 Running the emulator

In order to use SCMUlate there are other elements that need to be provided. 1) a file containing the description of the L1 program to be executed. 2) the ILP execution mode (i.e. sequential or superscalar as of right now). and 3) a memory region to be used as L2 memory. The current status of the SCMUlate software focuses on the emulation of a single L1 machine. However, L1 memory operations map to an L2 memory. There needs to be an address translation between the values used in the L1 program, and the actual memory allocated in L2 (i.e. memory allocated in the system).

To ease deployment of the emulator, the user is required to provide the memory region that represents L2. Therefore, before calling the SCMUlate runtime, it is necessary to allocate some memory, as well as manually handle allocation of data within this memory. Future versions of the runtime should also include memory management capabilities. Listing 6.3 shows an example use of the SCMUlate for matrix multiplication. The assembly Codelet program is not shown in order to emphasize the creation and use of the runtime.

```
 1  int main () {
 2      unsigned char* memory = new unsigned char[SIZEOFMEM];
 3      double *A = reinterpret_cast<double*>(memory);
 4      double *B = reinterpret_cast<double*>(&memory[B_offset]);
 5      double *C = reinterpret_cast<double*>(&memory[C_offset]);
 6      // Initialize A, B, and C
 7
 8      scm::scm_machine * myMachine;
 9      myMachine =
10        new scm::scm_machine("matMul128x1280.scm",
11                              memory,
12                              scm::SEQUENTIAL);
13      if (myMachine->run() != scm::SCM_RUN_SUCCESS) {
14          SCMULATE_ERROR(0, "THERE WAS AN ERROR WHEN RUNNING
               THE SCM MACHINE");
15          return 1;
16      }
17      // Obtain C results
18      delete [] memory;
19      delete myMachine;
20      return 0;
21  }
```

Listing 6.3: Use example of the SCMUlate runtime

In the example, line 2 is the allocation of memory that represents a single register of L2 that is an operator of the main Codelet. I remind the reader that there is an upper level Codelet called the main Codelet. In the emulation we are building, the L1 assembly program that the user provides represent the main Codelet. That

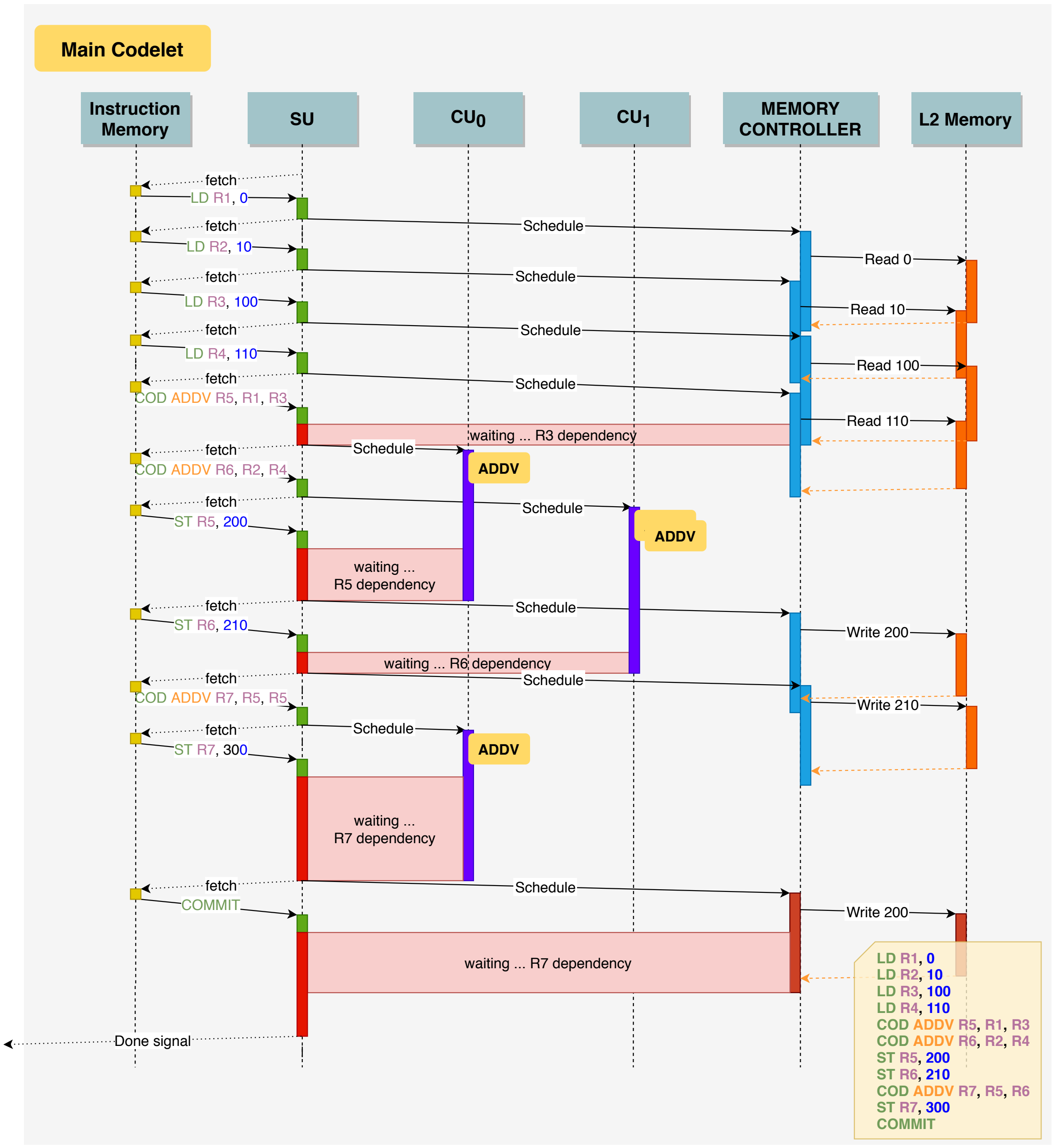
Codelet has some arguments, which are stored in the memory allocated by the user of the library. Lines 3 through 5 are part of the manual memory management process needed to refer to the L2 memory elements. In particular, we assign the addresses for A, B, and C matrices that are then used within the L1 program. Such translation is needed because the values of memory addresses assigned by the OS that is running the emulation are not known to the programmer at the moment of writing the L1 assembly Codelet program. Instead, the user will write offsets relative to the argument sent to the main Codelet (i.e. the memory allocated in line 1 and then sent to the runtime). As previously mentioned, there needs to be a better solution for memory managing in general.

Following, line 9 and 10 create the `scm_machine`. The assembly Codelet program is written in *matMul128x1280.scm*, a text file that uses the syntax described in Section 6.2.3. Additionally, we send the memory allocated, and which contains matrices A, B, and C. Also, it is necessary to specify the operation mode of the machine (i.e. sequential or superscalar currently). Continuing with the program description, lines 13 calls the runtime, followed by confirmation of a successful execution. Lines 18 and 19 are just to avoid memory leakage and clean up the runtime.

6.3 Runtime and Codelet Level Parallelism

So far we have focused on the different components as well as the API. As expected, parallelism is an optimization of the runtime, and not an essential part of it. In this section we focus on parallel execution. In particular, we analyze the timeline diagram of figure 6.4.

The program that is being executed does not correspond to any particular computation, but a synthetic code for the sake of demonstration. The `ADDV` Codelet adds two vectors element by element, each of size 16 bytes. Memory is byte addressable. All registers described with the letter R are 16 bytes long. Both the memory controller and the L2 memory have multiple channels. Memory controller has a mechanism to detect potential hazards between reads and writes.



```
LD R1, 0
LD R2, 10
LD R3, 100
LD R4, 110
COD ADDV R5, R1, R3
COD ADDV R6, R2, R4
ST R5, 200
ST R6, 210
COD ADDV R7, R5, R6
ST R7, 300
COMMIT
```

Figure 6.4: Time diagram of the execution of a Codelet using a superscalar approach with 2 Compute Units. See the executing Assembly Codelet Program on the bottom right

Starting from left, the first column represents the instruction memory. Next, there is the SU in the second column. The SU fetch instructions from the Instruction Memory, and decodes them. If the instruction is ready for execution, it schedules them to the appropriate unit. In green blocks we show the decoding and scheduling time. If an instruction is of type `MEMORY`, it is scheduled to the memory controller. If an instruction is of type `EXECUTE`, it represents a Codelets, thus it is scheduled to a CUs. The diagram shows two CUs, but a system may be consistent of more. The Memory controller communicates with L2 Memory. It also handles potential hazards when comunicating with memory. In the SU column, the red blocks represents stalls in the pipeline. We have extended some times to stress the stalling, but they would not necessarily represent the reality of a system. The light red block extending multiple columns contain the reason for the stall and the unit that the SU is waiting on before continuing execution.

This process is currently emulated in SCMUlate. The runtime is capable of discovering potential hazard free instructions that can be scheduled in parallel. It also stalls when a dependency may not be satisfied. As in the example case, registers are manually renamed to avoid anti- and output-dependencies. In the future Chapter 7 we show some real examples and their execution traces.

6.4 Profiling execution code

As previously mentioned, the SCMUlate framework include some tracing mechanism along with the runtime. This mechanism allows to obtain traces of the different elements that compose the SCM machine. It is also designed with hardware counters in mind.

In order to obtain the tracing data after execution, SCMUlate dumps the tracing information into a JSON formatted file. Following, an offline python script allows for visualization of data and better analysis. The current solution may not be ideal for large runs due to the large size of the dump JSON file, and the limitations of the python visualization. But for the time being, it provided a fast deployment of a tracing mechanism for the purpose of development. Other solutions are being studied.

Tracing is achieved by using high precision timers. At the beginning of the runtime, and right after initialization, a global clock is initialized. There is a vector for each element in the SCMUlate (i.e. SU, CU, and runtime). At a given point in the runtime, a new event is added to the vector containing an identifier of the event, and the current clock time. When dumping the events into the JSON file, the numbers are translated to other base time.

Currently, there is support for three type of timers: 1) A system timer for runtime events. 2) an SU timer for SU related events, and 3) a CUMEM timer for cu and memory interface related events. When multiple CUs exist, then there are multiple CUMEM timers. For system events we currently support two: beginning and end of runtime. For SU events we store: beginning and end of the SU unit, fetch an instruction, schedule (i.e. dispatch) an instruction, execute a control or basic arithmetic instruction, and enter in idle mode (e.g. stalling). Similarly, the CUMEM has the beginning and end events, an event when a Codelet starts executing, an event when a memory instruction starts executing, and an even when it goes into idle mode (e.g. waiting for work).

The final JSON file has a structure similar to the example in Listing 6.4. We are only showing a few examples of events and timers. The first timer (line 2) shows the system timer. Line 11 is the beginning of the SU timer. Finally, Lines 19 and 30 are the beginning of different CU and Memory timers. Inside the counter type indicates the different three types. Finally, the list of events containing an event type and a value as previously described.

```
{
    "SCM_MACHINE": {
        "counter type": "0",
        "events": [
          {
              "type": "0",
              "value": 0.000000
          }, ...
        ]
    },
    "SU_0": {
        "counter type": "1",
        "events": [
          {
            "type": "0",
            "value": 0.001469
          }, ...
    },
    "CUMEM_1": {
       "counter type": "3",
       "events": [
          {
             "type": "0",
             "value": 0.001466
          }, ...
        ]
    },
    "CUMEM_2": {
       "counter type": "3",
       "events": [
          {
             "type": "0",
             "value": 0.001477
          }, ...
        ]
    }, ...
}
```

Listing 6.4: JSON file structure of the tracing mechanism of SCMUlate

After obtaining the JSON file, it is processed by a python script that uses the Plotly library [85] to create an interactive trace. The output of the visualization tool is shown in Figure 6.5. As seen in the figure, it is possible to see the trace of the

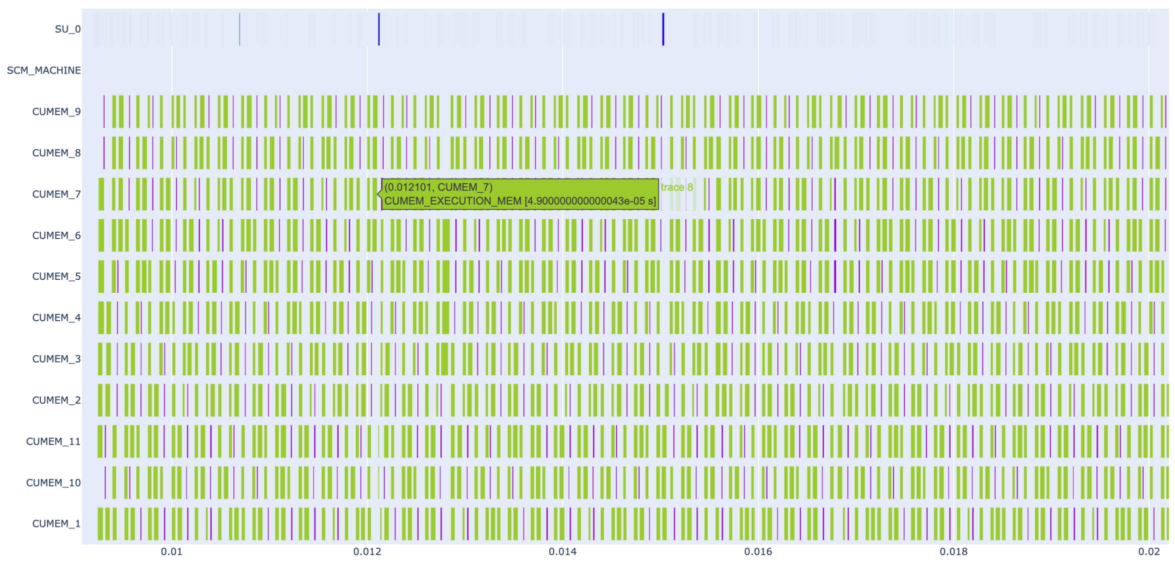


Figure 6.5: Interactive trace visualization example

different timers. Furthermore, it allows to visualize additional information of each of the elements shown in the trace. As we improve the tool and tracing capabilities, other information such as hardware counters are expected to be displayed in this plot.

Chapter 7
EVALUATION

Having explained the implementation of the SCMUlate framework in Chapter 6, this work evaluates the Sequential Codelet Model as well as the SuperCodelet architecture. A commodity Intel chip with two different architectures is used: the Intel gen9 GPU and the Intel x86 CPUs. This chapter describes evaluation results of the SCMUlate emulator.

7.1 Evaluation methodology

The SCMUlate emulator has two main purposes. First, to demonstrate that it is possible to achieve parallel execution of code through the use of the Sequential Codelet Model in heterogeneous systems. Second, to obtain early metrics of a system that implements the SCM, as well as understand the programming challenges and strategies when using the SCM model. The SCMUlate emulator is intended to run on commodity hardware, but it is also meant to faithfully mimic the behavior of the different parts of the system. The ideal outcome of the SCMUlate emulator is not to be faster than already existing parallel frameworks such as OpenMP or highly optimized libraries. Instead, it is meant to demonstrate the feasibility of the SCM machine as well as explore design parameters of the SuperCodelet architecture.

The proposed methodology intend to answer the following questions:

- Is it possible to separate program semantics from execution mechanisms by using a well define program execution model? Consequently, Is it possible to separate software and hardware evolution through a well established interface, similar to the Instruction Set Architecture?
- Can sequentially describe code be parallelize by using dataflow inspired execution engines? If so, what are the potential improvements?

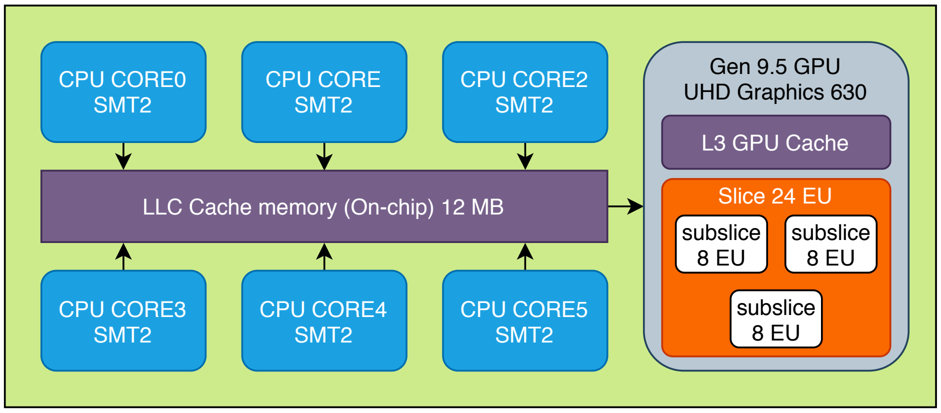


Figure 7.1: Architecture block diagram of the Intel Core i7-8700K containing the Intel Gen9 integrated GPU

- Is it possible to build an SCM machine around a heterogeneous system, allowing to utilize the same execution model for both GPU, CPU and potentially other architectures?
- What are the major challenges when translating already existing code into SCM semantics? How could we face these challenges?

Two different microbenchmarks are used for evaluation purposes: Vector addition, Matrix and multiplication.

7.2 Testbed

The ideal experimental test-bed is a system that contain CPUs and programmables GPUs in the same chip. This restriction is important to explore heterogeneous execution with CPU cores and GPU cores under the same program execution model. By having the two units in the same chip, the communication latency between the two architectures is determined by the on-chip memory (e.g. L3 cache). This should resulting in lower communication latency between CPU and GPU. The alternative would be using discrete GPUs. However, to communicate a CPU and a GPU core one must go through an external interconnect. Such approach would worsen the overhead of the emulation of the SCM memory hierarchy for the L1 level currently being emulated. Discrete GPUs on different dies could be used in L2 system emulations.

There exist several commercially available processors that contain CPU cores and GPU cores in the same die. Intel architectures often feature an integrated GPU that is fully programmable. Most recent generation of processors feature the Gen9 GPU architecture [83] coupled with CPU cores featuring Simultaneous MultiThreading technology (SMT). The L3 memory is shared between CPUs and the GPU cores. Therefore, L3 Cache can be seen as the Level 1 (i.e. SCM L1) register file when emulating the SuperCodelet architecture, connecting both GPU and CPUs under the same level.

This work uses an Intel Core i7-8700k processor. Figure 7.1 shows a simplified architecture diagram. There are 6 CPU cores, each with HyperThreading technology (i.e. SMT with 2 threads), for a total of 12 hardware threads. Each thread has a base frequency of 3.7 GHz with potential overcloking through Turbo Boost technology. Additionally, the Intel Core i7-8700k contains an Intel UHD Graphics 630 Gen 9.5 architecture integrated GPU. This GPU comes with 1 Slice containing 3 sub-slices, each with 8 execution units (EU), for a total of 24 EU. An slice is the equivalent of a GPU Core (e.g. Streaming Multiprocessor in NVIDIA GPUs, or an SIMD-Core or Data-parallel processor for the AMD GCN architecture). Each execution unit has a pool of threads with an independent register file and state. During a clock cycle, the EU schedules 4 threads simultaneously regardless of the kernel. The scheduled threads vary from cycle to cycle, selecting 4 different threads on each cycle. Finally, each thread uses SIMD-4 (32 bits) instructions that are assigned to one of two FPU (Floating point ALUs) capable of executing 1 addition and 1 multiplication simultaneously. Therefore, each EU can execute up to sixteen 32 bit instructions per cycle $(2FPUxSIMD4x(1add + 1multiply))$.

The Intel i7-8700k processor is hosted in a Dell Model Precision 3630 desktop tower. The motherboard contains a single socket configuration. Additionally, it is equipped with 32 GB of DRAM distributed in two 16 GB DDR4 DIMMs, each running at 2666 MHz.

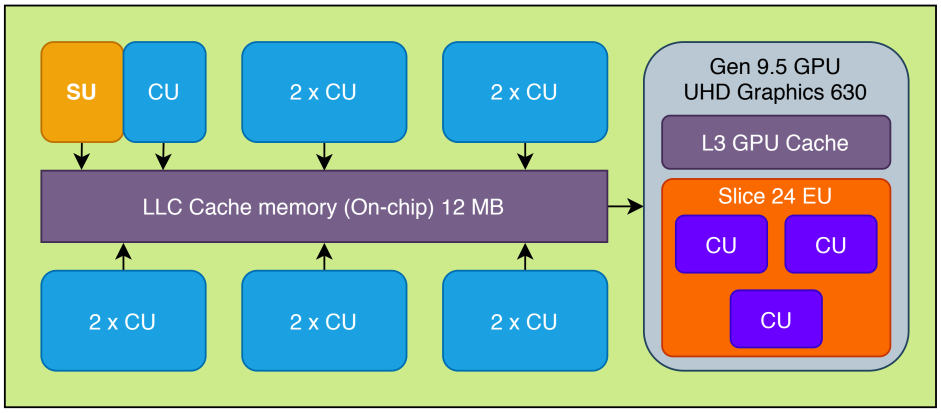


Figure 7.2: Architecture block diagram of the Intel Core i7-8700K containing the Intel Gen9 integrated GPU

7.3 Sequential Codelet Abstract Machine mapping

The first step of the runtime is to map the running system resources to the Sequential Codelet Abstract Machine by creating and pinning a software thread per hardware thread in the system, and assigning it a role, as described in Chapter 6. One single hardware thread is assigned as the Scheduling Unit. This thread does not participate in computation of Codelets or memory operations, but it is in charge of fetching L1 level instructions, as well as executing Control flow and arithmetic instructions (As seen in table 6.1). The rest of the threads are assigned Computation and Memory Unit roles simultaneously.

Figure 7.2 shows a modified version of the Intel's Core i7 diagram presented before. In this figure a single hardware thread is used as SU, while the rest are CUs.

The emulation strategy in SCMUlate provides major limitations that jeopardize the performance of the runtime and the application. First, the SU is mapped to a full hardware thread that is not used for computation of Codelets. This results in a sub-utilization of hardware resources. Second, the emulation of memory operations require extra data movements that are costly. Furthermore, Memory units emulation uses a whole hardware thread when executing memory instructions. Thus, wasting compute resources. Finally, commodity hardware architectures feature ILP optimizations that rely on the original von Neumann abstractions which are critical to achieve top performance in these systems. Architecture optimizations such as data prefetching and branch prediction are affected by the runtime, and reduce the overall efficiency of the application. Consequently, making a one-to-one comparison with other parallel programming models that rely on the original Von Neumann abstraction is challenging.

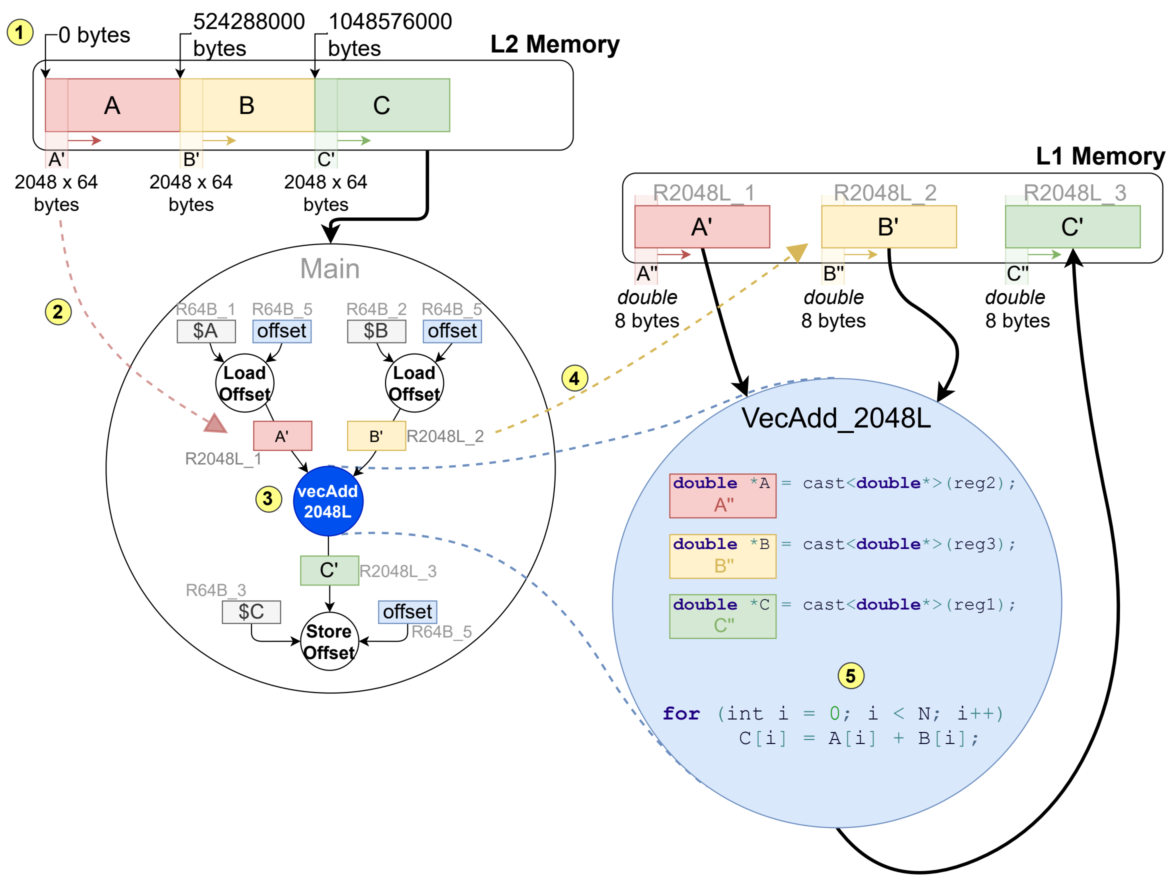
7.3.1 Register File:

On the Intel(R) Core(TM) i7-8700K system, there is a LLC cache of 12MB. The cache line size is 64 bytes long, and it is used as unit of measure for defining the register sizes. Therefore, the emulation of the register file through the LLC, as described in Chapter 6, distributes the memory as follows: 160 registers of 64bits, 140 registers of 1 LINE, 100 registers of 8 LINE, 100 registers of 16 LINE, 60 registers of 256 LINE, 60 registers of 512 LINE, 60 registers of 1024 LINE, and 40 registers of 2048 LINE. Where LINE corresponds to the size of the cache line. This arrangement results in a total register file size of 11.72 MB, corresponding to almost all the LLC Cache. These values have been selected in an almost completely aleatory manner. They reflect the need to use different register sizes, while providing an starting point for exploring what the appropriate register size is for the upper levels of the SCM architecture.

7.4 Example 1: Vector Addition

Vector addition is implemented as a gateway test for initial evaluation and explanation of SCMUlate. Vector addition is an embarrassingly parallel problem that allows for a simple implementation that illustrates the intentions of this work. The operation A[] = B[] + C[] is perform. A, B and C are vectors of type double, each of size $4000 * 2048 * 64 bytes = 524.2 MB$ for a total of 65536000 elements per vector.

Figure 7.3 represent the overall memory and compute structure at different levels of the SCM machine in the context of the vector addition program. Numbers in yellow are used to match the description in the rest of this section. There are three levels shown in this picture, L0 through L2. Recalling the explanation of Chapter 6, the reader should remember that the SCMUlate framework: 1) requires a level L2 memory



```
double *A = cast<double*>(reg2);
double *B = cast<double*>(reg3);
double *C = cast<double*>(reg1);

for (int i = 0; i < N; i++)
    C[i] = A[i] + B[i];
```

Figure 7.3: Structure of memory and computation for Vector Addition using SCM running on SCMUlate. 1) L2 memory contains the original 3 vectors (A,B,C) in a single flat memory allocation. 2) load and store operations of L1 fetch part of the whole array, according to the L1 register sizes (e.g. 2048x64 bytes) (A', B' and C'). 3) L1 Codelets use these registers to perform computation. They are evaluated at L0. 4) Each register in L1 belongs to L1 memory space. Then L0 computation access these registers through memory operations. Each read/write access as much as L0 register size (A", B" and C"). 5) Actual computation performed at L0

.

region that contains the vectors A, B and C; 2) emulates the execution of a single `Main` Codelet of level L2, described in L1 assembly and interpreted by the runtime; and 3) allows to define L1 Codelets as L0 assembly code for execution in commodity hardware.

Next to the yellow circle with the number 1 there is L2 Memory, allocated in DRAM at the beginning of the SCMUlate framework. L2 memory contains the whole program memory. Right under L2 memory, a circle represents the `Main` Codelet of the Vector Addition SCM program. The connection between the `Main` Codelet and the L2 Memory implies that this Codelet uses the L2 memory as its operand. Within level 1, memory addresses are calculated as offsets to the beginning of the L2 memory region, resolving the translation between the program memory (as managed by the operating system running the runtime) and L2 memory. For this example, L2 memory stores the vectors to be added, A and B, as well as the resulting vector C. To the right of L2 memory, there is L1 memory (i.e. L1 register file seen as memory from L0).

The code shown in Listing 7.1 contains the L1 assembly Codelet program that defines the `Main` Codelet. This code is interpreted and executed by the SCMUlate runtime, as explained in Chapter 6. Lines 1 through 3 represent the offset locations for each vector inside the L2 memory. Registers `R64B_1`, `R64B_2`, and `R64B_3` contain the location of A, B, and C respectively. Figure 7.3 shows these locations with arrows pointing to L2 memory. Lines 4 through 6 define registers `R64B_4`, `R64B_5`, `R64B_6` containing the iteration variable, the offset variable, and the total number of iterations respectively. This program fixes the number of iterations to 4000, defined by the size of vectors A, B and C. Following, lines 8 through 16 contain the iteration body. Line 9 corresponds to the control operation of the loop, jumping out of it to line 17 (i.e. PC + 8) when register `R64B_4` (iteration variable) is equal to the register`R63B_6` (limit) that contains the total number of iterations (i.e. 4000). Line 14 increases the iteration variable, and Line 15 increases the offset that points to the chunk to be processed in the next iteration.

```
1  LDIMM R64B_1, 0; //Vect A Base Addr
2  LDIMM R64B_2, 524288000; // B Base Addr
3  LDIMM R64B_3, 1048576000; // C Base Addr
4  LDIMM R64B_4, 0; // iteration variable
5  LDIMM R64B_5, 0; // offset
6  LDIMM R64B_6, 4000; // num of iterations
7
8  loop:
9    BREQ R64B_4, R64B_6, 8;
10   LDOFF R2048L_1, R64B_1, R64B_5;
11   LDOFF R2048L_2, R64B_2, R64B_5;
12   COD vecAdd_2048L R2048L_3, R2048L_1, R2048L_2;
13   STOFF R2048L_3, R64B_3, R64B_5;
14   ADD R64B_4, R64B_4, 1;
15   ADD R64B_5, R64B_5, 131072;
16   JMPLBL loop;
17 COMMIT;
```

Listing 7.1: `Main` Codelet for the Vector Addition program

The main computation of this program happens in lines 10 through 13. In each iteration, a chunk of A and B are loaded into the L1 register file, then the Codelet `VecAdd_2048L` is executed, and the result is also placed in an L1 register. Following, the value is stored back from the result register into L2 memory. This process is depicted in figure 7.3 inside the `Main` Codelet. Load and store operations receive an address, an offset and a register. The red dotted arrow next to the yellow circle with the number 2 shows the chunk of the vector A and its correspondence to the register (labeled A'). The load operation of A copies the values from a chunk of A in L2 memory into the register in L1 that contains A'. These registers are of size `R2048L_x`, equivalent to $2048 \times 64 = 131072$ *bytes*. This is also the value that is used to increment the offset variable in Line 15. Notice that the Codelet, depicted next to the yellow circle with the number 3, operates directly from and to the registers (i.e. the L1 register file). Also, notice that the representation inside the `Main` Codelet is a dataflow graph, while in the SCM program is a sequential description of this graph.

The description of Codelet `vecAdd_2048L` is defined in the assembly ISA language of the low level commodity hardware architecture (i.e. x86 in our testbed). Fortunately, current compilers are able to translate C++ program to low level ISA. Hence it is possible to use the approach described in Section 6.2 to defile L1 Codelets in L0 assembly. The yellow circle with the number 5 in Figure 7.3 shows the same description in the context of the whole SCM infrastructure. The blue circle on the right labeled `VecAdd_2048` corresponds to a zoom in into the `VecAdd_2048` Codelet used in the `Main` Codelet on the left.

Listing 7.2 shows the definition and implementation of the `vecAdd_2048L` Codelet in C++. Line 1 contains the definition of the Codelet with the `DEFINE_CODELET()` macro. This Codelet has 3 operands. Operand 1 corresponds to result vector C, and it is a write operand (i.e. `OP_IO::OP1_WR`). Operands 2 and 3 correspond to A and B respectively, and they are read operands (i.e. `OP_IO::OP2_RD | OP_IO::OP3_RD`). The implementation of the Codelet starts in line 3 by using the macro `IMPLEMENT_CODELET()`. Lines 6 through 10 cast the operand registers into arrays of type double. This is needed since, for the SCMUlate Runtime uses generic types `unsigned char *` for registers, while for the user these registers represent data in the format that makes sense to the application (i.e. `double *` in this case). Inside of the Codelet, A, B and C are vectors as large as registers `R2048L_x`. Therefore, each Codelet operates over $131072/8 = 16384$ elements where 8 is the size of a double. Finally, lines 12 and 13 is the actual computation that adds the different vector elements, iterating over each element of the vectors A, B and C. `numElements` is a constant value equal to 16384.

In Figure 7.3 the yellow circle with number 4 shows the relationship between the register used inside the `Main` Codelet, and the L1 register file. The bold arrows connecting the L1 Memory to the `VecAdd_2048L` Codelet represent the registers being used as operands of the Codelet. Notice that a single element inside of the for loop in the Codelet corresponds to a chunk of the L1 register file. This demonstrates the hierarchical memory organization of the SCM model.

```
DEFINE_CODELET(vecAdd_2048L, 3, scm::OP_IO::OP1_WR | scm::
    OP_IO::OP2_RD | scm::OP_IO::OP3_RD);

IMPLEMENT_CODELET(vecAdd_2048L,
  // Obtaining the parameters
  // Getting register 1
  double *A = this->getParams().getParamAs<double*>(1);
  // Getting register 2
  double *B = this->getParams().getParamAs<double*>(2);
  // Getting register 3
  double *C = this->getParams().getParamAs<double*>(3);

  for (uint64_t i = 0; i < numElements; i++)
    C[i] = A[i] + B[i];
);
```

Listing 7.2: L0 Codelet VecAdd_2048L implementation

In general, Figure 7.3 uses yellow circles with numbers 2 and 4 to follow this hierarchical organization. L1 registers are considerably smaller than arrays A, B and C. Therefore, a single load operation of L1 covers an small segment of the whole vector. These segments are referred to as A', B' and C' respectively in the Figure. The loop in the L1 program in Listing 7.1 allows for moving the chunk along each vector. Likewise, the `vecAdd_2048L` Codelet uses these chunks as operands in the form of registers of the L1 register file. This L1 register file is seen as memory from the perspective of L0. A single register in L0 can only hold a few elements of the vectors A', B' and C'. For example in the testbed architecture, x86 registers will hold a single element of the vector (assuming no SIMD extensions). In the Figure A", B" and C" represent these elements, while the loop inside the Codelet `vecAdd_2048L` allows to operate over the whole sub array A', B' and C'.

```
#define REPS 4000
#define REG2048L 64*2048
// Size of each vector 64*2048*4000/8 = 65536000 elements
#define SIZE_VECT ((REG2048L*REPS)/sizeof(double))

struct  __attribute__((packed)) packed_l2_mem {
  double A[SIZE_VECT];
  double B[SIZE_VECT];
  double C[SIZE_VECT];
};

int main () {
    packed_l2_mem * memory = new packed_l2_mem;
    // Initialize A, B, and C  ...
    scm::scm_machine * myMachine;
    scm::scm_machine myMachine("vectorAdd.scm",
                               (l2_memory_t) memory,
                               ILP_MODE);
    myMachine->run();
    // Verification and cleaning ...
    return 0;
}
```

Listing 7.3: SCMUlate program for Vector Addition. Runtime creation and `Main` Codelet creation.

Finally, in order to show the whole picture of the SCMUlate framework, it is necessary to illustrate the structure of the main C++ program that instantiates the runtime and creates the `Main` Codelet. Listing 7.3 shows a simplified version of the `main()` function in C++ for the Vector Addition program. In this listing, Lines 1 through 4 define three macros created to ease understanding of the program: `REPS`, `REG2048L` and `SIZE_VECT`. `REPS` corresponds to the number of iterations performed inside of the `Main` Codelet, as seen in listing 7.1 and its explanation. `REG2048L` corresponds to the size of registers in L1. `SIZE_VECT` determines the overall number of elements of A, B, and C. Following, Lines 6 through 10 define a `struct` that packs A, B and C into a consecutive memory region. The `__attribute__((packed))` is used to make sure there is no padding between vectors. L2 memory is then instantiated inside of the `main()` function in line 13. After initialization of A, B, and C (hidden in the Listing), the SCMUlate runtime is created in line 16. The runtime constructor receives 3 parameters: 1) the `.scm` file that contains the assembly code of the `Main` Codelet, 2) a pointer to L2 memory, and 3) the Codelet Level Parallelism mode (i.e. SEQUENTIAL, SUPERSCALAR, OOO as described in section 6.1.3.4). During construction of the runtime, the `.scm` file is interpreted and loaded into the instruction memory of the SCM machine. Finally, line 19 calls the runtime for execution, storing the result into the C vector stored in L2 memory. The result is check and the code clean (hidden in the listing), and the SCMUlate program finishes.

7.4.1 Vector Addition results

The Vector addition problem is executed using the different Codelet Level Parallelism mode supported by the SCMUlate framework, as described in 6.1.3.4. Furthermore, scalability with the number of Computational Units is studied. An OpenMP implementation with Vector Addition is used to compare the results and estimate the SCMUlate overhead. Furthermore, we manually apply the Loop Unrolling and Register Scheduling compiler optimization techniques to demonstrate that it is possible to take advantage of already existing Compiler techniques when used with the SCM Program Execution Model.

```
LDIMM R64B_1, 0; // Loading base address A
LDIMM R64B_2, 524288000; // Loading base address B
LDIMM R64B_3, 1048576000; // Loading base address C

LDIMM R64B_4, 0; // For iteration variable
LDIMM R64B_5, 0; // For offset 1
LDIMM R64B_6, 131072; // For offset 2
LDIMM R64B_7, 262144; // For offset 3
LDIMM R64B_8, 393216; // For offset 4
LDIMM R64B_9, 4000; // For number of iterations

loop:
  BREQ R64B_4, R64B_9, 23;
  LDOFF R2048L_1, R64B_1, R64B_5;
  LDOFF R2048L_2, R64B_2, R64B_5;
  LDOFF R2048L_4, R64B_1, R64B_6;
  LDOFF R2048L_5, R64B_2, R64B_6;
  LDOFF R2048L_7, R64B_1, R64B_7;
  LDOFF R2048L_8, R64B_2, R64B_7;
  LDOFF R2048L_10, R64B_1, R64B_8;
  LDOFF R2048L_11, R64B_2, R64B_8;
  COD vecAdd_2048L R2048L_3, R2048L_1, R2048L_2;
  COD vecAdd_2048L R2048L_6, R2048L_4, R2048L_5;
  COD vecAdd_2048L R2048L_9, R2048L_7, R2048L_8;
  COD vecAdd_2048L R2048L_12, R2048L_10, R2048L_11;
  STOFF R2048L_3, R64B_3, R64B_5;
  STOFF R2048L_6, R64B_3, R64B_6;
  STOFF R2048L_9, R64B_3, R64B_7;
  STOFF R2048L_12, R64B_3, R64B_8;
  ADD R64B_4, R64B_4, 4;
  ADD R64B_5, R64B_5, 524288;
  ADD R64B_6, R64B_6, 524288;
  ADD R64B_7, R64B_7, 524288;
  ADD R64B_8, R64B_8, 524288;
  JMPLBL loop;

COMMIT;
```

Listing 7.4: Loop unrolling of size 4 for Vector addition. : A

7.4.1.1 Manually applying optimization techniques: Loop Unrolling and Register Scheduling

Depending on the Codelet Level Parallelism used to implement the Sequential Codelet Model, it may or may not be possible to remove anti- and output dependencies. This is the case, for example, of the Superscalar implementation explained in section 6.1.3.4. On the other hand, the properties defined for Codelets in conjunction with the hierarchical organization of memory in the SCM model, provides a bounded execution time for Codelet instructions. This property is critical for SCM as it allows for the static analysis of code usually found in compiler techniques used in sequential execution of programs. Therefore, it is possible to imagine an SCM compiler that takes as input code similar to the one found in Listing 7.1, together with information for each Codelet (e.g. execution time, memory requirements, computational complexity, or other metadata), and apply already existing optimizations to improve the execution of code.

To demonstrate this principle, two well known compiler techniques are applied to the SCM assembly code of Vector Addition: 1) Loop Unrolling and 2) register renaming. Listing 7.4 shows the result of applying unroll 4 operation on the code of Listing 7.1 and register scheduling. Ultimately, this code removes false dependencies that existed in the original Code. As a result, a simplified implementation of the SCM machine that does not feature a full out of order engine (e.g. just superscalar) could still parallelize this code.

7.4.1.2 Sequential execution

The `ILP_MODE::SEQUENTIAL` execution mode uses a single Compute Unit (CU) and it does not attempt to exploit Codelet Level Parallelism. Figure 7.4 shows a segment of the trace when executing vector addition running on SCMUlate using Sequential mode. The horizontal axis represents time, while each of the bars represent a different unit in the SCM machine. For Sequential, 1 bar represents the SU and another bar the CU. The total execution time of the SCM program is shown on the title, as well as the time the execution starts and end.

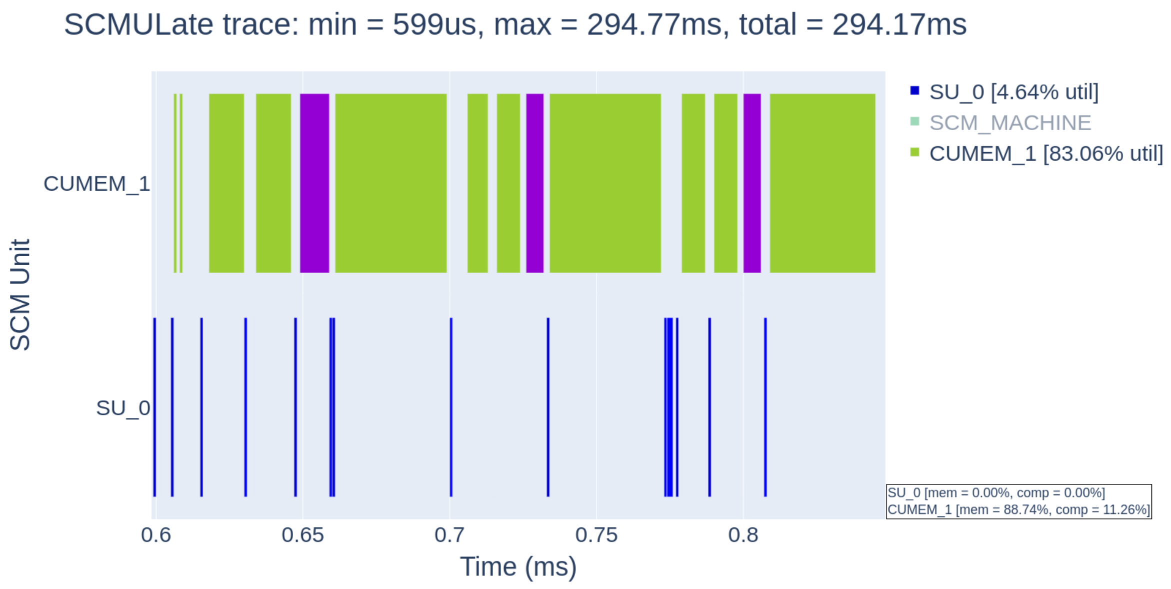


Figure 7.4: Vector Addition execution trace for the sequential mode on the SCMUlate emulator. 3 loop iterations on 1 CU

The bottom bar labeled `SU_0` corresponds to the trace of the Scheduler Unit. The blue segments represent the scheduler processing time. The SU also executes basic arithmetic operations and control flow instructions, included in the blue bars. On the top right label, the percentage of time the SU was used for scheduling can be seen inside the brackets. For this particular program and execution mode the `SU_0` use is only 4.64% of the total execution time. The rest of the time, this unit was idle waiting for work. This behavior is expected and it supports the idea that an Scheduler Unit does not have to be a really optimized core, but it could be a simplified architecture or circuit logic. The `CUMEM_1` bar shows the compute unit which executes Codelets (in purple) and memory instructions (in green). The utilization for this core is 83.06% (as presented in the label on the top right).The remaining 16.94 % of the time, the CU was waiting for work from the SU, as well as synchronizing with the SU.

The zoomed-in segment presented in Figure 7.4 focuses on the beginning of the execution of the program. To understand this trace, it is possible to compare it to Listing 7.1. The CU executes memory instructions and Codelets one at a time as shown in the trace. At the beginning of the program, several `LDIMM` instructions are issued.

These are seen as thin bars at the beginning of the CUMEM_1 bar. Following there is a loop. First, two `LDOFF` instructions are issued, followed by a Codelet `vecAdd_2048L` and a `STOFF` operation. This loop is seen as a repetitive patter of 2 green bars, followed by a purple bar, followed by another larger green bar. This trace shows 3 iterations of the program.

Figure 7.4 also shows a break down of the CU memory vs compute execution time in the annotation on the bottom right. Excluding the time the CU was idle, 88.7% of the real execution time was spend in memory operations, while 11.26% of the time was spend in compute operations.

7.4.1.3 Superscalar execution

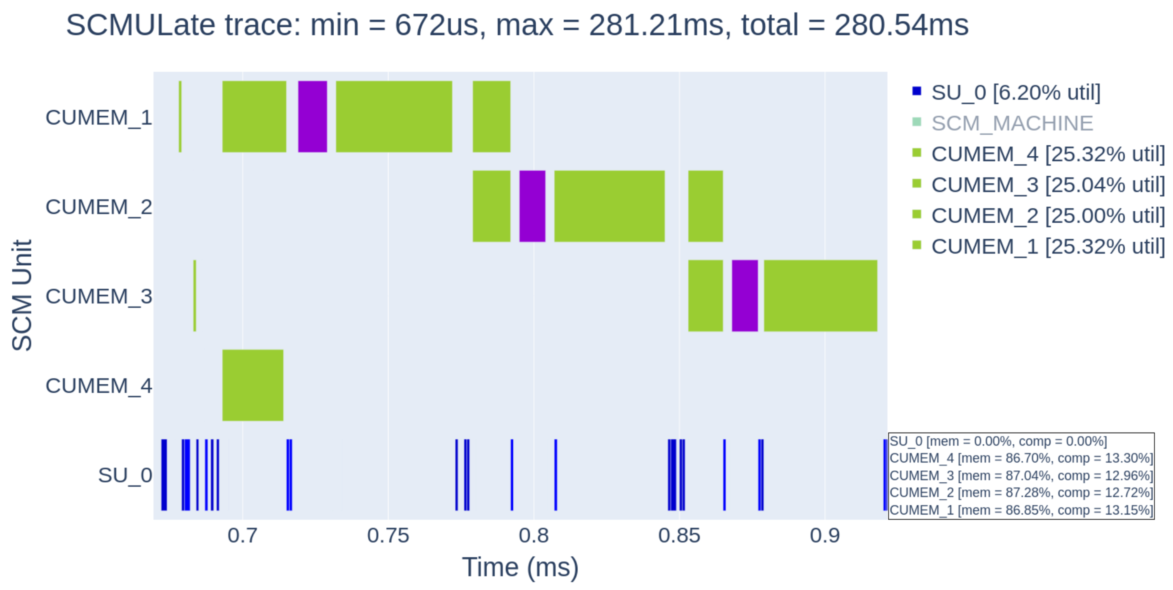


Figure 7.5: Vector Addition execution trace for the Superscalar Codelet Level Parallelism mode on the SCMUlate emulator. 3 Loop iterations on 4 CUs.

Figure 7.5 shows the execution trace for the `ILP_MODE::SUPERSCALAR` parallel execution of the same program. This mode enables Codelet Level Parallelism in 4 different cores. In this execution mode, the bottom bar represents the `SU_0`, as explained in the sequential execution. The top 4 bars represent each of the CUs. Green represent memory operations and purple Codelet operations. We observe an slight increase in the overall SU utilization, raising to 6.20% of the execution time. The total aggregated execution time in Sequential is 13.6 ms, while in the SuperScalar version is 17.39 ms.

Although the overall execution time of the machine is lower than the sequential execution, the speed up is only 4.8%. The trace shows how this execution lacks of parallelism. When analyzing the dependencies of Listing 7.1 it is possible to observe a write after read dependencies between iterations on registers `R2048L_1` and `R2048L_2`. Superscalar execution mode cannot resolve these dependencies, resulting in a stalls after every iteration. The utilization of each thread lowers considerably to about 25%, since CUs are most of the time waiting for work.

To overcome this limitation, the unrolling technique presented in Section 7.4.1.1 is applied. Figure 7.6 shows the execution trace for the 4-times unrolled version of vector addition of Listing 7.4. First, we observe a considerable improvement in the execution time of 241.2% with respect to the sequential execution. Second, the utilization of each unit goes up to about 60%, in comparison to 25% without unrolling. The relative SU utilization is also increased to 14.91 %. However, the total accumulated SU time is 18.18 ms, close to previous results. 6 CUs are used to highlight the gap between iterations caused by remaining dependencies, as seen in the trace between t=0.95ms and t=1 ms mainly affecting CUs 3 an 4.

Unrolling has removed some write after read dependencies and structural hazards presented in the original code. However, the unrolled version still contains dependencies between iterations. The number of instructions that can be scheduled is determined by the number of independent instructions between dependencies. As the number of CUs is increased, there will be a point where not enough instructions are available for execution, resulting in subutilization of resources. To allow for more parallelism, it is necessary to unroll even more loop iterations. Later in this chapter we show results for unroll 8.

On the other hand, it is possible to see that the ratio between compute and memory operations per CU changes slightly in comparison to the sequential version. The superscalar execution with loop unrolling shows an average 90% of the time spend in memory, and 10% in computation. As will be seen later in this chapter, unrolled execution has a considerable increase in the execution time of memory instructions, justifying the increased participation in the overall computation.

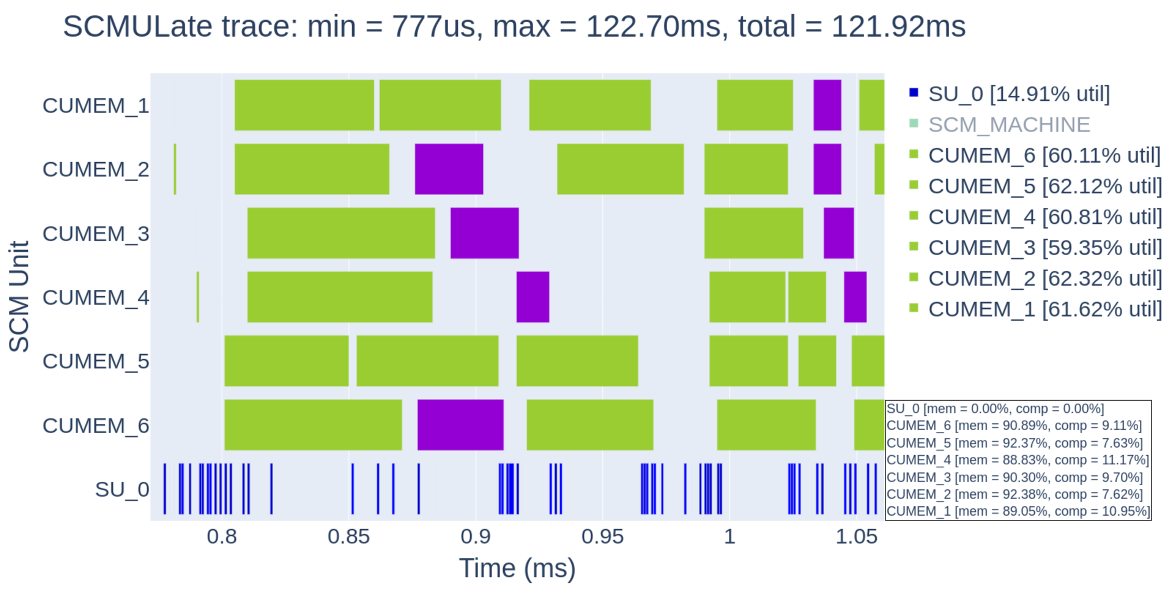


Figure 7.6: Vector Addition unrolled 4 times. Execution trace for the Superscalar Codelet Level Parallelism mode on the SCMUlate emulator. Zooming on at least 4 loop iterations.

7.4.1.4 Out of order execution

Out of order execution is the most advanced Codelet Level Parallelism architecture implemented in SCMUlate. This mode is capable of discovering and eliminating all output and anti dependencies within a window of instructions. Figure 7.7 shows the execution trace of the `ILP_MODE::OOO` (Out of Order) execution. A continuous overlapping of instructions is observable across all the different compute units. The total execution time is reduced to 88.15 ms, a speed up of 318.25% in comparison to the Sequential execution mode, and 138.30 % in comparison to the Superscalar execution mode.

SU utilization also increases considerably to 54.85%, an expected trend due to the additional analysis and resource management needed for dependency discovery.

The total accumulated Scheduling time is up to 48.35 ms. The overall utilization of the CUs is increased to about 68% in average while the memory to compute ration is back to about 88% for memory.

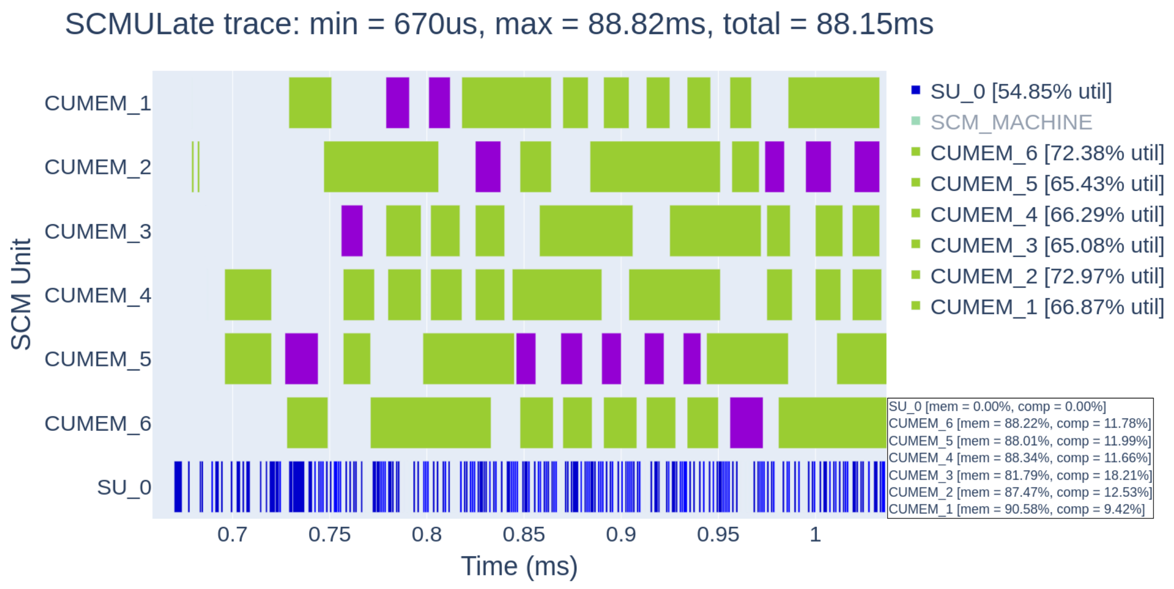


Figure 7.7: Vector Addition execution trace for the Out of Order Codelet Level Parallelism mode on the SCMUlate emulator. Zooming in to the initial loops.

7.4.1.5 Comparison between execution modes

It is clear that the SCM model yields to improvements in execution time. Figure 7.8 shows a summary of an experiment that runs the Vector Addition Code in all three modes. Out of Order execution and Superscalar execution modes use 5 CUs, while sequential uses a single CU. the orange bar represents the execution time after applying compilation techniques. although it is not the same code, it demonstrates the importance of the Program Execution Model that enables the use of traditional compiler techniques.

To better understand the scalability of the Sequential Codelet Model and the different Codelet Level Parallelism techniques, it is necessary to study the effect of the number of CUs in the execution. The use of vector addition is not arbitrary. This simple problem allows exposing the limits of the emulation approach or running system thanks

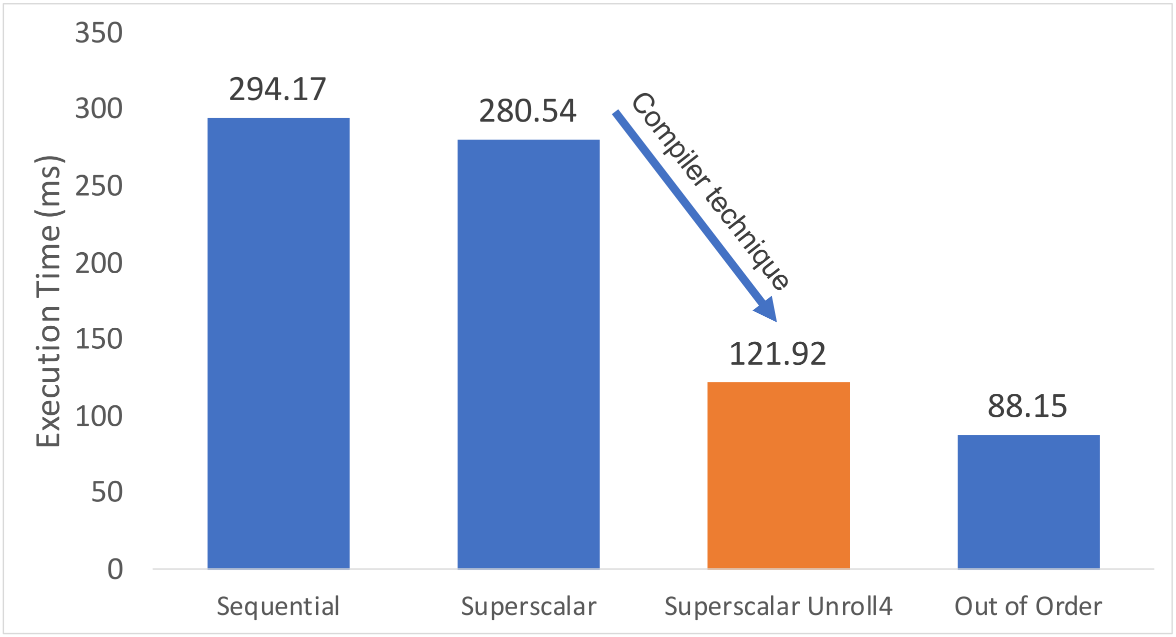


Figure 7.8: Execution time comparison for different execution modes. Superscalar and Out of order used 5 CUs

to its ability to saturate the system with extensive parallelism. Additionally, results of the SCMUlate runtime are compared against a simple OpenMP implementation. This baseline provides a comparison with state of the art execution models.

First, the execution time for the different program versions and available Codelet Level Parallelism techniques are obtained. Figure 7.9 shows the execution time of the OpenMP baseline, the Out of Order execution and the Superscalar version with no unrolling, and 2 and 8 times unrolling. As expected, Superscalar no unrolling is the slowest execution with little scalability, followed by unroll 4, unroll 8, Out of Order and OpenMP.

Following we compare scalability within each execution mode. This means, comparing the speed up between the sequential version of each execution mode and the execution with different number of CUs. Figure 7.10 shows the speed up progression as the number of threads grows. The Superscalar no unrolling version does not achieve any noticeable speed up, given its inability to exploit parallelism. Out of order execution achieves the greatest speed up in comparison to its sequential version. This may be due

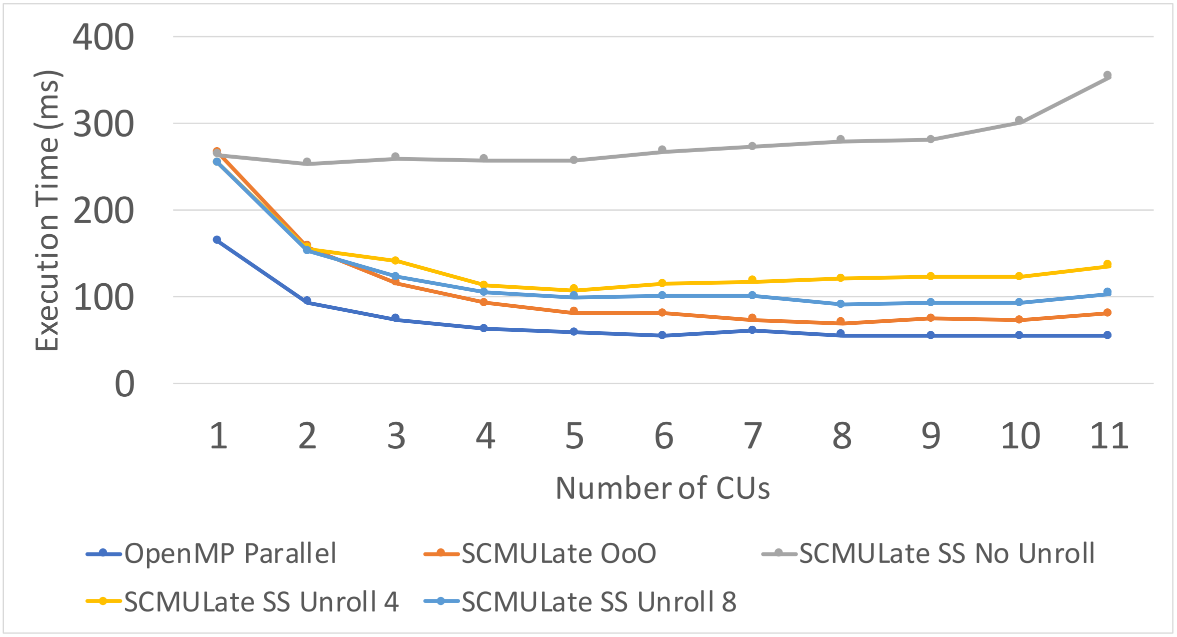


Figure 7.9: Vector Addition execution time for different number of CUs. The size of the vector is the same in all the cases. Lower is better

to the high overhead that the OoO engine has to pay to discover parallelism, making the sequential version slower than the parallel versions. The reader must be careful in comparing these figures as they all have different comparison baselines. But this figure gives a quantitative analysis on the ability to achieve better performance as the number of CUs increase within a given design.

The next figure is the comparison of the different approaches vs the OpenMP execution. OpenMP is used as an ideal execution time. It should still be possible to obtain better performance than the OpenMP version used in this comparison. However, as it has been previously mentioned, the intention is not to achieve the best performance, but to have a quantitative analysis of both SCMUlate and the SCM model, while having a comparison point with traditional execution models created for Von Neumann based architectures. Figure 7.11 shows how the Out of Order execution mode achieves up to 80 % of the execution time of OpenMP. Considering the overhead of the runtime, this result speaks well of the ability of the SCM model to achieve parallel execution of Code.

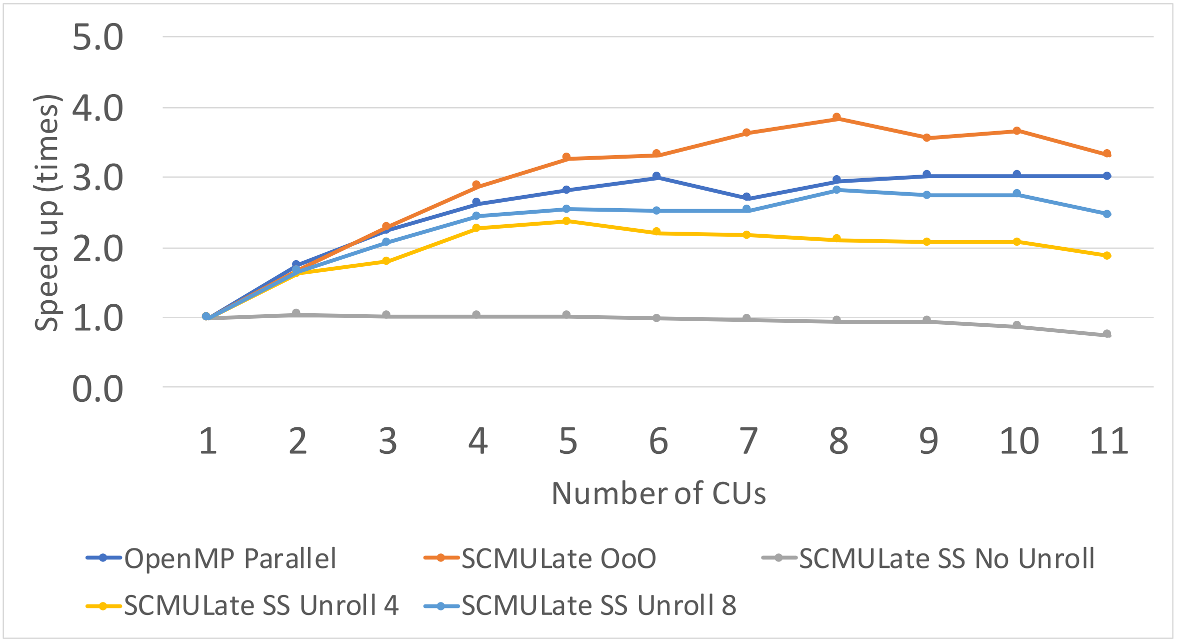


Figure 7.10: Speed up progress with the number of CUs as compared against the execution with 1 thread within each mode of operation.

Finally, a Codelet is expected to have a bounded execution time regardless of the workload of the system. If a Codelet has a bounded execution time, it is possible for a Compiler to optimize an SCM code based on static assumptions. The average execution time of the three major instructions across the different execution modes has been analyzed. These are `LDOFF`, `STOFF`, and the `vecAdd_2048L` Codelet.

Figure 7.12 Shows the slow down between the execution time of the instructions in sequential mode, and its degradation as the number of CUs increases and the execution mode is modified. This figure shows two trends. In comparison to the stand alone execution for 1 CU, instructions `VecAdd_2048L` and the `STOFF` have a performance degradation of up to 1.8 times when a heavy workload is present in the system. For `LDOFF` the trend is different for Superscalar execution and the out of order execution. The Superscalar execution shows a significant degradation in the execution time of Load instructions of up to 4.3 times. For Out of order execution the degradation is as bad as 1.4 across the different instructions.

The differences in the LDOFF can be explained by the increased number of

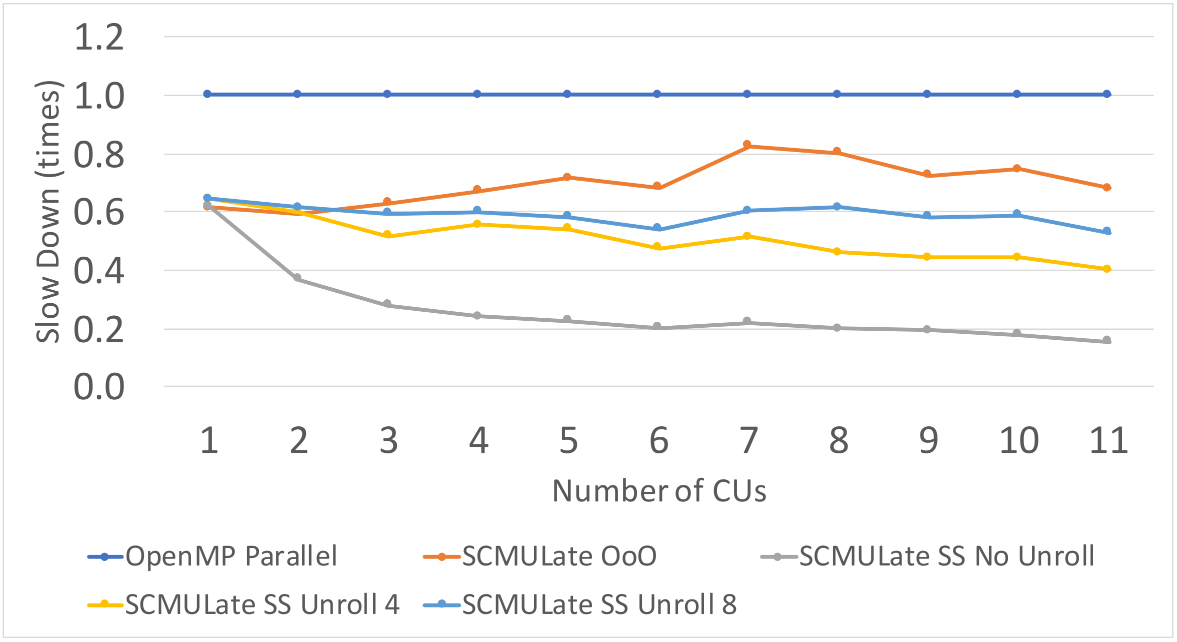


Figure 7.11: Slow down in comparison to a simple baseline using OpenMP. The higher the better.

consecutive loads in the unrolled versions. These consecutive loads create a bottleneck in the memory and increases the overall traffic of the memory interconnect. Modifying the order of load operations, interleaving some compute operations could solve these issues. Such changes can be applied by using compiler techniques similar to modulo scheduling [86]

7.4.2 Analysing the Vector Addition example

The Vector addition is a simple code, yet it shows the benefits and drawbacks of both the SCM model and the SCMUlate emulator. Furthermore, it provides guidelines on how to appropriately map the SuperCodelet architecture to a real architecture. First, it is worth clarifying that under a Von Neumann architecture, there will be better approaches to improve the overall performance of vector addition. Furthermore, Intel architectures are usually equipped with countless of features to improve performance of both sequential and parallel workloads. Elements such as data prefetching, speculative execution and improved cache coherency protocols are some of the features that one way

Execution Mode	Instruction	Average Execution Time
Out of Order	LDOFF	5.71 us
	STOFF	2.11 us
	vecAdd_2048L	4.68 us
Superscalar	LDOFF	4.42 us
	STOFF	1.97 us
	vecAdd_2048L	3.6 us
Superscalar U4	LDOFF	4.7 us
	STOFF	1.96 us
	vecAdd_2048L	3.6 us
Superscalar U8	LDOFF	5.2 us
	STOFF	2.02 us
	vecAdd_2048L	4.28 us

Table 7.1: Baseline execution time for the 3 most important instructions in Vector Addition, as seen in different execution modes when running with 1CU

or another affect the ability of SCMUlate to deliver the best results. Some drawbacks and important results are identified.

7.4.2.1 Evaluation drawbacks

The first difficulty that the SCMUlate runtime has is the overhead of the emulation of the L1 register file. Since the register file is actually a separate memory region that resides in DRAM, load and store operations from and to the register file result in additional data movements that considerably increase the overhead of this part of the system. These movements would not exist in a whole realization of the SCM model. This overhead results in the reduced performance as observe in the results that compared SCMUlate with other traditional computational models such as OpenMP. Furthermore, it is observable with the workload distribution between Compute Codelets and Memory instructions. All the different execution modes that were tested used more than 88 % of the executed time for memory operations. Furthermore, as can be seen in table ??, store memory operations in average took longer than the compute operation `vecAdd_2048L` and the Load operations. The store operation was, in average, 2.5 times slower than the Load operation, and 1.2 times slower than

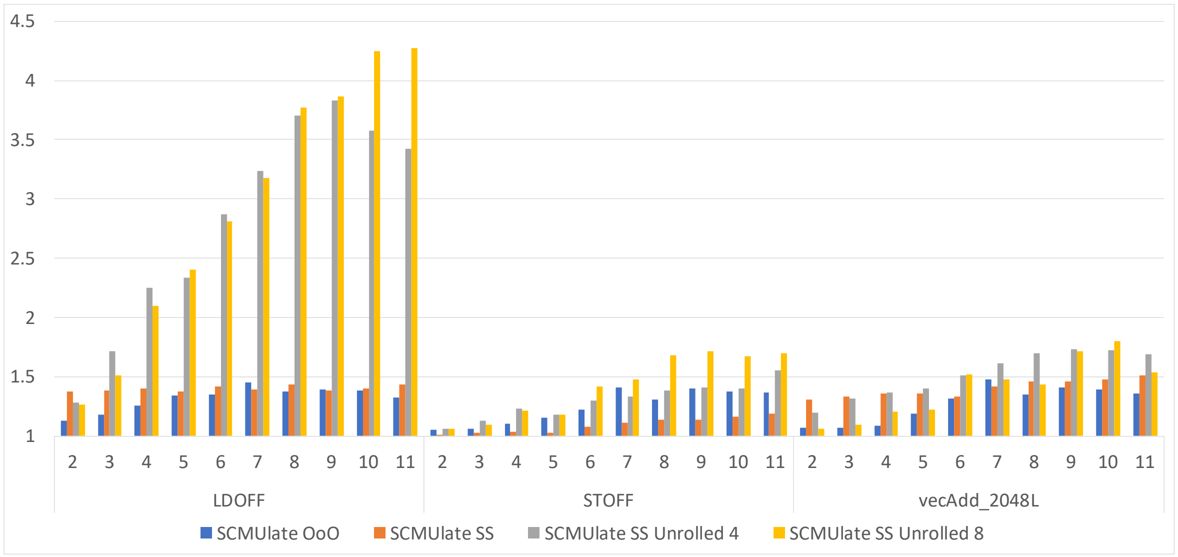


Figure 7.12: Average execution time of the three most important instructions in the execution of Vector Add in comparison to the number of CUs and the implementation approach. Lower is better

the compute operations. Although Vector addition is a memory intensive operation, featuring 2 Loads and 1 store per 1 addition, data prefetching and cache mechanisms in current architectures usually offset the cost of memory by guaranteeing data to be close to to the core when the Load operation occurs. On the other hand, our emulation of L1 register file increases the number of memory transfers and traffic in the network.

It is possible to overcome some of these drawbacks. For emulation purposes, memory bypassing transfers in the form of register renaming are currently being explored. The intuition is that for a load operation, it should be possible to move the reference for the register in DRAM, instead of performing a data movement (similar to a register renaming in L1). Second, for a hardware implementation, the CU should not be in charge of executing memory instructions. Instead, a special DMA engine should issue these operations. If done correctly, and without the burden of the cache mechanisms (e.g. scratchpad memory or more predictable cache protocols), the CU should be free to perform other computations while data is being moved. Under these considerations the cost of memory access should be considerably reduced. Finally, there will be other applications that are compute intensive, which would take advantage of data in the L1 Register file multiple times.

A second difficulty in the use of Vector addition is that vector addition is an embarrassingly parallel problem. This problem is usually best resolved in SIMD or SPMD execution modes. For the Sequential Codelet Model, a possible solution to this issue is to extend the SCMUlate runtime and ISA with instructions that are similar to SIMD extensions. Therefore, allowing Codelets to map to multiple CU units at the same time, and registers to be used by different CUs concurrently. An extension for SCM can be done to match SPMD execution models, but this is currently outside the scope of the project.

7.4.2.2 Important results

Despite the aforementioned drawbacks, there are important and positive observations that can be drawn from these experiments. First, the early evaluation in SCMUlate shows that it is possible to observe performance benefits from Codelet Level Parallelism through the extension of ILP-inspired optimizations that heavily borrow from dataflow models of computation. Results show implicit parallel execution of code with up to 3.8x improvements in comparison to the sequential execution of the same code in SCMUlate.

Furthermore, the SCMUlate shows that it is possible to separate memory from compute operations. Allowing the compute operations to achieve high performance and utilization of the compute resources. Efficiency can be improved by allowing the user to program the behavior of memory operations. One of the principles that the Sequential Codelet Model borrowed from the original Codelet Model is that Codelets must be atomically scheduled and non preemptive. This is when it is possible to guarantee that all the required resources for the execution of a Codelet are available before the Codelet can start executing. The control flow organization, together with the separation of Compute and Memory in the Sequential Codelet Model makes it possible for Codelets to take full advantage of the CPU resources.

The average execution time of the `vecAdd_2048L` is as good as 3.6 us. Considering that during this time the CPU must perform 16384 additions, the peak flops for the execution of this Codelet is 4.5 GFLOPS. The implementation of the Codelet also benefits from SIMD instructions, as its code is simple enough for the compiler to exploit SIMD extensions in the x86 architecture. In general, Codelets have the potential to provide extra optimization opportunities for Codelets, as they limit the code and leverage its properties.

Another important finding from this experiment is the noticeable low execution time of SU operations. Although for Out of Order execution the utilization is relatively high. Many of the operations performed in the SU emulate elements that would be implemented in real circuitry featuring logic gates. The emulation results are encouraging and demonstrate that the SU can be a simplified architecture that can easily fit in the die area.

However, when comparing this scheduling execution time to the execution time of Codelet and Memory instructions, and the scalability results, it is possible to conclude that the size of Codelets (i.e. size of emulated registers) for the current architecture is too small. The asymptotic behavior in Figure 7.10 can be explained by the inability to maintain the CUs occupied for a longer time. The SU takes a given time to schedule an instruction into a CU. In order to maintain all CUs busy, the time to schedule instructions for all units must be much smaller than the time a CU is busy. If instructions are too small, the scheduling bottleneck drives computation and scalability. This will be further supported in the Matrix Multiplication example.

Finally, these experiments have also shown how it is possible to use traditional compiler techniques to improve performance in the execution of SCM programs. The similarities between the SCM assembly code, and traditional assembly code, in addition to the properties for execution of SCM instructions enable static analysis that could potentially benefit the overall execution of the program. We observe also an opportunity for scheduling techniques, that would, for example, reduce the divergence of execution of memory instructions as observed in figure 7.12.

7.5 Example 2: Dense Matrix Multiplication

Dense Matrix multiplication is one of the most widely used kernels in scientific computation and data sciences. This kernel has been studied thoroughly and it is well known to most readers. Furthermore, this kernel can benefit from the SCM execution model. Tiling is a well known pattern used for exploiting parallelism [87]. In matrix multiplication, tiling provides an structure that can be easily mapped to the hierarchical structure of the Sequential Codelet Model.

The underlying strategy adopted by this work is a tiled Matrix Multiplication. A Codelet of level L1 is created which performs a single matrix multiplication over two L1 registers, storing the result in another L1 register. This Codelet is called `MatMult_2048L`. The matrix multiplication performed inside this Codelet is fixed in size, and it is used as a tile of larger matrices. For the experiments presented in this work, square matrices that are multiple of the tile size were used to avoid dealing with corner cases that will not necessarily provide valuable information at the moment. Figure 7.13 shows an overall diagram that describes the Matrix Multiplication algorithm. The operation to be performed is $C = C + A * B$.

The `MatMult_2048L` uses registers of size 2048L (i.e. $2048 \times 64 = 131072$ bytes). These registers can fit a square tile of dimension $\sqrt{\frac{2048*64}{8}} = 128$. When copying data between memory and registers, depending on the overall matrix size, the elements in memory may not be contiguously allocated. Consequently, an special 2D memory access operation is required. Two memory Codelets are defined for loading and storing tiles form and to memory. These Codelets are named `LoadSqTile_2048L` and `StoreSqTile_2048L` respectively. As depicted in Figure 7.13, each `load tile` Codelet has an operand called `dist` (for distance). Distance corresponds to the padding between two consecutive rows (assuming row major storage for the matrices). Once tiles A, B and C have been loaded, the `MatMult_2048L` Codelet is ready to execute. Notice that the tile C is shown in both the input of the Codelet and the output. The C tile is an example of a `READWRITE` operand since it reads from C and writes to C.

Inside of the implementation of the Codelet `matMul_2048L` a simple 128x128

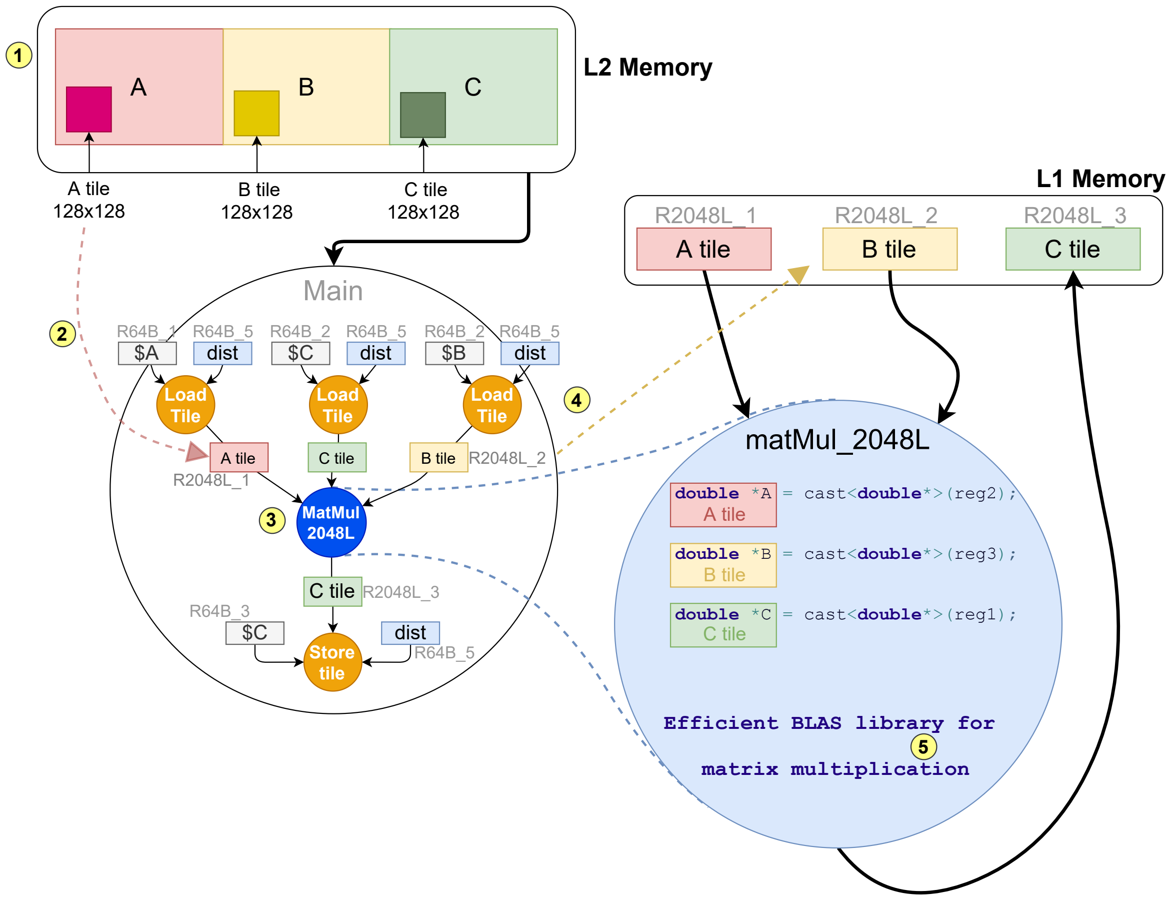


Figure 7.13: Tiling strategy for matrix multiplication. 1) Matrices are divided into tiles of 128x128. 2) A special memory Codelet performs a load operation on tiles A, B and C. Since tiles are usually non-contiguous, a distance is send as part of the operands. 3) the matrix multiplication Codelet is applied. 4) the `Rnum_2048R` registers contain the 128x128 tile. 5) Regular matrix multiplication can be applied, it is possible to use highly optimized BLAS libraries.

matrix multiplication operation is performed. It is possible to use an optimized version of this Codelet by referring to any of the available BLAS libraries. This work uses three implementations: No optimized Matrix Multiplication, user optimized Matrix Multiplication, and Matrix Multiplication with Intel's MKL library. We use these versions to study Codelet optimizations and its effect on the program execution time with respect to scalability of the architecture.

```
1  LDIMM R64B_1, 0; // Loading base address A
2  LDIMM R64B_2, 131072; // Loading base address B
3  LDIMM R64B_3, 262144; // Loading base address C
4
5  // Single tile code
6  COD LoadSqTile_2048L R2048L_1, R64B_1, 128; //Load A tile
7  COD LoadSqTile_2048L R2048L_2, R64B_2, 128; //Load B tile
8  COD LoadSqTile_2048L R2048L_3, R64B_3, 128; //Load C tile
9  COD MatMult_2048L R2048L_3, R2048L_1, R2048L_2;
10 COD StoreSqTile_2048L R2048L_3, R64B_3, 128; //Store C tile
11
12 COMMIT;
```

Listing 7.5: Matrix Multiplication 1 tile: C

Listing 7.5 shows the matrix multiplication L1 SCM code for a single tile of A, B and C. The first three lines load the offsets in memory where matrices A, B and C are stored. This example hardcodes these values, but it is possible to load values from memory, as can be seen in the full matrix multiplication example in Appendix A. Lines 6 through 10 correspond to the process of multiplying a single Tile of A and B and add it into C. This is equivalent to the `Main` Codelet depicted in Figure 7.13. Notice that all operations are Codelets. Lines 6, 7, 8 and 10 are Memory Codelets. Line 9 is the matrix multiplication for this single tile.

The Memory Codelet `LoadSqTile_2048L` loads a tile, in 2D, into the register. Likewise the Codelet `StoreSqTile_2048L` stores a tile, in 2D, back into memory. In these two Memory Codelets, the first argument corresponds to the L1 register that holds the tile. The second argument contains the address where the tile begins in L2 memory. And, the third argument corresponds to the padding between rows of the same tile in memory. The padding is the distance that needs to be jumped in a 1D memory space, in order to arrive to the next row. In this particular case, the padding is for Level 2 memory. Once the tile is loaded into Level 1, all the elements are consecutive. If A, B and C are as big as a single tile, and this tile has a side of 128 elements, the distance between rows is 128 elements. Memory Codelets are powerful tools for memory transformation across levels, including gather/scatter operations.

Listing 7.6 shows the implementation for `LoadSqTile_2048L`. In SCMUlate there are three macros used to implement Memory Codelets: `DEFINE_MEMORY_CODELET`, `MEMRAGE_CODELET`, and `IMPLEMENT_CODELET`.

The `DEFINE_MEMORY_CODELET` macro let the developer define a new memory Codelet and specify its properties. Line 1 in the Listing shows the name of the Codelet and the number of operands. Line 2 describes the direction of each operand (Read, write or readwrite). Line 3 mark certain operands as address operands. When an Operand represents a memory address of L2 or it is needed to calculate an address, it must be handled carefully. If an `address` operand is not ready when an instruction is fetch, it would not be possible to determine memory dependencies. Allowing other memory instructions to be processed could possibly result in incorrect memory order of operations. In SCMUlate there is a stall in the pipeline when an address operand is not available. Once the operand is available the stall is resolved, and the instruction analysis continues as normal.

The macro `MEMRAGE_CODELET` allows the user to specify the memory ranges (in L2) that the Codelet is intend to access during its execution. The memory Codelet class features a container that stores read/write memory ranges to be accessed. No memory operation is performed in this function, instead, the container is filled with the corresponding memory locations. Line 15 uses the method `addReadMemRange()` that adds a new read range, specifying memory location and size. During execution of SCMUlate, the SU scheduling mechanism uses the container to determine memory dependencies, allowing a correct in-order memory access during out of order executions. In general, when accessing memory, the superCodelet architecture must be cautious not to change the total order of memory operations as described in the sequential program.

```
DEFINE_MEMORY_CODELET(LoadSqTile_2048L, 3,
  scm::OP_IO::OP1_WR | scm::OP_IO::OP2_RD | scm::OP_IO::OP3_RD,
  scm::OP_ADDRESS::OP2_IS_ADDRESS | scm::OP_ADDRESS::OP3_IS_ADDRESS);

MEMRANGE_CODELET(LoadSqTile_2048L,
  unsigned char *reg2 =
    getParams().getParamAs(2); // Getting register 2
  unsigned char *reg3 =
    getParams().getParamAs(3); // Getting register 3
  uint64_t address = reinterpret_cast<uint64_t>(reg2);
  uint64_t ldistance = reinterpret_cast<uint64_t>(reg3);
  ldistance *= sizeof(double);

  for (uint64_t i = 0; i < TILE_DIM; i++)
    addReadMemRange(address+ldistance*i, TILE_DIM*sizeof(double));
);

IMPLEMENT_CODELET(LoadSqTile_2048L,
  unsigned char *reg1 =
    this->getParams().getParamAs(1); // Getting register 1
  double *destReg = reinterpret_cast<double*>(reg1);
  int i = 0;
  for (auto it = memoryRanges->reads.begin();
        it != memoryRanges->reads.end();
        it++) {
    // Address L2 memory to a pointer of the runtime
    double *addressStart =
        reinterpret_cast<double *> (getAddress(it->memoryAddress));
    std::memcpy(destReg+TILE_DIM*i++, addressStart, it->size);
  }
);
```

Listing 7.6: Definition and implementation of the LoadSqTile_2048L Codelet

The third macro, `IMPLEMENT_CODELET`, represents the actual behavior of the

Codelet. Therefore, memory copy are for performed in this stage. This macro is the same in both Memory and Compute Codelets, since it represents the functional description of the Codelet. In the case of Listing 7.6, the ranges calculated in the macro `MEMRAGE_CODELET` are used to access memory. The `for` loop in Line 23 uses the attribute `memoryRanges` of the Codelet Class which contains the memory ranges calculated in the aforementioned macro. The line 28 uses the `getAddress` method that transforms an L2 address into an OS accessible address, leaving the SCMUlate world, and allowing programmers to think of L2 regardless of the SCMUlate memory locations at runtime.

The strategy for memory Codelets used by SCMUlate, as it was described in previous paragraphs, is merely an implementation decision, and it could be change in favor of appropriate hardware (i.e. a unique memory controller with a queue that respects the order of memory operations) when implementing the SuperCodelet architecture. The definition and implementation of `StoreSqTile_2048L` are omitted given its similarities to `LoadSqTile_2048L`. The only difference is in the direction of the operands, and the direction of the memory ranges. SCMUlate, as well as this example, is open source and publicly available on GitHub [84].

7.5.1 Three different implementations

Three different implementations of the `MatMult_2048L` Codelet were explored. Each implementation has a different performance profile. Having multiple implementations enables an analysis of how the size of a Codelet affects the SCM program execution model. The term `Codelet size` is briefly defined in this context as the time it takes to execute. This is an incomplete definition. A Complete one should consider memory requirements and computational complexity. However, this short definition will be used moving forward.

Listing 7.7 presents the definition of the `MatMult_2048L` Codelet. The `DEFINE_CODELET()` macro is used to describe the compute Codelet and its properties. Line 1 shows the Codelet name and determines that it uses 3 operands. Line 2 describes the direction of the first operand. This operand corresponds to the C tile, therefore it has a readwrite direction. Line 3 contains the direction of the second operand (read tile of A), and Line 4 the direction of the third operand (read tile of B).

Listing 7.7 also shows the first implementation version: No-optimized Matrix Multiplication. The macro `IMPLEMENT_CODELET()` contains the naïve Matrix Multiplication implementation in lines 12 through 16. No further details are necessary.

```
1  DEFINE_CODELET(MatMult_2048L, 3,
2    scm::OP_IO::OP1_WR | scm::OP_IO::OP1_RD |
3    scm::OP_IO::OP2_RD |
4    scm::OP_IO::OP3_RD);
5
6  IMPLEMENT_CODELET(MatMult_2048L,
7    // Obtaining the operands
8    double *A = this->getParams().getParamValueAs<double*>(2);
9    double *B = this->getParams().getParamValueAs<double*>(3);
10   double *C = this->getParams().getParamValueAs<double*>(1);
11
12   for (int i=0; i<TILE_DIM; i=i+1)
13     for (int j=0; j<TILE_DIM; j=j+1)
14       for (int k=0; k<TILE_DIM; k=k+1)
15         C[i*TILE_DIM + j] +=
16             ((A[i*TILE_DIM + k])*(B[k*TILE_DIM + j]));
17 );
```

Listing 7.7: No optimized version of `matMult_2048L`

Following, Listing 7.8 contains the second implementation version: user optimized Matrix Multiplication. The `DEFINE_CODELET` macro has no changes, therefore it is not shown in this Listing. Inside the `IMPLEMENT_CODELET()` macro there is only a single change in comparison to Listing 7.7. Line 8 and 9 have been swapped. This change allows a modern compiler to easily vectorize this code using SIMD extensions.

As will be shown later in the results, the execution time of this version is considerably smaller than the naïve implementation.

```
IMPLEMENT_CODELET(MatMult_2048L,
  // Obtaining the operands
  double *A = this->getParams().getParamValueAs<double*>(2);
  double *B = this->getParams().getParamValueAs<double*>(3);
  double *C = this->getParams().getParamValueAs<double*>(1);

  for (int i=0; i<TILE_DIM; i=i+1)
    for (int k=0; k<TILE_DIM; k=k+1)
      for (int j=0; j<TILE_DIM; j=j+1)
        C[i*TILE_DIM + j] +=
            ((A[i*TILE_DIM + k])*(B[k*TILE_DIM + j]));
);
```

Listing 7.8: User optimized version of `matMult_2048L`

```
IMPLEMENT_CODELET(MatMult_2048L,
  // Obtaining the operands
  double *A = this->getParams().getParamValueAs<double*>(2);
  double *B = this->getParams().getParamValueAs<double*>(3);
  double *C = this->getParams().getParamValueAs<double*>(1);

  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
              TILE_DIM, TILE_DIM, TILE_DIM, 1, A,
              TILE_DIM, B, TILE_DIM, 1, C, TILE_DIM);
);
```

Listing 7.9: Intel MKL version of `matMult_2048L`

Finally, Listing 7.9 uses Intel's Math Kernels Library (MKL) to highly optimize the execution of Matrix Multiplication. This version can also be seen as the most a developer could optimize this Codelet, given the advanced optimizations used by the MKL library.

7.5.2 A GPU implementation of Matrix Multiplication

Recent development on compilers for High Performance Computing has resulted in an easy translation of code for acceleration devices. In particular, OpenMP offloading features, introduced in version 4.0, allows compilers to easily target sections of code to accelerators. Likewise, OpenMP 5.0 introduced code annotations that enable unified shared memory between accelerator and host. The Intel OneAPI Framework features a new compiler [88] capable of offloading to Intel GPUs, including the Gen 9.5 architecture in the testbed used by this work. Therefore, it is possible to write code that offloads directly into the GPU.

Listing 7.10 shows the implementation of the Matrix Multiplication Codelet using OpenMP 5.1 offloading features to target the Intel integrated GPU. The `_Pragma()` form is used because `IMPLEMENT_CODELET()` is a C macro, therefore the commonly used `#pragma` is not permitted here. Notice that, in comparison to the other three versions previously introduced, the only change of this code is the annotation that enables GPU offloading. Furthermore, OneAPI allows to use the MKL version of `cblas_dgemm` for the GPU by using OpenMP's 5.1 feature, `target variant dispatch`. For more information about this please refer to the OpenMP 5.1 specification document.

There are two important considerations. First, the reader must understand the purpose of using OpenMP as to be able to easily target the GPU. When using heterogeneous computation in the SCM model, the CU will possibly have a different execution model to SCM. Therefore, the use of OpenMP is limited to translation of code to target the GPU and its execution model. This does not undermine nor it contradicts the principles of the SCM model. Second, both the OpenAPI compiler and the use of heterogeneous computation in this code are in early development stages, as will be observed in the results presented later on in this Chapter.

```
IMPLEMENT_CODELET(MatMultGPU_2048L,
  double *A = this->getParams().getParamValueAs<double*>(2);
  double *B = this->getParams().getParamValueAs<double*>(3);
  double *C = this->getParams().getParamValueAs<double*>(1);
_Pragma("omp target data
         map(to:A[:TILE_DIM*TILE_DIM],B[:TILE_DIM*TILE_DIM])
         map(tofrom:C[:TILE_DIM*TILE_DIM])")
  {
_Pragma("omp target variant dispatch
             use_device_ptr(A, B, C)")
    {
      cblas_dgemm(CblasRowMajor, CblasNoTrans,
          CblasNoTrans, TILE_DIM, TILE_DIM, TILE_DIM,
          1, A, TILE_DIM, B, TILE_DIM, 1, C, TILE_DIM);
    }
  }
);
```

Listing 7.10: Definition of the GPU Codelet matMulGPU_2048L using Intel's MKL support for gen9 GPU and OpenMP 5.1 features

7.5.3 Matrix Multiplication CPU Results

To properly evaluate Matrix Multiplication it is necessary to use different Matrix dimensions. These experiments focus on square matrix multiplication. Four different dimensions, multiple of the tile size, are used: 10, 20, 30 and 40. A single tile is 128x128 elements. Therefore, the 10x10 tile experiment, for example, computes matrix multiplication for 1280x1280 elements in each matrix. Additionally, strong scaling is studied. Finally, experiments lead to an exploration on how Codelet size and Scheduling time relate to each other, providing the first hints towards answering the question: What is the appropriate Codelet Size in the SCM Model?. All experiments use Out of Order execution mode only. For CPU results, the three aforementioned versions for

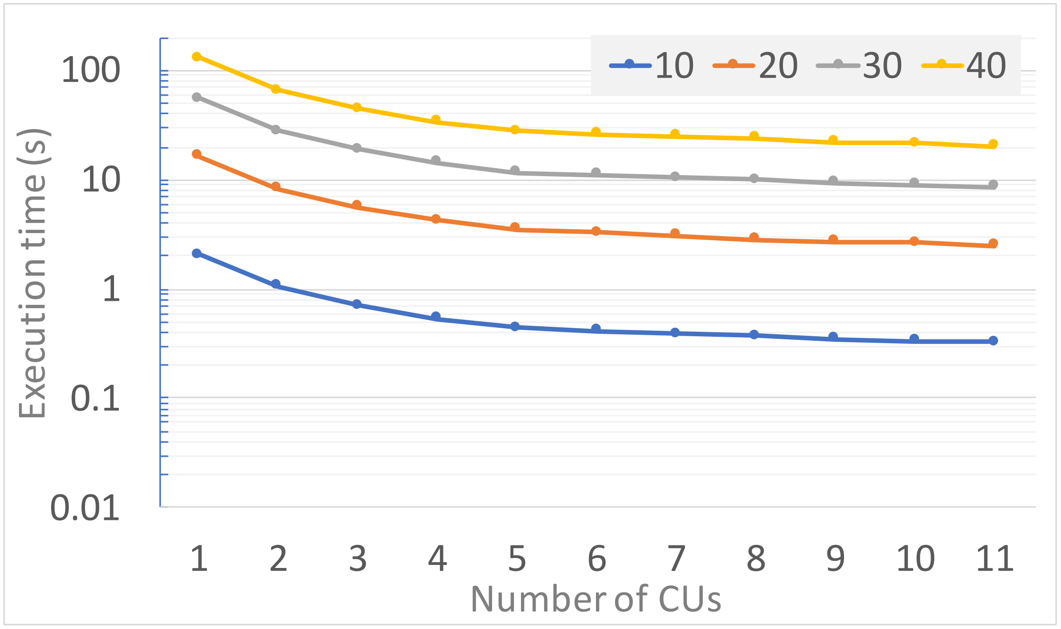


Figure 7.14: Execution time vs number of CUs for the naïve version of Matrix Multiplication. Logarithmic scale in the vertical axis.

CPU are used: No-optimization, user optimized and Intel MKL. Results will be shown first, and their analysis will be left to Section 7.5.5.

7.5.3.1 Execution Time

Let us begin with the execution time. As expected, the execution time depends on the number of Compute Units (CUs). The lower the execution time, the better the application performance is. Likewise, as we increase the number of CUs, it is expected that the execution time decreases. Ideally, as more compute capabilities area added, less execution time should be observed.

Figure 7.14 shows the execution time as a factor of the number of CUs for the naïve Matrix Multiplication implementation of the `MatMult_2048L` Codelet. Figure 7.15 shows the same metric for the user optimized matrix multiplication, and Figure 7.16 for the Intel MKL matrix multiplication version.

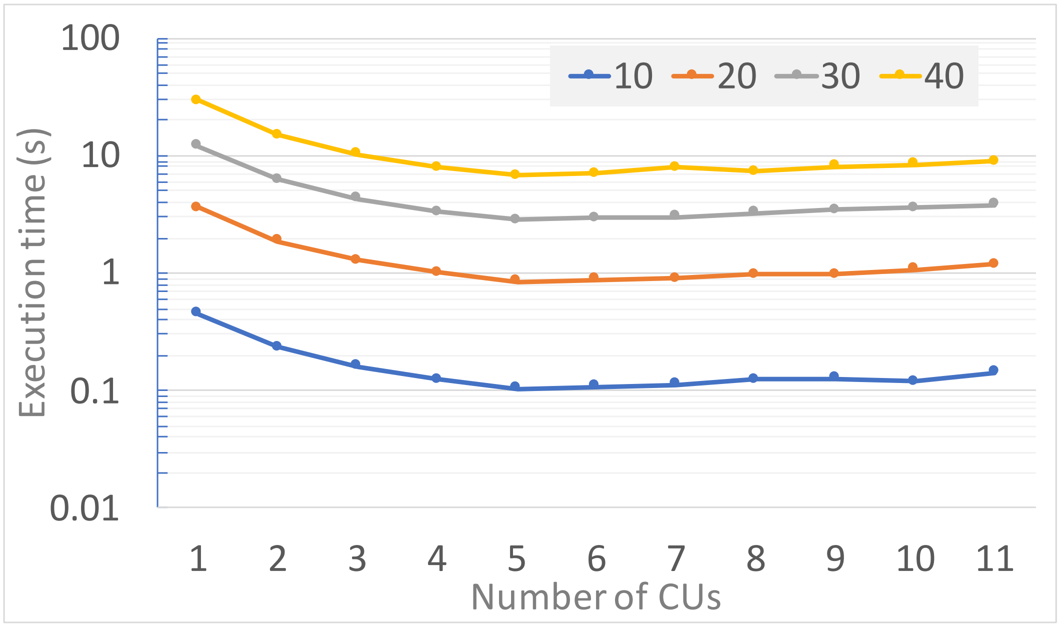


Figure 7.15: Execution time vs number of CUs for the optimized version of Matrix Multiplication. Logarithmic scale in the vertical axis.

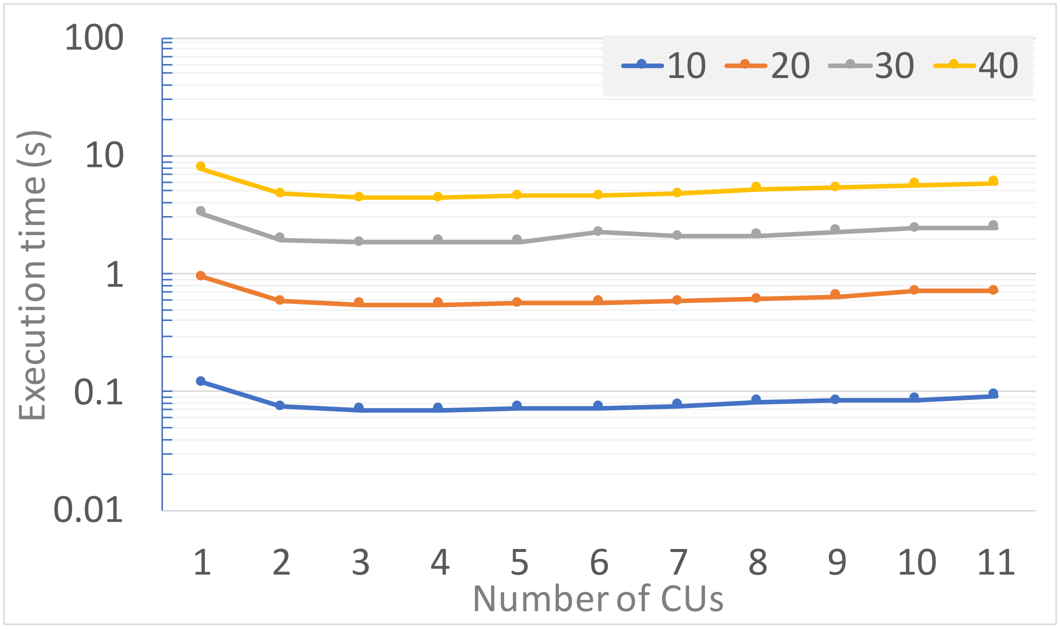


Figure 7.16: Execution time vs number of CUs for the Intel MKL version of Matrix Multiplication. Logarithmic scale in the vertical axis.

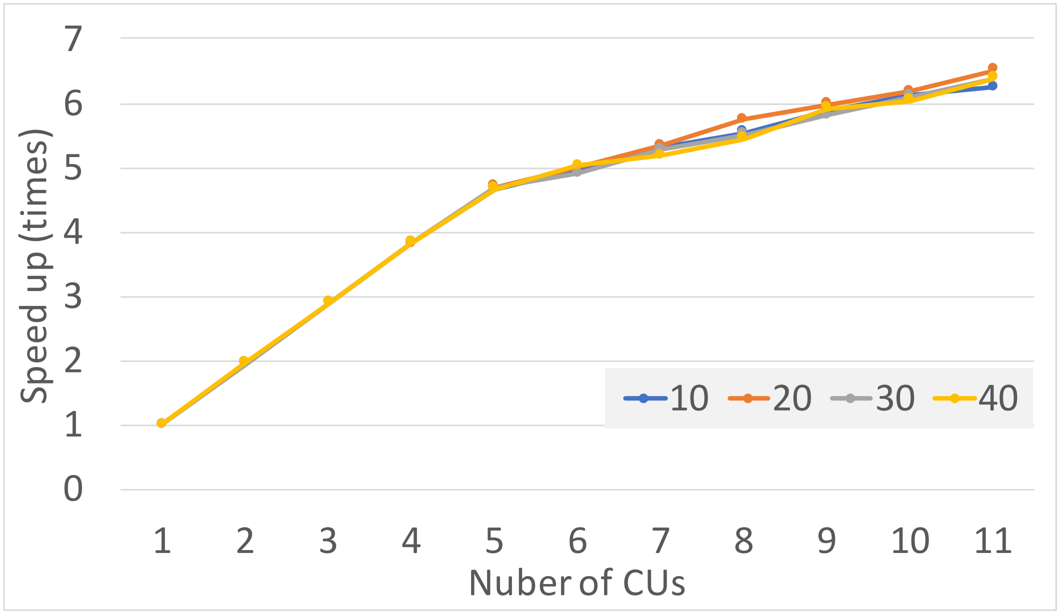


Figure 7.17: Strong scaling for the naïve version of Matrix Multiplication.

7.5.3.2 Scalability

Strong scaling is used to understand the effect that increasing the number of CUs has in the execution time. Scalability is calculated with respect to the sequential execution (CU=1) in eac problem size and optimization version. The following results are intended to answer two questions: 1) can the SCM model scale? and 2) how the size of Codelets tampers scalability?. The second question is important for designing a hardware implemented SuperCodelet architecture. The expected behavior is that for each CU added, the scalability factor is increased by 1, forming the straight line $x = y$.

Each experiment has a different baseline, therefore, the reader must be careful when comparing results. The overall behavior is what is being compared, with the intention to evaluate the machine and not the kernel itself. Additionally, these experiments aim to understand if the size of the problem has implications with respect to the scalability.

Figure 7.17 shows strong scaling results for the original Matrix Multiplication, with no optimizations applied. Figure 7.18 shows the same metrics for the User Optimized version. And, finally, 7.19 for the version that uses Intel MKL for the CPU

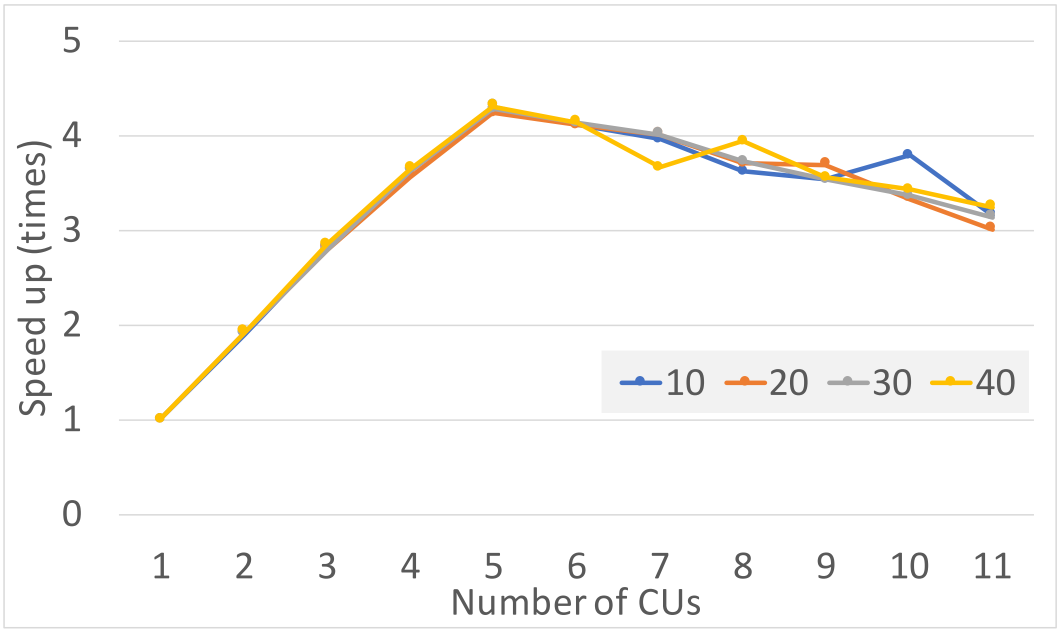


Figure 7.18: Strong scaling for the user optimized version of Matrix Multiplication.

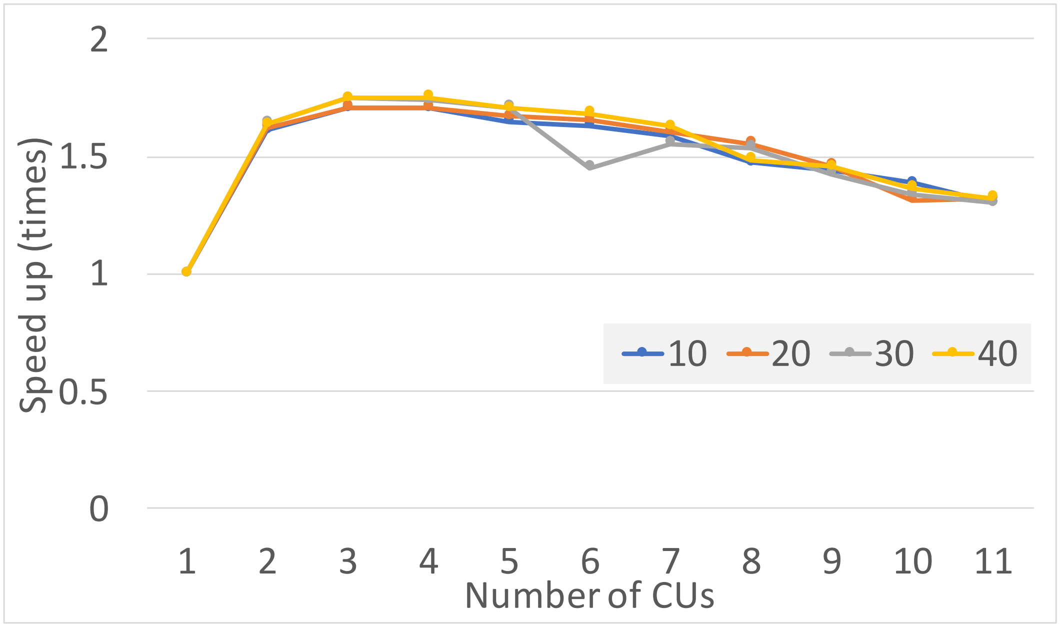


Figure 7.19: Strong scaling for the Intel MKL version of Matrix Multiplication.

inside of the `MatMult_2048L` Codelet. Vertical axis corresponds to the speed up, in times, with respect to $CU = 1$, horizontal axis corresponds to the number of CUs.

7.5.3.3 Instruction size

Another important aspect to understand the time and scalability results is the execution time for the three most important instructions of this kernel: `StoreSqTile_2048L`, `LoadSqTile_2048L`, `MatMult_2048L`. To this end, the SCMUlate tracing tool outputs statistics on instruction execution.

These results aim to answer the question: What is the effect on the number of CUs in the execution of a Codelet? Ideally, as described in the theoretical background, no changes to the execution time should occur. The larger the slow down, the more performance degradation exists with respect to the sequential execution time. The sequential execution time is used as baseline since this execution mode is expected to have the less noise from the rest of the system and runtime.

Figure 7.20 shows the instruction degradation as the number of CUs increases. The value for 1 CU is hidden since it is the baseline (i.e. $slowdown = 1$). Furthermore, only matrix size $M = N = K = 10$ is shown. No considerable changes were observe in the degradation with respect to the size of the matrix.

7.5.4 Matrix Multiplication GPU Results

For evaluating the behavior of the SCMUlate system in the presence of GPUs, the same experiments were executed but for the `MatMult_2048L` GPU Codelet. Likewise, different dimensions, as a factor of the tile size, are shown. These experiments aim to act as proof of concept of a heterogeneous SuperCodelet architecture, as described in Chapter 5.

As previously mentioned, these experiments use the Intel Gen 9.5 architecture. In particular, the Intel HD Graphics 630, containing 24 Intel GPU Execution Units [83]. Support for GPU offloading in these GPUs is limited and actively under development,

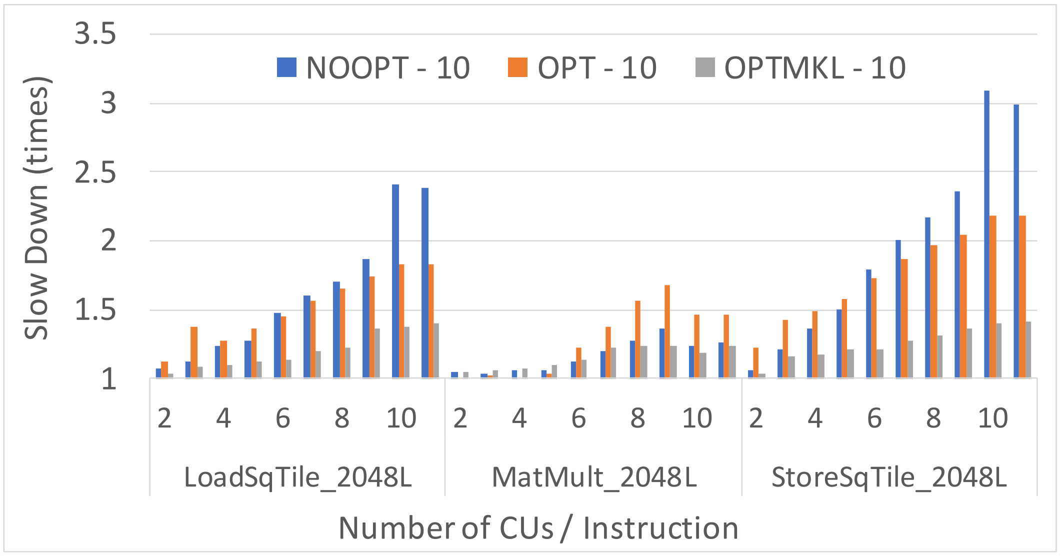


Figure 7.20: Codelet Performance degradation for Matrix Multiplication. M=N=K=10 tiles

during the time of writing this thesis, therefore, software and tools used to create these graphs are mostly in beta versions.

7.5.4.1 Execution time

Figure 7.21 shows the execution time of the Matrix Multiplication microbenchmark when running on the GPU. The lower the execution time, the better the performance of the application.

The number of CUs represent the number of CPU threads actively pushing work into the GPU. SCMUlate runtime has not been leveraged to be GPU-aware. Instead, a GPU Codelet is used which contains a call to the GPU Kernel. The execution of the GPU Codelet stalls the CPU core that manages the instantiation of the GPU Kernel. This approach results in considerable reduction of system utilization, since CPUs cores are just waiting for the GPU. A combined CPU+GPU execution is currently being explored.

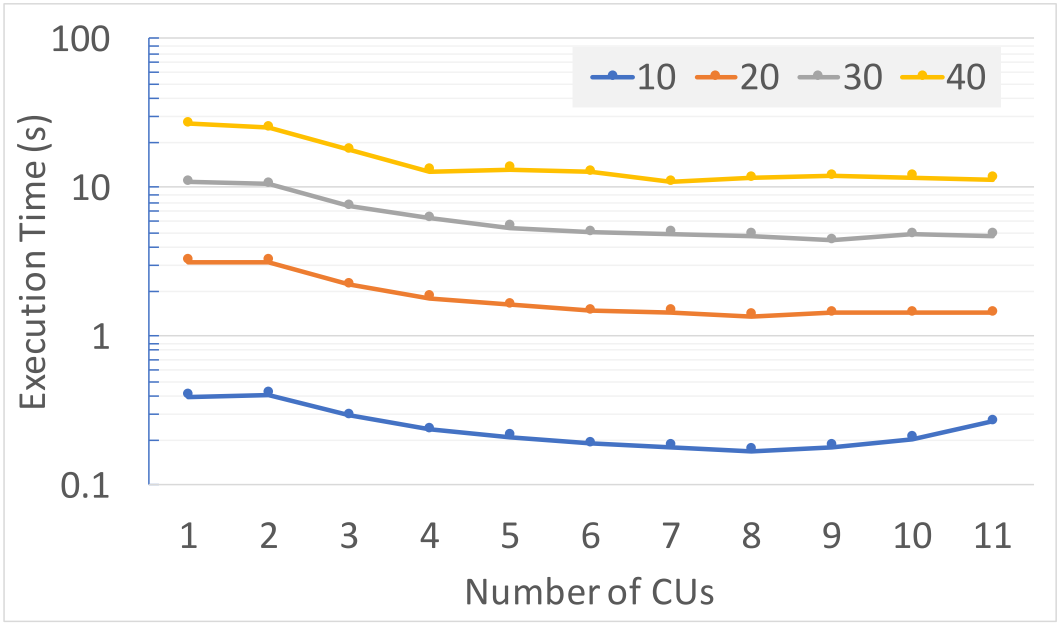


Figure 7.21: Execution time vs number of CUs for the GPU version of the Matrix Multiplication Codelet. Logarithmic scale in the vertical axis.

7.5.4.2 Scalability

Figure 7.22 shows the strong scaling results for the GPU version of the Matrix Multiplication Codelet.

As mentioned in the execution time results, for GPU execution, a CU represents a CPU thread pushing work into the GPU. A more appropriate mapping between GPU Execution Units and the SCMUlate is still required to better map the Sequential Codelet Abstract Machine to the test base architecture. Consequently, the strong scaling results must be carefully considered.

7.5.4.3 Instruction size

Finally, the instruction performance degradation is studied for the GPU execution mode. Figure 7.23 shows the slow down in the execution of compute Codelets as the number of Computational Elements is increased. More details will be analyzed and discussed in the following section.

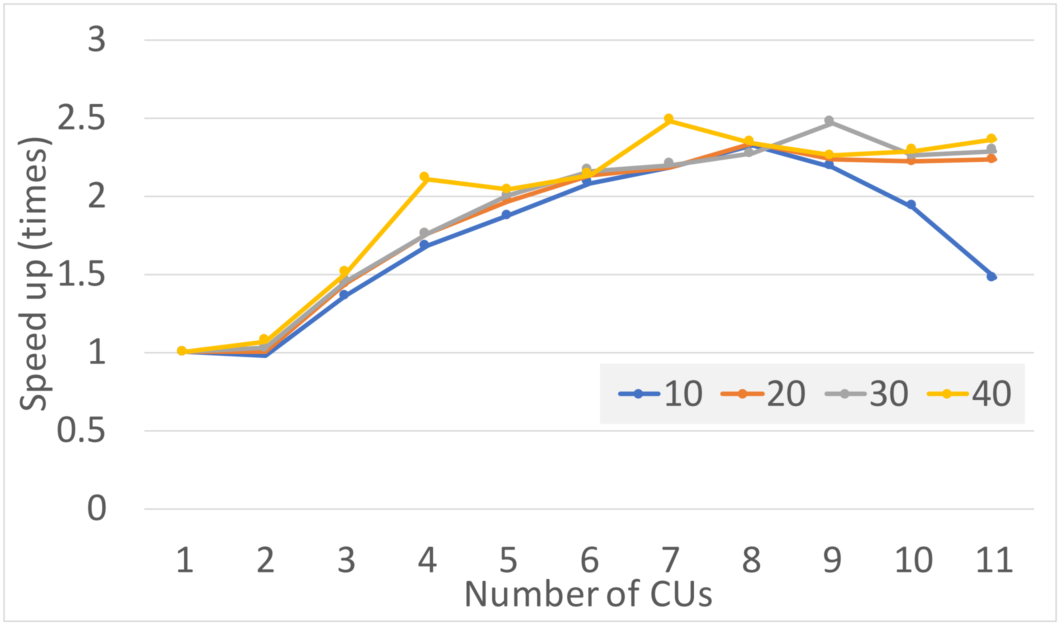


Figure 7.22: Strong Scaling for the GPU version of the Matrix Multiplication Codelet. Logarithmic scale in the vertical axis.

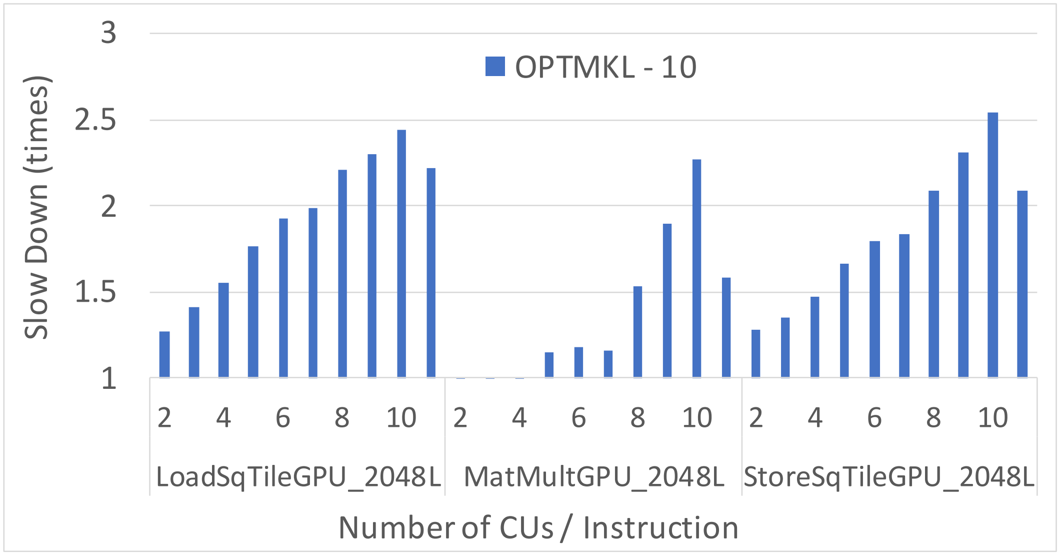


Figure 7.23: Strong Scaling for the GPU version of the Matrix Multiplication Codelet. Logarithmic scale in the vertical axis.

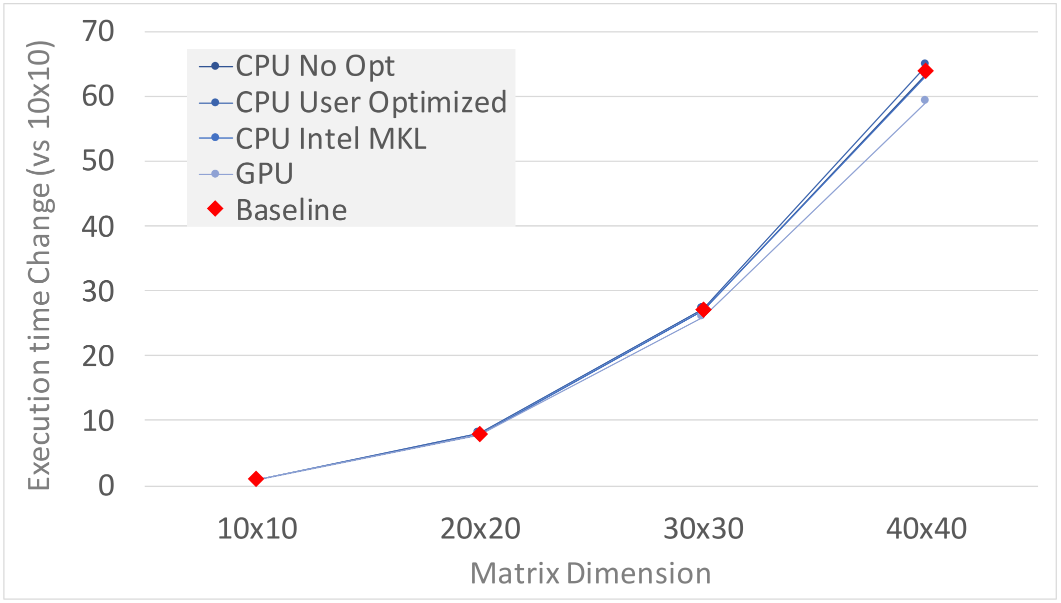


Figure 7.24: Big O for Matrix Multiplication as observed from L1. $\mathcal{O}(N^3)$ for the implemented MM.

7.5.5 Comparison and analysis

Having lay down all the results obtained from the SCMUlate emulator for the Matrix Multiplication implementation, this section presents some important observations and conclusions that can be drawn from them. Some additional graphs are presented to summarize or emphasize the findings.

First, the execution time results are studied. The operational complexity for the commonly used square matrix multiplication algorithm is $\mathcal{O}(n^3)$. The complete SCM code for Level L1 that implements Matrix Multiplication is presented in Appendix A. Regardless of algorithm used inside the the Matrix Multiplication Codelet (i.e. no-optimization, user optimized, and Intel MKL), the execution time for a Codelet is static, and known at compile time. Therefore, at level L1 it is possible to study Big O operational complexity in terms of the number of tiles. Based on the algorithm presented in Appendix A it is possible to observe the same $\mathcal{O}(N^3)$ complexity. Where N, is expressed in terms of number of tiles.

Figure 7.24 studies the change in execution time as the size of N is increased. The experiment for $N = 10$ is used as comparison point for all the other execution

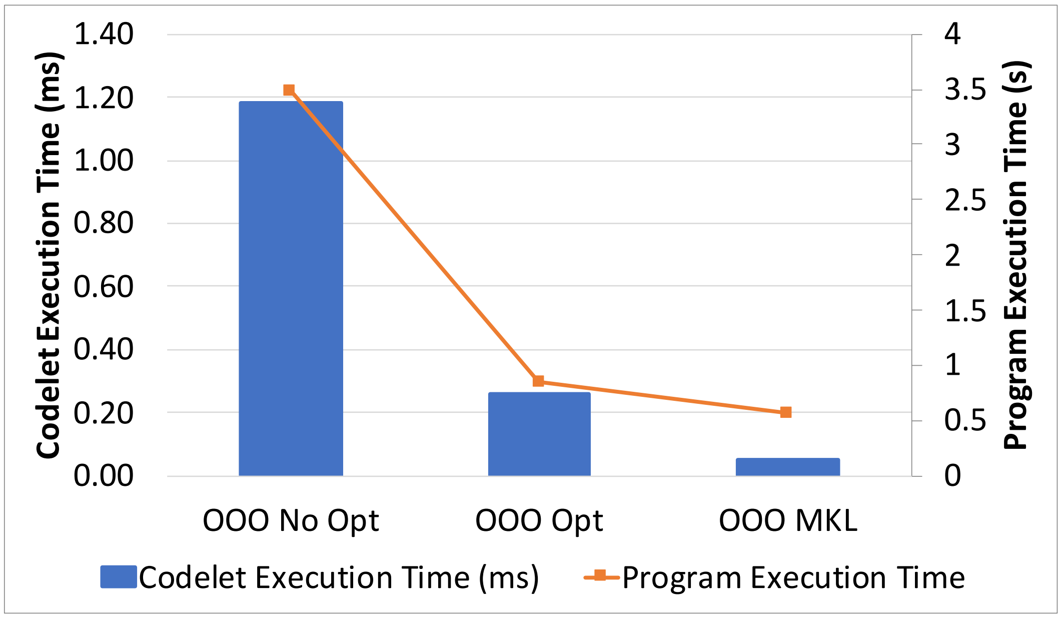


Figure 7.25: 5 CUs and M=N=K=20. Bars: Codelet execution time (left axis). Line: Program Execution Time (right axis).

times. The red dots correspond to ideal $\mathcal{O}(N^3)$ for $N = 1, 2, 3, 4$, considering the baseline 10×10 is $N = 1$. Despite of the execution mode or architecture (CPU or GPU), the observed operational complexity in terms of the size of the problem is the expected value. The ability to interpret an algorithm in the upper levels of the SCM machine is paramount for performance analysis and algorithm design in future large scale computer systems. The isolation of levels in the SCM also provides a framework for using already existing compiling techniques.

Additionally, the different versions to implement the `MatMult_2048L` Codelet yields to different execution time. Between versions, it is only the execution of a Codelet that changes. Therefore, it is expected that the trend in reduction in the overall program execution time is proportional to the execution time of the Codelet. Figure 7.25 overlays the Codelet execution time with the overall program execution time. The left axis is used for the Codelet execution time, and the right axis is used for the program execution time. They are 3 orders of magnitude apart.

The ratio between Codelet execution time and program execution time between

Out of Order No optimized and user optimized is about the same as the ratio between program execution time between the same two experiments. For Codelet execution time $\frac{OOO\_NoOpt_{cod}}{OOO\_Opt_{cod}} = 4.57$, and for program execution time $\frac{OOO\_NoOpt_{prog}}{OOO\_Opt_{prog}} = 4.10$. However, this is not the same for the MKL version, in comparison to the optimized version. For Codelet execution time $\frac{OOO\_Opt_{cod}}{OOO\_MKL_{cod}} = 4.87$, and for program execution time $\frac{OOO\_Opt_{prog}}{OOO\_MKL_{prog}} = 1.49$.

As expected, the reduction in the execution time is due to the improvement in the Codelet size. However, there seems to be another bottleneck that is limiting the ability for the program to continue scaling in the same proportion. To that end, let us analyze both the scalability of the program and trend in the instruction execution time.

Figures 7.17, 7.18, and 7.19 show an interesting trend. There is no change in the scalability behavior between different matrix sizes. Figure 7.26 focuses only on size $M = N = K = 40$ for the three different implementations. Starting with the no optimized Codelet, an almost perfect strong scaling occurs between 1 and 5 CUs. Once the 6th CU is introduced, the trend is reduced, but it is still increasing. For the user optimized version, strong scaling has a similar trend. However, after the 5th CU, the execution time worsen, and the program stops scaling. Such trend is even worse for the MKL Version.

When comparing this plot to the evolution of the Codelet execution time presented in Figure 7.20 it can be concluded that some of the performance degradation is due to the degradation of the execution time of the Codelets. Furthermore, the break after the 5th CU can be attributed in part to the testbed architecture. As can be seen in Figure 7.2, a single core of the Intel i7 8700K contains two hardware threads in SMT configuration. Furthermore, after the 5th CU all the system cores has been used by some part of the architecture. 5 Cores are used by the CUs, and the SU is occupying the 6th core. Therefore, as the number of CUs is increased beyond the number of physical cores, resource sharing inside of the Intel architecture occurs, resulting in further degradation. Such behavior is worst in the UserOpt version in comparison to

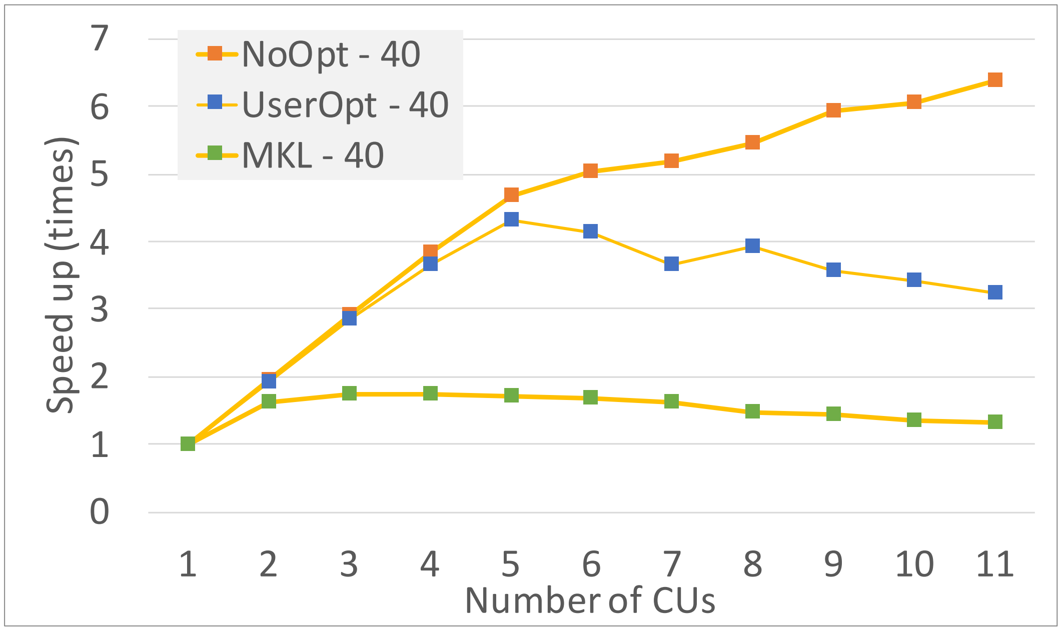


Figure 7.26: M=N=K=40. Scalability comparison for different implementations

the non optimized version, since the Codelet is using SIMD execution mode. SIMD hardware is often shared across SMT threads, after the 5th CU, each core would suffer from higher contention. Finally, memory bandwidth is also a consequence of the performance and scalability degradation. Without controlling the number of CUs that simultaneously access memory, the bandwidth to L3 is saturated. This is also the reason why the degradation for Memory Codelets seems to be worst than that of the compute Codelets.

Despite all these factors, the lack of scalability in the MKL version cannot be fully associated to them. Although some performance degradation occurs, the major problem with MKL comes from the sequential bottleneck. As the Codelet execution becomes smaller, the ability of the SU to schedule enough work across CUs is tampered. This can be explained by Amdahl's law.

7.6 Defining the appropriate size of Codelets

Figure 7.27 shows a diagram that allows to create a mathematical foundation of the number of CUs and the size of a Codelet. Let us assume that the SU will take

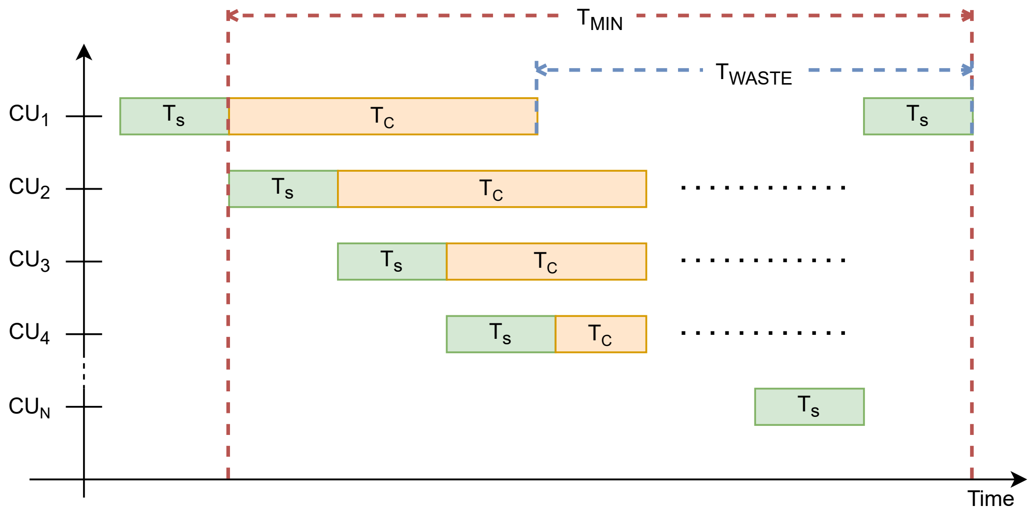


Figure 7.27: Understanding Sequential bottleneck. T_s Time to schedule an instruction into a CU. T_c Compute time. T_{min} minimum compute time to avoid T_{waste} sub-utilization

a constant time T_S to schedule an instruction (Codelet) into a CU. For simplicity, the scheduler uses a round robin mechanism, although any other scheduler with static scheduling time would result in the same analysis. The Codelet has a compute time T_C. Notice that as the number of CUs increases, the time it takes for the scheduler to finish assigning work to all the units grow, proportional to the number of CUs. Assuming no synchronization cost (i.e. the Codelet starts execution as soon as it is assigned to a CU), and no extra overhead in the system, the time between one allocation and the next one, for the same CU is equal to $T_s \times N_{CU}$ where N_{CU} is the number of CUs in the systems. This time is referred to as T_{MIN}, and it will be the shortest time for all CUs to be busy all the time. If T_C is shorter than T_{MIN}, there will be a waste in execution equal to T_{WASTE} in the figure.

This analysis is overly optimistic. It assumes that that, at any time, instructions will be available for the SU to schedule (e.g. no dependencies between instruction stream). If a program does not provide enough parallelism within the window of instructions that concerns the Out of Order execution engine, there will not be enough instructions to be scheduled. However, this is a application specific issue that cannot be solved in the architecture side. It is always possible to increase the window that the OoO engine is aware.

Another drawback of this analysis is that it ignores there will be some minor instructions (e.g. basic arithmetic and control flow instructions), allocated to some ALU that will take certain execution time. Even if the T_S for these instructions is negligible (i.e. $T_{S_{ALU}} << T_S$, programs often include several of these instructions between Codelet instructions. Such problem can be mitigated if T_C is considerably increased beyond T_{MIN}. However, if the size of a single Codelet is too large, any dependent instruction after the Codelet will have a long waiting time in the OoO queue.

7.6.1 Empirical observations from Matrix Multiplication

Empirically, we observe that the execution time of the Codelet for the No-optimized Matrix Multiplication Codelet is $1.2ms$. Using the trace, it was observed that the distance between consecutive instructions scheduled was between $20us$ and $30us$, influenced by some other instructions on the program. For 11 CUs, is between $T_{MIN} = 220us$ and $T_{MIN} = 330us$. In Matrix Multiplication it seems possible that a single order of magnitude would provide enough time to allow for more parallelism.

For the User Optimized Matrix Multiplication, the Codelet execution time $T_C = 349us$, For 6 CUs, $T_{MIN} = [120us, 180us]$. This means that the ratio between T_{MIN} and T_C is between 2 and 3 times. Given the additional noise in the system, this is the point where not enough parallelism is achievable.

Finally, for the MKL Codelet version, the compute time is about $T_C = 50us$. With a $T_S = 20us$, It is not possible to scale beyond 2 CUs. Figure 7.26 and 7.19 show how these results are confirmed by the observations in SCMUlate.

7.6.2 Designing L1 with T_S

In order to allow for more CUs in a single Level L0 while maintaining scalability, there are two strategies that could be used. For a given Codelet size T_S it is possible to improve T_S so that $T_S < \frac{T_C}{N_{CU}}$.

A particular case for this strategy is using SIMD-like execution modes in L1. Such change would lead to modifying the definition of T_{MIN} to not be dependent of the number of CUs. For example, the concept of warp in a modern GPU requires a single scheduling time for all the different hardware threads that form the warp. However, not every application would necessarily benefit from SIMD-like execution models.

A second strategy would be to increase the size of the Codelet. The size of a Codelet is determined by the time and memory complexity of the underlying algorithm in the Codelet implementation. For a given compute capabilities for L0 cores (i.e. FLOPS), and memory bandwidth between L0 and L1, it is possible to determine what would be the size for registers and problem within the Codelet that would yield to the desired compute time. Such information could be used to also specify the size of registers.

Designing L1 will be a combination of using the equations presented in 7.6 and applying knowledge of the different limits in the technology to utilize. The compute capabilities in L0, as measured in FLOPS, and the bandwidth between L1 and L0 could be use to calculate T_C. Two different extremes are observed. The execution time for a memory Codelet that uses no operations in L0 is bounded by the bandwidth. And a Compute Codelet that access no memory in registers is bounded by FLOPS. Given the desired N_{CU}, and T_C it is possible to calculate the target T_S. Or given a T_S limit, it is possible to calculate the number of N_{CU}.

If more CUs are desired, it is always possible to continue the design on L2, using multiple L1s as compute units of L2. The analysis would be similar to the one described in this section.

7.6.3 Application and Codelet Size

When porting an application to the SCM, it may be the case the T_C cannot be achieved due to limited memory or compute complexity. In such cases, the reader is reminded that each level of SCM is a Turing complete machine. It is not always necessary to map Codelets to the levels below. It is possible to use the Compute Capabilities of the levels above to execute parts of the program. Any of the in-memory compute or on-network compute strategies that exists can be used as extensions of the SCM model.

Chapter 8
RELATED WORK

Computer architecture is the art of defining the structures of different components and the abstractions that rule the behavior and interaction of those components. The search space of parameters that define the architectures is large. However, the objective is to create a system that is capable of solving problems of different natures, that is, a general purpose computer. There has been several attempts to create general purpose parallel architectures. In this section we focus on some architectures that have influenced this work the most, and that compare to the proposed approach.

Computation is about data transformation according to mathematical operations. Data is stored in some form of numerical value that encodes information. For example, while numbers are usually encoded in a base 2 numeral system (i.e. binary numbers), other data use numeric representations with a predefined format definition. For example colors are usually represented as a triplet of numbers (e.g. RGB). Computers rely on the interaction of three different set of components: computational elements that perform mathematical operations, memory elements that store data, and I/O interfaces that allows the system to interact with the outer world. These components are linked together through interconnection networks. At a high level programming a computer consist of assigning work to the different components of the machine through the use of pre-determined instructions. This work usually results in performing a complex operation through the transformation of data in memory by using the available hardware operations.

On the software side, programming models define abstract machines and execution models as well. These models must be map to the execution model of the underlying hardware. Runtime systems bridge the abstraction of the execution model of the programming language to the execution model on the underlying architecture. A compiler bridge the programming language syntax and its semantics to the runtime systems and underlying hardware ISA.

Two major groups of computational models exist. They represent two ends of a spectrum: Dataflow machines and Von Neumann Machines. In the middle there are hybrid architectures that take advantage, one way or another, of elements of the two larger groups. These models may be implemented purely in software, hardware or as a combination of hardware software co-design strategies.

8.1 Dataflow systems

Karp and Miller first described the concept of dataflow computational models in 1966 [35]. Programs defined in Dataflow are described as a directed graph where each node represents an operation. The arcs connecting the nodes represent data and control flow dependencies between operations. In 1974, Jack Dennis defined the first Dataflow architecture [36] and its corresponding programming language [37].

An important note is that dataflow is a model of computation. Therefore, it can influence the design of abstract machines of software and hardware. In this section we refer to hardware architectures for which the instructions are expressed as a dataflow diagram and which have no concept of program counter or sequential program execution order whatsoever. Dataflow architectures are characterized for having many computational elements interconnected together. Some dataflow systems use a centralized token memory, while others use a point to point communication between processing elements.

After Dennis' first data flow machine [36], many more dataflow systems were defined. We describe some of the most influential systems. Dennis revisited his original design in 1980 [89] and proposed another architecture with an emphasis in the interconnection network system. It revisited how to maintain communication cost as low as possible while the number of cores increased. Dennis recognize that dataflow based interconnections tend to grow in complexity faster than traditional multiprocessing switching interconnections, but its grow can be maintained almost linear.

Some dataflow machines depend on circular pipelines. These are composed of processing elements, a token storage and a matching mechanism to assign tokens to instructions. The processing elements execute instructions and send the results back into the pipeline. Results are obtained by the token match mechanism and stored in the token memory. Finally an scheduling mechanism selects the next ready instruction and assign it to the processing elements. An example of a circular pipeline was used in James Rambaugh's multicore architecture [38] in 1977. Rambaugh's system consisted of multiple cores connected to an scheduler logic and an structure memory. An static dataflow approach was used within each processor and executed through a circular pipeline as described above. An execution frame enclosed an static dataflow graph and mapped into a single processing element. This enclosing execution frame resembles procedures that map into a single core. Inter-procedural creation and return values happens through the use of a special `APPLY` operation. This operation creates new procedures that map to other computational elements. If computational elements are starved by the number of procedures, the context was swapped into a special external memory.

Another important architecture was the MIT Tagged Token Dataflow Architecture (MTTD) by Arvind et al defined in the late 80's [39]. This architecture used a technique of coloring dataflow tokens to allow for dynamic re-utilization of dataflow programs. Contrary to the static versions, dynamic dataflow allowed the same graph to be executed multiple times, while differentiating executions through a tagging mechanism. New operations allowed to create new colors, as well as to change the colors of tokens back to the caller's color. The MIT TTDA also used the Id programming language to express dataflow graphs. The Monsoon project [90] by Papadopoulos and Culler evolved from the MIT TTDA and introduced the Explicit Token Storage (ETS) architecture. The ETS creates a frame for each independent execution of the dataflow graph. While this frame is not novel by itself, the Monsoon machine shows a potential implementation that also uses Arvind's I-structures [68] for storing tokens and data structures.

In addition to these architectures, many others were proposed: LAU (1976) [43], DDM1 Micro (1976) [41], Data Driven Processor Array (DDPA, 1983), [91], Distributed Data Driven Processor (DDDP, 1983) [92], Manchester Dataflow Computer (1985) [93], PIM-D 1986 [94], HDFM (1985) [42], and more recently the Intel CSA patent in 2016 [95].

8.2 Out of Order Execution and other ILP techniques

While architectures that use Out of Order execution are not dataflow architectures in the strict sense, these engines are inspired by dataflow mechanisms. OoO uses a dependency analysis within a window of instructions to exploit parallelism. Several dataflow analysis that occurs during OoO execution rely on important work done in dataflow architectures. Out of Order execution takes advantage of the locality provided by the user that writes the code sequentially, as well as the potential compiler optimizations that could increase the parallelism.

Among the first out of order execution systems ever created, there is the Scoreboard mechanism used in the CDC6600 [55] (1964). Following, the Tomasulo's algorithm [57] removed potential stalls on Read After Write (RAW) conflicts in the pipeline. The IBM 360 Model 91 [4] implemented the Tomasulo's algorithm (1966). Out of order and ILP based optimizations for sequential execution of code based on dataflow techniques was later explored in the High Performance Silicon (HPS, 1985) system [96], and HPSm, a minimal simulated version of HPS (1986) [97] micro-architecture designs. Finally, interruptions in the context of Out of Order executions were defined in 1988 [98]. Out of order are still used in many single core architectures design (Intel Core processors, IBM POWER7, and others), and they were fundamental for the success of many commercially available processors in the 90's (IBM Power 1, MIPS R10000, Intel Pentium Pro, AMD K5, and others).

Among other innovations in sequential computers that increased instruction level parallelism [99] there is the concept of pipelinining [100] introduced in 1959. Pipeline breaks down the single cycle machines, into independently stages of instruction execution. Furthermore, superscalar architectures [55] firstly introduced in 1966 CDC6600, are essential for ILP. These systems feature multiple execution ALU and floating point arithmetic units. Moreover, speculative execution [101] and branch prediction [102] mechanisms allow to make progress in the execution of code, considerably increasing the performance of computer architectures. Despite their success these techniques have been recently been re-consider as they have been shown to create security risks such as the Spectre [103] and Meltdown [104] attacks.

Very Large Instruction World enables the grouping of several operations in the same instruction [105]. These techniques aim to also increase the instructions per cycle. They can benefit from compiler optimizations that are able to find set of instructions to combine.

In this work we recognize the importance of these ILP optimizations and we encourage a combination between the SuperCodelet architecture and these other architectures. To this end, the Hierarchical Turing Machine should be the one element that is able to join them under the same umbrella.

8.3 Other parallel architectures

Other parallel architectures have also been proposed. One example are the systolic arrays by Kung and Leiserson in 1978 [106]. Additionally, the vector processors described in the 60's and 70's and popularized by the Cray-1 machine [12] in 1978. Vector inspired processors where also later successfully used in the Connection Machines CM-1 and CM-2, among others [107] in 1988. Vector systems heavily influenced SIMD-based architectures widely used in GPGPU accelerators.

Several multicore and multithreading approaches have also been proposed. Simultaneous Multithreading (SMT) [108] extends the register file and program counter of an architecture, taking advantage of multithreaded programmers and the multi process capabilities of Operating Systems. The First implementation of the SMT dates back to the IBM Advanced Computer Systems project: the ACS 360 in 1968 [109]. The Two Instructions Counter approach gave the impression of two different CPUs, while allowed sharing resources through instruction coloring. Currently, it is common to find SMT-2 and SMT-4 approaches in systems such as Intel architectures, IBM POWER PC architectures, and AMD systems.

Another architecture that is worth mentioning is the Traleika Glacier architecture by Intel developed during the x-Stack project [110]. An important lesson learned by this architecture is the use of control loops to manage energy goals through frequency domains and DVFS techniques. The Master Thesis by Aaron Landwehr [111] explored different self aware techniques that used software annotations to communicate with the runtime to make better decisions towards energy savings. This is another line of work that is important in the context of this project.

Similarly, Andrew Chien proposed the 10x10 architecture [112] that relies on a new highly heterogeneous paradigm that instead of using heavy architectures to execute most of the workload, it distributes the execution across several highly optimize computational units with different capabilities. Currently, we have seen an increased interest in neuromorphic chips. These often use Dataflow approaches to execute neural networks (NN). Similar to dataflow computation, NN are represented as a graph of neurons, each with a given functionality (e.g. Convolution, ReLU, Sigmoid functions and others). Neuromorphic chips often map these neurons to dataflow inspired chips with application specific circuitry for Artificial Intelligence (AI) and Machine Learning (ML). Some examples are Google TPUs [9], IBM TrueNorth [113], and the Tianjic Chip [114].

It would be necessary to create a really extensive survey to cover all the different innovations in the field of parallel computer architectures. However, many of the elements that have already been explored would be able to fit the description of the sequential Codelet Model, either by acting as application specific ALU, or by redefining an implementation at any of the hierarchical Von Neumann architectures.

8.4 Von-Neuman/Dataflow hybrid systems

As previously mentioned, dataflow-inspired architectures and the Von Neumann-inspired architectures are two opposite sides of an spectrum of possible architectures. At one end, Von Neumann offers shared memory abstractions where operations can be mapped to any location. At the other end, dataflow uses completely independent memory arcs that connect operations with a limited mapping of memory location to instruction operands. In the middle, there has been several attempts to have hybrid models. Thorough surveys on these approaches can be found in [49] [115], [116], and [117].

The closest relatives to the work proposed in this thesis are the original Codelet Model (as explained in Chapter 2) and its predecessor project, the EARTH-MANNA architecture [62]. EARTH-MANNA project was a fully functional testbed for the EARTH model. The EARTH model aimed to demonstrate that in multithreading architectures running on multicore systems, it is possible to support fine grain parallelism with tolerable synchronization and communication overheads. The abstract machine of the EARTH model also relied on an SU unit for resource management and allocation of tasks (the equivalent of Codelets).

The Monsoon project also proposed multithreaded version for parallelism [118] (MT Monsoon in 1991). A program is divided in threads, each executed and described sequentially. The system is composed of multiple cores, each capable of executing several threads at the same time. The ISA is extended with two new instruction that allowed fork/join capabilities. On a fork, a new thread was created with an allocated memory frame associated to it. A continuation memory space was used to synchronize data coming back to the join operation. The split-phase transaction was used to synchronize the continuation points.

The Epsilon architecture (1989) [119][120] relies on pure dataflow execution of instructions. However, by grouping together multiple instructions into sequentially executed groups, it is possible to control and reduce the overhead of synchronization. That is, it allows to control the granularity of the instructions. For a grain size of 1 (i.e. one operation per group), the execution is a purely dataflow execution. The larger the grain size the more the execution resembles a Von Numann sequential model. Memory is organized in frames of execution that form a tree, while split-phase transactions enable synchronization of producer and consumer of data. Other architectures such as the EM-4, the EM-X (1995) [121], and the RWC-1 (1994) [122] extended these principles and included pre-fetching mechanisms and token pre-matching.

Other approaches that aim to combine Von Neumann execution and dataflow are closer in nature to the proposal presented in this thesis. Task SuperScalar [123][124] for example uses out of order execution techniques for scheduling of tasks while dependencies determine the order in which tasks are executed. Hardware mechanisms have been proposed in [125] for the implementation of task superscalar. While several of the ideas of this thesis and the SuperCodelet architecture overlap with the Task SuperScalar model, the major difference relies on the view of the system as a hierarchy of Von Neumann machines. At each level we have created a Turing Complete system that does not see the memory space as a contiguous and monolithic element. The concept of the Hierarchical Turing Machine and Hierarchical Von Neuman Architecture are the key differentiating factors of this work.

8.5 Software approaches and other efforts

There has been several programming models for parallelism. Tasking is a parallel programming paradigm that inherits ideas of dataflow models of computation. It uses a graph of operations connected through data and control dependencies. Contrary to dataflow architectures, tasks are described as a large collection of instructions that form a coarser grain scheduling unit. The execution of the code that describes the operations performed by a task is often sequential. Most recently, tasking terminology has also been used in the context of parallel GPGPUs system execution.

Across the most popular tasking mechanisms there is OpenMP Tasking [58], Legion [60], OCR [61], Habanero [126], Argobots [127], and many others. Tasking and dataflow inspired execution has also influenced the design of other important runtimes such as Tensorflow for ML and AI.

On the software world it is important to mention the HiHAT effort [128] which aims to unify the tasking interfaces between hardware and software in order to achieve coordination between the different execution models and programming models. NVIDIA has also included CUDA graphs which allow users to define control dependencies as a graph prior to the execution of the computation. These dependencies can be spawned across host and device, and they rely on hardware runtime to execute [64].

The main difference of the introduced approach is that it does not aim to provide yet another programming model, but instead it tries to re-define the hardware/software interface, by defining an execution model, and its programming interface. The model presented in this work is envisioned through hardware/software co-design, while it also tries to take advantage of performance, productivity (programmability) and portability. This approach parts from the model of computation to the architecture, while it is not intended as a language for the creation of tasks and its corresponding software implemented runtime.

Chapter 9
FUTURE WORK

In order to satisfy the fast evolution of computer systems moving forward, it is necessary to guarantee Performance, Portability and Productivity. This work proposes one solution for a Program Execution Model that aims to nurture these elements. The main objective of the Sequential Codelet Model is to serve as a general model of computation. A unified strategy for the design of computer system infrastructure. The Sequential Codelet Model, as a program execution model, should serve as base for the design of Compilers, languages, systems, operating systems and more.

This work leaves many open questions and avenues to move forward. First, What are the appropriate strategies for translating already existing code into the Sequential Codelet Model? By find better and more complicated benchmarks a more valuable analysis of these strategies should be possible. It is necessary to learn from user's experience what are the translation strategies that allows for a programmer to efficiently use the SCM abstraction. Such benchmarks should allow to take advantage of re-utilization of registers of the upper levels, instead of limit to a single Codelet operations. Furthermore, the use of heterogeneous computation between CPU, GPU and other architectures is really important. While this work demonstrates that it is possible to run the Sequential Codelet Model in GPUs, the results are far from competitive. I believe that heterogeneous is a key aspect of future computer systems. Other ideas to combine the Sequential Codelet Model with highly heterogeneous abstract machines such as the 10x10 architecture [112] are avenues to explore moving forward.

Second, How to leverage already existing programming models to adapt the hierarchical organization of the SCM abstraction? The definition of a higher level programming language or abstraction should be possible thanks to the extensive evolution of programming languages. We envision an initial extension that uses a directive based approach to be able to define Codelets at the different levels.

Furthermore, hardware emulation is also a critical path for the development of systems base on the Sequential Codelet Model. A Hardware/Software co-design strategy has been recently proposed through the DEMAC system [129][130][131]. This open source project aims to create a platform for the definition of a unified architecture of the computer systems. An FPGA based implementation on a system that uses scratchpad-like memory systems would could serve also as an initial realization of the SuperCodelet architecture, while being able to obtain metrics on system utilization, performance, overhead and others. These avenues are currently being explored. This work leaves the necessary building blocks for a hardware implementation.

When exploring hardware architectures that implement the SCM. It is possible to explore energy awareness mechanisms that dynamically determines the organization of the machine. Annotation of Codelet and Codelet specialization for different optimization objectives could provide a way to adapt the execution of the program to user defined optimization metrics (e.g. energy, performance or reliability). These ideas have already been explored in the context of the Codelet Model [111], however, they need to be extended to consider the hierarchical organization of the Sequential Codelet Model.

Extensions to the Codelet Model needs to be realized for exploring Single Program Multiple Data abstractions, event driven and interruption based execution of Codelets, Atomic operations across execution units of the same level, and, in general, how to adapt current concepts of parallelism into the Sequential Codelet Model.

Additionally, further exploring the upper levels of the hierarchy is also necessary. The use of the SCM model into distributed computation will allow for a description of programs that execute on large HPC infrastructures. Furthermore, the use of the SCM model may allow for better concurrent use on HPC ecosystems. Currently, batch schedulers require total allocation of the hardware resources. By using strategies similar to those used in Operating systems, it may be possible to have multiple SCM programs running concurrently under the same environment, without interference on one another.

As we go higher in the abstraction, we also must ask what is the role of the Internet in the Sequential Codelet Model? The Internet has allowed cloud computing and computational services to be more available. The use of APIs allows for a programmer to offload computation to servers by performing application specific operations. What would happen if an API is created for cloud computing with semantics similar to those used in the SCM machine?

After almost a century, Turing's computation model has demonstrated to have a tremendous influence in computer systems and the possible applications these machines can solve. Moving forward in the next 100 years of computers it is still an open problem: will the Turing Machine, or any of its proposed extensions (including the Hierarchical Turing Machine), be able to meet the challenges of future computation?. A recent example of such challenges have been discussed in private communications within DoE. It is currently believed that reaching over 50000x times speed ups is critical for many scientific and data intensive applications. Is there room for these requirements for Turing based computers? Or is it necessary to find other abstractions?

Chapter 10

CONCLUSIONS

This thesis introduces several concepts towards the definition of a unified model of computation for general purpose, parallel, distributed and heterogeneous computation. First, this thesis defines the Hierarchical Turing Machine. A Multi-tape Turing Machine and model of computation that re-structures the original Turing Machine to consider hierarchical organization of data and computation. Following it creates the hierarchical Von Neumann architecture based on the Hierarchical Turing Machine. This machine model is then used to create the Sequential Codelet Model abstract machine, and the Sequential Codelet Program Execution Model (SCM). Based on the Sequential Codelet Model, this thesis creates the SuperCodelet architecture. By using principles of dataflow and design strategies similar to those used in Out of Order execution engines, an architecture for distributed/parallel and heterogeneous computation is defined.

The Sequential Codelet Model is a program execution model that explains how a program interacts with a parallel, distributed and heterogeneous architecture that implements an abstract machine organization. The Sequential Codelet Model envisions the machine as a hierarchical composition of Von Neumann systems. This hierarchy maps to the natural organization of memory in computer systems. At the bottom, commodity architectures are used as execution engines of a 5 stages pipeline built around the concept of a Core. The Core can be any currently existing architecture. The 5 stages pipeline forms the first level of the hierarchy. Level two is formed by using Level 1 as the execution stage. As we progress in the hierarchy, the level below becomes the execution engine of the 5 stages pipeline in the level above. The hierarchy spans from the concept of core, as currently used in the literature, to the aggregation of cores in sockets (level 1), aggregation of sockets into nodes (level 2), aggregation of nodes into clusters (level 3), and so on. In the Sequential Codelet Model parallelism is achieved by means similar to those used in Instruction Level Parallelism for sequential architectures.

An SCM program is also hierarchically organized to map to the hierarchical abstract machine organization. A level of the program maps to a level of the machine. A program is a sequential description of operations, similar to current assembly code. The operands in the program correspond to memory locations (akin registers) at the level in which the program is executed. Different to assembly code, instructions can be user defined, determined by a program that runs in the level below. Instructions are referred to Codelets. A Codelet belongs to a level as an instruction of that level, but it is implemented as a program of the level below.

In order to understand the behavior of the SuperCodelet architecture and the Sequential Codelet Model, this thesis creates an emulator, namely SCMUlate. This program is composed of an interpreter and a runtime. The interpreter reads SCM programs and translate them into the runtime API. The runtime executes on commodity x86 CPU and Gen9 GPU hardware and it uses different configurations (different machine implementations). Two microbenchmarks: Matrix Multiplication and Vector addition have been used in SCMUlate. These benchmarks have been selected as important kernels for HPC and the characteristics of future computational workloads, as learned throughout years of experience interacting with application teams from the ECP project. These benchmarks provide an early evaluation and justification for the Sequential Codelet Model.

Early evaluation through SCMUlate demonstrates that it is possible to obtain a hardware implementations that achieve parallel execution of codes by means of the SCM abstraction. Furthermore, that it is possible to create different system implementations capable of executing the same abstraction. Therefore, by keeping the same program execution model Application Programming Interface (API), it is possible to port the execution of code into different implementations, while allowing to prioritize optimizations in different levels of the abstraction. Furthermore, results presented in this work show that, the Sequential Codelet Model abstraction could lead to compiler optimization techniques, as well as static analysis of computational and memory complexity of applications. all under the umbrella of the same Program Execution Model.

An emulation runtime on commodity Intel architectures does not necessarily reflect the intention of the Sequential Codelet Model as a model of computation. In fact, it provides the wrongful idea that SCMUlate is yet another parallel programming model or framework akin OpenMP, Legion, OCR or others. Instead, the Sequential Codelet Model represents a way of thinking and mapping computation to hardware resources, which is expected to be realized in hardware, and to be extended through software tools. Moreover, a software approach provides additional overhead, and reduces the capacity of compiler optimizations, that could potentially benefit the final performance of the program. We will do a more in depth analysis of these results in the following subsection.

Through the use of SCMULate, this thesis demonstrates that it is possible to achieve scalable execution of code in multicore and heterogeneous architectures. The sequential semantics considerably improves programmability. Furthermore, the static definition of Codelets at a given level allows for already existing Compiler techniques and algorithmic analysis tools to be used to improve the execution time of a SCM program, at each level of the hierarchy. The analysis of the results obtained by SCMULate result in a mathematical formulation that works as basis for the creation of actual hardware implementations of the SCM model.

BIBLIOGRAPHY

[1] J. L. Hennessy and D. A. Patterson, “A new golden age for computer architecture,” *Commun. ACM*, vol. 62, pp. 48–60, Jan. 2019.

[2] A. M. Turing, “On computable numbers, with an application to the Entscheidungsproblem,” *Proceedings of the London Mathematical Society*, vol. 2, no. 42, pp. 230–265, 1936.

[3] J. v. Neumann, “First draft of a report on the EDVAC,” tech. rep., 1945.

[4] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, “Architecture of the IBM System/360,” *IBM J. Res. Dev.*, vol. 8, p. 87–101, Apr. 1964.

[5] M. Bohr, “A 30 year retrospective on dennard’s MOSFET scaling paper,” *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, 2007.

[6] P. Qu, J. Yan, Y.-H. Zhang, and G. R. Gao, “Parallel turing machine, a proposal,” *Journal of Computer Science and Technology*, vol. 32, pp. 269–285, Mar 2017.

[7] J. B. Dennis, “Programming generality, parallelism and computer architecture,” in *IFIP Congress*, 1968.

[8] J. B. Dennis, “A parallel program execution model supporting modular software construction,” 1997.

[9] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, “In-datacenter performance analysis of a tensor processing unit,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA ’17, (New York, NY, USA), pp. 1–12, ACM, 2017.

[10] A. S. Cassidy, J. Sawada, P. Merolla, J. V. Arthur, R. Alvarez-Icaza, F. Akopyan, B. L. Jackson, and D. S. Modha, "Truenorth: A high-performance, low-power neurosynaptic processor for multi-sensory perception, action, and cognition," 2016.

[11] Top 500, "Top 500 list. june 2018 list highlights." https://www.top500.org/lists/2018/06/highlights/.

[12] R. M. Russell, "The cray-1 computer system," *Commun. ACM*, vol. 21, p. 63–72, Jan. 1978.

[13] S. Chandra, J. R. Larus, and A. Rogers, "Where is time spent in message-passing and shared-memory programs?," *SIGOPS Oper. Syst. Rev.*, vol. 28, p. 61–73, Nov. 1994.

[14] H. Shan and J. P. Singh, "A comparison of mpi, shmem and cache-coherent shared address space programming models on a tightly-coupled multiprocessors," *International Journal of Parallel Programming*, vol. 29, no. 3, pp. 283–318, 2001.

[15] Y. Yan, J. R. Hammond, C. Liao, and A. E. Eichenberger, "A proposal to openmp for addressing the cpu oversubscription challenge," in *OpenMP: Memory, Devices, and Tasks* (N. Maruyama, B. R. de Supinski, and M. Wahib, eds.), (Cham), pp. 187–202, Springer International Publishing, 2016.

[16] M. Raynal and A. Schiper, "A suite of definitions for consistency criteria in distributed shared memories," *Annales des Telecommunications/Annals of Telecommunications*, vol. 52, no. 11-12, pp. 652–661, 1997.

[17] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, p. 690–691, Sept. 1979.

[18] S. R. Walli, "The POSIX family of standards," *StandardView*, vol. 3, no. 1, pp. 11–17, 1995.

[19] E. A. Lee, "The problem with threads," *Computer*, vol. 39, pp. 33–42, May 2006.

[20] D. Chisnall, "C is not a low-level language," *Queue*, vol. 16, pp. 10:18–10:30, Apr. 2018.

[21] J. M. Monsalve Diaz, S. S. Pophale, O. R. Hernandez, D. Bernholdt, and S. Chandrasekaran, "OpenMP 4.5 validation and verification suite for device offload," 8 2018.

[22] M. D. Jose M, *OpenMP 4.5 validation and verification testsuite: design and implementation for offloading features*. 2020.

[23] Johannes Doerfert, "SOLLVE: OpenMP for HPC and Exascale." https://www.exascaleproject.org/highlight/sollve-openmp-for-hpc-and-exascale/.

[24] A. Church, “An unsolvable problem of elementary number theory,” *American Journal of Mathematics*, vol. 58, pp. 345–363, Apr. 1936.

[25] A. Church, “A note on the entscheidungsproblem.,” *J. Symb. Log.*, vol. 1, no. 1, pp. 40–41, 1936.

[26] C. Petzold, *The Annotated Turing: A Guided Tour Through Alan Turing’s Historic Paper on Computability and the Turing Machine.* Wiley Publishing, 2008.

[27] B. E. Carpenter and R. W. Doran, *A. M. Turing’s ACE Report of 1946 and Other Papers.* USA: Massachusetts Institute of Technology, 1986.

[28] F. J. Corbató and V. A. Vyssotsky, “Introduction and overview of the multics system,” in *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, AFIPS ’65 (Fall, part I), (New York, NY, USA), p. 185–196, Association for Computing Machinery, 1965.

[29] R. C. Daley and J. B. Dennis, “Virtual memory, processes, and sharing in multics,” *Commun. ACM*, vol. 11, p. 306–312, May 1968.

[30] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, “One-level storage system,” *IRE Transactions on Electronic Computers*, vol. EC-11, no. 2, pp. 223–235, 1962.

[31] H. N. S. Aldin, H. Deldari, M. H. Moattar, and M. R. Ghods, “Consistency models in distributed systems: A survey on definitions, disciplines, challenges and applications,” *CoRR*, vol. abs/1902.03305, 2019.

[32] F. Pong and M. Dubois, “Verification techniques for cache coherence protocols,” *ACM Comput. Surv.*, vol. 29, p. 82–126, Mar. 1997.

[33] P. Stenstrom, “A survey of cache coherence schemes for multiprocessors,” *Computer*, vol. 23, no. 6, pp. 12–24, 1990.

[34] Arvind and R. A. Iannucci, “Two fundamental issues in multiprocessing,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 295 LNCS, pp. 61–88, 1988.

[35] R. M. Karp and R. E. Miller, “Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing,” *SIAM Journal on Applied Mathematics*, vol. 14, no. 6, pp. 1390–1411, 1966.

[36] J. B. Dennis and D. Misunas, “A preliminary architecture for a basic data flow processor,” in *Proceedings of the 2nd Annual Symposium on Computer Architecture, Houston, TX, USA, December 1974*, pp. 126–132, 1974.

[37] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium* (B. Robinet, ed.), (Berlin, Heidelberg), pp. 362–376, Springer Berlin Heidelberg, 1974.

[38] J. Rumbaugh, "A data flow multiprocessor," *IEEE Transactions on Computers*, vol. C-26, pp. 138–146, Feb 1977.

[39] Arvind and R. S. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," *IEEE Transactions on Computers*, vol. 39, pp. 300–318, March 1990.

[40] G. M. Papadopoulos and D. E. Culler, "Monsoon: An explicit token-store architecture," *SIGARCH Comput. Archit. News*, vol. 18, p. 82–91, May 1990.

[41] A. L. Davis, "The architecture and system method of DDM1," in *Proceedings of the 5th annual symposium on Computer architecture - ISCA '78*, vol. 1, (New York, New York, USA), pp. 210–215, ACM Press, 1978.

[42] R. Vedder and D. Finn, "The Hughes Data Flow Multiprocessor," *ACM SIGARCH Computer Architecture News*, vol. 13, pp. 324–332, jun 1985.

[43] A. Plus, D. Comte, and J. C. Syre, "Lau system architecture: a parallel data-driven processor based on single assignment," 1976.

[44] J. B. Dennis, G. R. Gao, and V. Sarkar, "Determinacy and repeatability of parallel program schemata," in *2012 Data-Flow Execution Models for Extreme Scale Computing*, pp. 1–9, 2012.

[45] J. E. Rodrigues and J. E. Rodriguez Bezos, "A graph model for parallel computations," tech. rep., USA, 1969.

[46] S. S. Patil, *Closure Properties of Interconnections of Determinate Systems*, p. 107–116. New York, NY, USA: Association for Computing Machinery, 1970.

[47] R. M. Keller, "An approach to determinacy proofs," 1978.

[48] P. J. Denning, *On the Determinacy of Schemata*, p. 143–147. New York, NY, USA: Association for Computing Machinery, 1970.

[49] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion, "Hybrid dataflow/von-Neumann architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1489–1509, 2014.

[50] B. Robič, J. Šilc, and T. Ungerer, "Beyond dataflow," *Journal of Computing and Information Technology*, vol. 8, no. 2, pp. 89–101, 2000.

[51] J. Silc, B. Robič, and T. Ungerer, “Asynchrony in parallel computing: from dataflow to multithreading,” in *Progress in computer research*, pp. 459–469, IEEE, 1997.

[52] J. Silc, B. Robic, and T. Ungerer, “Processor Architecture: From Dataflow to Superscalar and Beyond,” no. November 2014, p. 389, 1999.

[53] G. E. Moore, “Cramming more components onto integrated circuits,” *Electronics*, vol. 38, April 1965.

[54] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, “Design of ion-implanted MOSFET’s with very small physical dimensions,” *IEEE Journal of Solid-State Circuits*, vol. 9, pp. 256–268, Oct 1974.

[55] J. E. Thornton, *Design of a Computer—The Control Data 6600*. Scott Foresman & Co, 1970.

[56] N. Vasseghi, K. Yeager, E. Sarto, and M. Seddighnezhad, “200-mhz superscalar risc microprocessor,” *IEEE Journal of Solid-State Circuits*, vol. 31, no. 11, pp. 1675–1686, 1996.

[57] R. M. Robert Tomasulo, “An efficient algorithm for exploiting multiple arithmetic units,” *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[58] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, “The design of OpenMP tasks,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, 2009.

[59] J. Suettlerlein, S. Zuckerman, and G. R. Gao, “An implementation of the codelet model,” in *Euro-Par 2013 Parallel Processing* (F. Wolf, B. Mohr, and D. an Mey, eds.), (Berlin, Heidelberg), pp. 633–644, Springer Berlin Heidelberg, 2013.

[60] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, “Legion: Expressing locality and independence with logical regions,” in *SC ’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2012.

[61] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, Min Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, and N. Vrvilo, “The open community runtime: A runtime system for extreme scale computing,” in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2016.

[62] H. H. Hum, O. Maquelin, K. B. Theobald, X. Tian, G. R. Gao, and L. J. Hendren, “A study of the EARTH-MANNA multithreaded system,” *International Journal of Parallel Programming*, vol. 24, no. 4, pp. 319–348, 1996.

[63] R. S. Nikhil, G. M. Papadopoulos, and Arvind, "T: A multithreaded massively parallel architecture," in *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*, pp. 156–167, 1992.

[64] NVIDIA, "Cuda graphs." https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-graphs.

[65] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero, "Task superscalar: An out-of-order task pipeline," *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pp. 89–100, 2010.

[66] R. H. Halstead, "Multilisp: A language for concurrent symbolic computation," *ACM Trans. Program. Lang. Syst.*, vol. 7, p. 501–538, Oct. 1985.

[67] Wen-Yen Lin and Jean-Luc Gaudiot, "I-structure software cache: a split-phase transaction runtime cache system," in *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Technique*, pp. 122–126, 1996.

[68] Arvind, R. S. Nikhil, and K. K. Pingali, "I-structures: Data structures for parallel computing," *ACM Trans. Program. Lang. Syst.*, vol. 11, p. 598–632, Oct. 1989.

[69] J. B. Dennis, "Fresh breeze: A multiprocessor chip architecture guided by modular programming principles," *SIGARCH Comput. Archit. News*, vol. 31, p. 7–15, Mar. 2003.

[70] I. T. A. Lee, S. Boyd-Wickizer, Z. Huang, and C. E. Leiserson, "Using memory mapping to support cactus stacks in work-stealing runtime systems," *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, vol. 2010, pp. 411–420, 2010.

[71] Intel, "First the tick , now the tock : Intel ® microarchitecture (nehalem) introducing a new dynamically and design-scalable microarchitecture," 2009.

[72] G. Gao, J. Suetterlein, and S. Zuckerman, "Toward an Execution Model for Extreme-Scale Systems - Runnemede and Beyond." Technical Memo, April 2011.

[73] D. E. Culler, K. E. Schauser, and T. von Eicken, "Two fundamental limits on dataflow multiprocessing," tech. rep., USA, 1992.

[74] J. Arteaga, S. Zuckerman, and G. R. Gao, "Multigrain parallelism: Bridging coarse-grain parallel programs and fine-grain event-driven multithreading," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 799–808, 2017.

[75] J. Monsalve, K. Harms, K. Kalyan, and G. Gao, "Sequential codelet model of program execution. a super-codelet model based on the hierarchical turing machine.," in *2019 IEEE/ACM Third Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)*, pp. 1–8, 2019.

[76] M. Yannakakis, "Hierarchical state machines," in *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics* (J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses, and T. Ito, eds.), (Berlin, Heidelberg), pp. 315–330, Springer Berlin Heidelberg, 2000.

[77] W. Zhang, B. Gao, J. Tang, P. Yao, S. Yu, M.-F. Chang, H.-J. Yoo, H. Qian, and H. Wu, "Neuro-inspired computing chips," *Nature Electronics*, vol. 3, pp. 371–382, 07 2020.

[78] K. Livingston, A. Landwehr, J. M. Diaz, S. Zuckerman, B. Meister, and G. R. Gao, "Energy avoiding matrix multiply," in *Languages and Compilers for Parallel Computing - 29th International Workshop, LCPC 2016, Rochester, NY, USA, September 28-30, 2016, Revised Papers*, pp. 55–70, 2016.

[79] G. R. Gao and V. Sarkar, "Location consistency-a new memory model and cache consistency protocol," *IEEE Trans. Comput.*, vol. 49, p. 798–813, Aug. 2000.

[80] J. B. Dennis, "A parallel program execution model supporting modular software construction," in *Proceedings. Third Working Conference on Massively Parallel Programming Models (Cat. No.97TB100228)*, pp. 50–60, 1997.

[81] S. Raskar, T. Applencourt, K. Kumaran, and G. Gao, "Position paper: Extending codelet model for dataflow software pipelining using software-hardware co-design," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, pp. 640–645, July 2019.

[82] NVIDIA, "NVIDIA Tesla V100 GPU Architecture. The world's most advanced data center GPU." http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, August 2017.

[83] Intel, "The compute architecture of intel processor graphics Gen9," Aug 2015.

[84] Jose M Monsalve Diaz, "SCMUlate Emulator Source Code." https://github.com/josemonsalve2/SCM.

[85] P. T. Inc., "Collaborative data science," 2015.

[86] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, (New York, NY, USA), p. 63–74, Association for Computing Machinery, 1994.

[87] J. Xue, *Loop Tiling for Parallelism.* USA: Kluwer Academic Publishers, 2000.

[88] Intel Corporation., "Intel oneAPI: A Unified X-Architecture Programming Model." https://software.intel.com/content/www/us/en/develop/tools/oneapi.html.

[89] Dennis, "Data flow supercomputers," *Computer*, vol. 13, no. 11, pp. 48–56, 1980.

[90] G. M. Papadopoulos and D. E. Culler, "Monsoon: An explicit token-store architecture," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, (New York, NY, USA), p. 82–91, Association for Computing Machinery, 1990.

[91] N. Takahashi and M. Amamiya, "A data flow processor array system," *ACM SIGARCH Computer Architecture News*, vol. 11, pp. 243–250, jun 1983.

[92] M. Kishi, H. Yasuhara, and Y. Kawamura, "DDDP-a Distributed Data Driven Processor," *ACM SIGARCH Computer Architecture News*, vol. 11, pp. 236–242, jun 1983.

[93] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester prototype dataflow computer," *Communications of the ACM*, vol. 28, no. 1, pp. 34–52, 1985.

[94] N. Ito, M. Sato, E. Kuno, and K. Rokusawa, "The architecture and preliminary evaluation results of the experimental parallel inference machine PIM-D," *ACM SIGARCH Computer Architecture News*, vol. 14, pp. 149–156, jun 1986.

[95] K. E. Fleming, K. D. Glossop, S. C. Steely, J. Tang, and A. G. Gara, "Processors, methods, and systems with a configurable spatial accelerator," Feb 2020.

[96] Y. N. Patt, W. mei Hwo, and M. C. Shebanow, "HPS, a New Microarchitecture: Rationale and Introduction.," *MICRO: Annual Microprogramming Workshop*, pp. 103–108, 1985.

[97] W.-r. Hwu and Y. N. Part, "H P S m , a High Performance Restricted Data Flow Architecture Having Minimal Functionality Computer Science Division , University of California , Berkeley," pp. 297–306, 1986.

[98] J. E. Smith and A. R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors," *IEEE Transactions on Computers*, vol. 37, no. 5, pp. 562–573, 1988.

[99] T. Agerwala and J. Cocke, "High performance reduced instruction set processors," vol. 4, 1987.

[100] E. Bloch, "The engineering design of the stretch computer," *Proceedings of the Eastern Joint Computer Conference, IRE-AIEE-ACM 1959*, pp. 48–58, 1959.

[101] M. D. Smith, M. S. Lam, and M. A. Horowitz, "Boosting beyond static scheduling in a superscalar processor," *Conference Proceedings - Annual Symposium on Computer Architecture*, pp. 344–354, 1990.

[102] Lee and Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, vol. 17, pp. 6–22, jan 1984.

[103] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting Speculative Execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[104] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," *Proceedings of the 27th USENIX Security Symposium*, pp. 973–990, 2018.

[105] J. A. Fisher, "Very Long Instruction Word architectures and the ELI-512," in *Proceedings of the 10th annual international symposium on Computer architecture - ISCA '83*, (New York, New York, USA), pp. 140–150, ACM Press, 1983.

[106] "Systolic arrays (for VLSI)," in *Sparse Matrix Proceedings 1978*, vol. 1, pp. 256–282, Society for industrial and applied mathematics, 1979.

[107] L. Tucker and G. Robertson, "Architecture and applications of the Connection Machine," *Computer*, vol. 21, pp. 26–38, aug 1988.

[108] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading : Maximizing On-Chip Parallelism x x full issue slot empty issue slot horizontal waste = 9 slots," 1995.

[109] M. K. Smotherman, E. H. Sussenguth, and R. J. Robelen, "The IBM ACS Project," *IEEE Annals of the History of Computing*, vol. 38, no. 1, pp. 60–74, 2016.

[110] V. Cavé, R. Clédat, P. Griffin, A. More, B. Seshasayee, S. Borkar, S. Chatterjee, D. Dunning, and J. Fryman, "Traleika glacier: A hardware-software co-designed approach to exascale computing," *Parallel Computing*, vol. 64, pp. 33 – 49, 2017. High-End Computing for Next-Generation Scientific Discovery.

[111] A. Landwehr, "An experimental exploration of self-aware systems for exascale architectures," 2016.

[112] A. A. Chien, A. Snavely, and M. Gahagan, "10x10: A general-purpose architectural approach to heterogeneity and energy efficiency," *Procedia Computer Science*, vol. 4, pp. 1987 – 1996, 2011. Proceedings of the International Conference on Computational Science, ICCS 2011.

[113] A. S. Cassidy, J. Sawada, P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, F. Akopyan, B. L. Jackson, and D. S. Modha, "TrueNorth: a High-Performance, Low-Power Neurosynaptic Processor for Multi-Sensory Perception, Action, and Cognition," *IBM Research*, vol. Almaden Re, pp. 341–344, 2016.

[114] J. Pei, L. Deng, S. Song, M. Zhao, Y. Zhang, S. Wu, G. Wang, Z. Zou, Z. Wu, W. He, F. Chen, N. Deng, S. Wu, Y. Wang, Y. Wu, Z. Yang, C. Ma, G. Li, W. Han, H. Li, H. Wu, R. Zhao, Y. Xie, and L. Shi, "Towards artificial general intelligence with hybrid Tianjic chip architecture," *Nature*, vol. 572, no. 7767, pp. 106–111, 2019.

[115] B. Robič, J. Šilc, and T. Ungerer, "Beyond dataflow," *Journal of Computing and Information Technology*, vol. 8, no. 2, pp. 89–101, 2000.

[116] J. Silc, B. Robic, and T. Ungerer, "Processor Architecture: From Dataflow to Superscalar and Beyond," no. November 2014, p. 389, 1999.

[117] J. Silc, B. Robič, and T. Ungerer, "Asynchrony in parallel computing: from dataflow to multithreading," in *Progress in computer research*, pp. 459–469, IEEE, 1997.

[118] G. M. Papadopoulos and K. R. Traub, "Multithreading: A revisionist view of dataflow architectures," *Conference Proceedings - Annual Symposium on Computer Architecture*, pp. 342–351, 1991.

[119] V. Grafe and J. Hoch, "The Epsilon-2 multiprocessor system," *Journal of Parallel and Distributed Computing*, vol. 10, pp. 309–318, dec 1990.

[120] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes, "Epsilon Dataflow Processor.," *Conference Proceedings - Annual Symposium on Computer Architecture*, no. 16, pp. 36–45, 1989.

[121] Y. Kodama, H. Sakane, M. Sato, H. Yamana, S. Sakai, and Y. Yamaguchi, "EM-X parallel computer: Architecture and basic performance," *Conference Proceedings - Annual International Symposium on Computer Architecture, ISCA*, pp. 14–23, 1995.

[122] S. Sakai, H. Matsuoka, K. Okamoto, T. Yokota, H. Hirono, Y. Kodama, and M. Sato, "Rwc-1 massively parallel architecture," 1994.

[123] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero, "Task superscalar: An out-of-order task pipeline," *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pp. 89–100, 2010.

[124] Y. Etsion, A. Ramirez, and R. Badia, "Task superscalar: Using processors as functional units," *HotPar*, 2010.

[125] F. Yazdanpanah, D. Jimenez-Gonzalez, C. Alvarez-Martinez, Y. Etsion, and R. M. Badia, “Analysis of the task superscalar architecture hardware design,” *Procedia Computer Science*, vol. 18, pp. 339–348, 2013.

[126] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, “Habanero-Java,” p. 51, 2011.

[127] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castello, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kale, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman, “Argobots: A Lightweight Low-Level Threading and Tasking Framework,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, 2018.

[128] H. H. A. T. Project, “HiHAT: Hierarchical heterogeneous asynchronous tasking project.” https://hihat-wiki.modelado.org/Hierarchical_Heterogeneous_Asynchronous_Tasking.

[129] CAPSL Research Group, “DEMAC1 website.” https://www.capsl.udel.edu//demac_cluster.shtml.

[130] CAPSL Research Group, “DEMAC1 documentation.” https://www.capsl.udel.edu//demac_cluster/documentation/.

[131] D. R. Perdomo, R. Kabrick, J. M. Diaz, S. Raskar, D. Fox, and G. Gao, “DEMAC and CODIR: A whole stack solution for a hw/sw co-design using an mlir codelet model dialect,” May 2020.

Appendix A

COMPLETE MATRIX MULTIPLICATION CODE

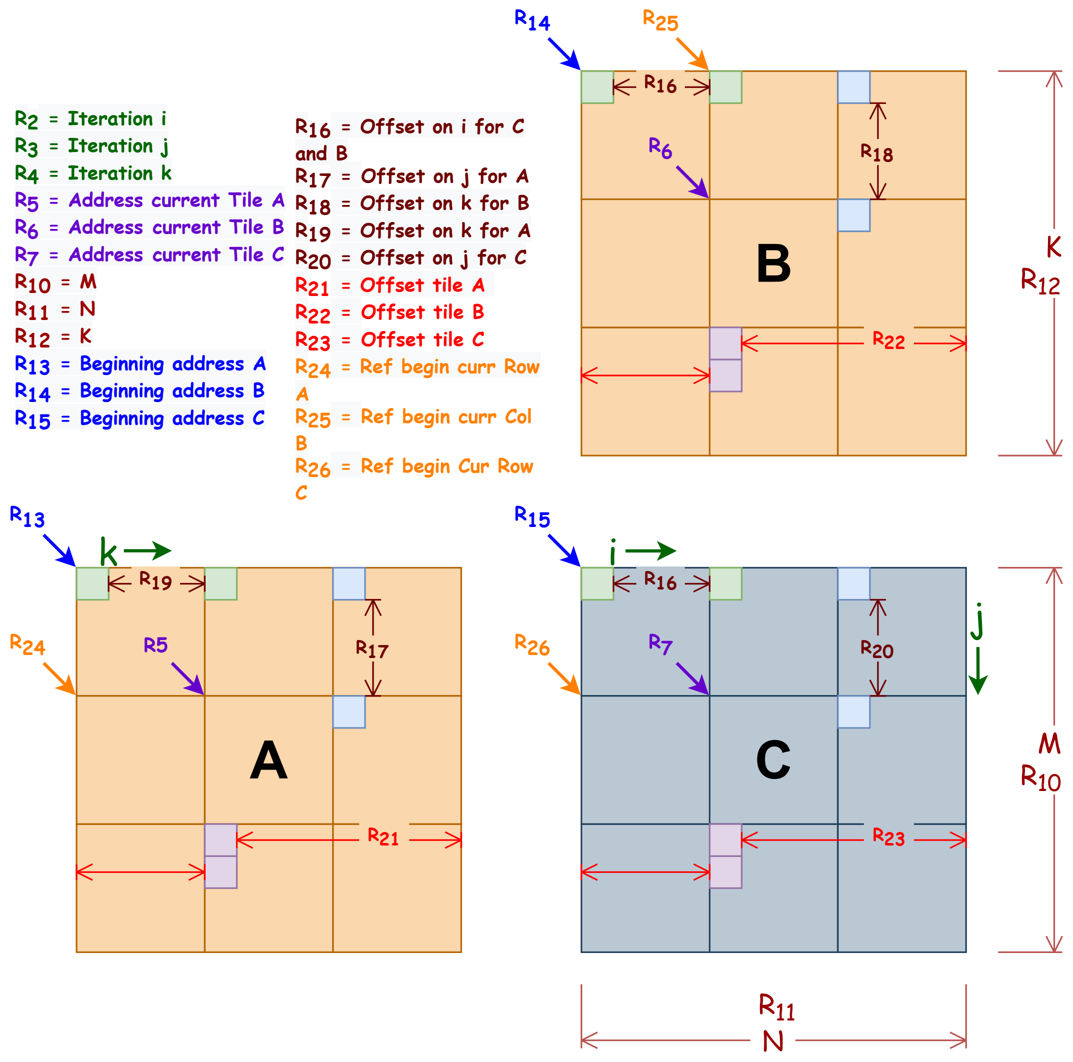


Figure A.1: Guide image for Matrix Multiplication Implementation in SCM. Registers and their corresponding meaning.

In this appendix we show the complete version of the Matrix Multiplication SCM implementation, as discussed in Chapter 7.5. This code is used to multiply any matrix size multiple of the base tile 128x128.

The L2 memory structure is presented in Listing A.1, and L1 SCM Code is presented in Listings A.2-A.7. Finally, Figure A.1 is intended to show a diagram of the meaning of each register with respect to the matrix multiplication algorithm.

```
struct  __attribute__((packed)) l2_memory {
  uint64_t M;
  uint64_t N;
  uint64_t K;
  // MATRIX ADDRESSES
  uint64_t Add_a;
  uint64_t Add_b;
  uint64_t Add_c;
  // OFFSETS A, B, and C in Row Major
  uint64_t Off_cbi;
  uint64_t Off_aj;
  uint64_t Off_bk;
  uint64_t Off_ak;
  uint64_t Off_cj;
  // OFFSETS inside tile in Row Major
  uint64_t Off_a;
  uint64_t Off_b;
  uint64_t Off_c;
  double A[numElementsA];
  double B[numElementsB];
  double C[numElementsC];
};
```

Listing A.1: L2 Memory structure for Matrix Multiplication on SCM

```
LDIMM R64B_1, 0; // Just zero
// MATRIX DIMMENSIONS
LDOFF R64B_10, R64B_1, 0;    // Load M in tiles
LDOFF R64B_11, R64B_1, 8;    // Load N in tiles
LDOFF R64B_12, R64B_1, 16;   // Load K in tiles
// MATRIX ADDRESSES IN MEMORY
LDOFF R64B_13, R64B_1, 24;   // Load Address_a
LDOFF R64B_14, R64B_1, 32;   // Load Address_b
LDOFF R64B_15, R64B_1, 40;   // Load Address_c
// OFFSETS Tiles
LDOFF R64B_16, R64B_1, 48;   // Load Off_cbj in bytes
LDOFF R64B_17, R64B_1, 56;   // Load Off_aj in bytes
LDOFF R64B_18, R64B_1, 64;   // Load Off_bk in bytes
LDOFF R64B_19, R64B_1, 72;   // Load Off_ak in bytes
LDOFF R64B_20, R64B_1, 80;   // Load Off_cj in bytes
// Offsets for third arg of matmul
LDOFF R64B_21, R64B_1, 88;   // Load Off_a in elements
LDOFF R64B_22, R64B_1, 96;   // Load Off_b in elements
LDOFF R64B_23, R64B_1, 104;   // Load Off_c in elements
// ITERATION VARIABLES
LDIMM R64B_2, 0; // For i iteration variable. Tile by tile
LDIMM R64B_3, 0; // For j iteration variable. Tile by tile
LDIMM R64B_4, 0; // For k iteration variable. Tile by tile
// Offset pointing to the first value of the tile in mem
LDIMM R64B_5, 0; // For A_off_cnt iteration variable.
LDIMM R64B_6, 0; // For B_off_cnt iteration variable.
LDIMM R64B_7, 0; // For C_off_cnt iteration variable.
```

Listing A.2: Matrix Multiplication NxM tiles: C = C + A*B. Constant declaration.

```
ADD R64B_5, R64B_5, R64B_13; // Beginning of A
ADD R64B_6, R64B_6, R64B_14; // beginning of B

ADD R64B_7, R64B_7, R64B_15; // Beginning of C

// References to the beginning of row to restart inner counter
ADD R64B_24, R64B_24, R64B_13; // ref Beginning row in A
ADD R64B_25, R64B_25, R64B_14; // ref Beginning row in B
ADD R64B_26, R64B_26, R64B_15; // ref Beginning row in C

// We are using j for the rows (A and C) and i for the cols (B and C)

loop_j:
  BREQ R64B_3, R64B_10, after_loop_j; // if (j == M) jump out of loop
  ADD R64B_3, R64B_3, 1; // j++
  loop_k:
    BREQ R64B_4, R64B_12, after_loop_k; // if (k == K) jump out of loop
    ADD R64B_4, R64B_4, 1; // k++

    // Load the tile of A
    COD LoadSqTile_2048L R2048L_1, R64B_5, R64B_21;
```
Listing A.3: Matrix Multiplication NxM tiles: C = C + A*B. Loop j and k

```
  loop_i:
    BREQ R64B_2, R64B_11, after_loop_i; // if (i == N) jump
        out of loop
    ADD R64B_2, R64B_2, 1; // i++

    // Load tiles of B and C
    COD LoadSqTile_2048L R2048L_2, R64B_6, R64B_22;
    COD LoadSqTile_2048L R2048L_3, R64B_7, R64B_23;

    // Do actual MM
    COD MatMult_2048L R2048L_3, R2048L_1, R2048L_2;

    // Store partial result of C
    COD StoreSqTile_2048L R2048L_3, R64B_7, R64B_23;

    // Move on i tile by tile over B and C
    // Move B along i. Increase by tile size in row major
    ADD R64B_6, R64B_6, R64B_16;
    // Move C along i. Increase by tile size in row major
    ADD R64B_7, R64B_7, R64B_16;

    JMPLBL loop_i; // go to beginning of loop

  after_loop_i:
```

Listing A.4: Matrix Multiplication NxM tiles: C = C + A*B. Loop i

```
    // Advance A, tile by tile, over K.
    // *A + Off_ak (Increase by tile size in row major)
    ADD R64B_5, R64B_5, R64B_19;

    // Advance B, tile by tile, over K. Reset the
    // pointer that moves B, tile by tile, over i direction
    // Set to beginning of B for new k
    ADD R64B_25, R64B_25, R64B_18;
    // Reset inner count for B along i
    LDIMM R64B_6, 0;
    // Set count to beginning
    ADD R64B_6, R64B_6, R64B_25;

    // Reset the counter for C, will stay in the same j
    LDIMM R64B_7, 0;
    // Set to beginning of C in same j
    ADD R64B_7, R64B_7, R64B_26;

    // Reset iteration counter
    LDIMM R64B_2, 0;

    JMPLBL loop_k;

  after_loop_k:
```

Listing A.5: Matrix Multiplication NxM tiles: C = C + A*B. Iteration variable increments loop i

```
// Advance A over j a whole tile, and Reset the
// pointer that moves A tile by tile over k
// Move A along j. (increase by tile size*tile size*K in
   row major)
ADD R64B_24, R64B_24, R64B_17;
// Reset inner count for A
LDIMM R64B_5, 0;
// Set to beginning of A
ADD R64B_5, R64B_5, R64B_24;

// Reset B to the beginning of the B matrix, then reset
   counters
LDIMM R64B_25, 0;
// Move ref to beginning of B
ADD R64B_25, R64B_25, R64B_14;
// Reset inner count for B along i
LDIMM R64B_6, 0;
// Set to beginning of matrix B
ADD R64B_6, R64B_6, R64B_25;
```

Listing A.6: Matrix Multiplication NxM tiles: C = C + A*B. Iteration variable increments loop k

```
  // Advance C over j a whole tile, and reset pointer that
     moves
  // C tile by tile over i
  ADD R64B_26, R64B_26, R64B_20; // Move C along j
  LDIMM R64B_7, 0; // Reset inner count C
  ADD R64B_7, R64B_7, R64B_26; // Set to beginning of C at j

  // Reset iteration counter
  LDIMM R64B_4, 0;

  JMPLBL loop_j;
after_loop_j:

COMMIT;
```

Listing A.7: Matrix Multiplication NxM tiles: C = C + A*B. Iteration variable increment loop j