Sequential Codelet Model for Parallel Execution

Jose M Monsalve Diaz
Guang R. Gao

CAPSL Technical Memo 132
May, 2019

Copyright © 2019 CAPSL at the University of Delaware
# Contents

1 Introduction ................................................. 4

2 Background .................................................. 6
   2.1 The Universal Turing Machine ......................... 6
   2.2 Von Neumann Architecture ............................. 7
   2.3 Instruction Set Architectures .......................... 8
   2.4 Program Execution Models .............................. 8

3 Computer Systems, a myriad of models .................. 9

4 Sequential Codelet Model .................................. 10
   4.1 A hierarchical Turing Machine ....................... 10
   4.2 The Sequential Codelet Abstract Machine .......... 11
   4.3 A convolution example ................................. 12
      4.3.1 Language convention ............................. 14
      4.3.2 Three level program description ............... 14

5 Achieving Parallelism ..................................... 18
   5.1 Superscalar ............................................ 18
   5.2 Out of order execution ............................... 18
   5.3 Register renaming ................................... 18
   5.4 Heterogeneous architectures .......................... 18
   5.5 Compiler techniques .................................. 19

6 Conclusions .................................................. 19
Abstract

The last two decades has been fascinating for parallel computation and parallel programming. Nowadays, there is no doubt of the importance of parallelism for advancing the limit of what computers can do. Yet, it still seems as if there is no parallel programming model or parallel execution model that has been satisfying enough to be widely adapted by application developers, portable across multiple architectures and systems, and easy to reason about to reduce programming errors. This paper presents the Sequential Codelet Model (SCM), an extension of the previously defined Codelet Model. The Sequential Codelet Model highly borrows from sequential execution of code and low level programming models for single core architectures. It allows using the large experience obtained throughout more than 50 years of sequential computation such as compilation techniques and ILP hardware optimizations. The major observation that lead to proposing the SCM is that single core architectures are now parallel machines, yet the user is not concerned with possible side effects of its execution. This is possible due to the well defined behavior of the instructions exposed by the sequential execution model through the Instruction Set Architecture. The SCM describes: 1) A description of the proposed hierarchical Turing Machine, where each level uses an infinite storage tape for the level above, and it uses the level below to compute state behavior and transition. 2) An architectural machine model based on the hierarchical Turing Machine that can be seen as a hierarchical organization of Von Neumann-like architectures, and 3) a description of a program defined for this machine model as well as its execution. At each level a five stage pipeline is defined where the execution stage is implemented using the level below as the operational unit (e.g. acting as ALU and FP of current single core architectures). An operation at one level is defined as a Codelet, which is described as a sequence of operations supported in the level below. Memory for a level is treated as register file for the operations in that level, and as global memory of the level below. A program in the SCM is expressed in a hierarchical manner that maps to such machine model.

1 Introduction

Computer systems have been around for almost eight decades. Although, current machines are not comparable in many forms to the original computer systems, current machines are still based on the most fundamental model of computation of all times: The Turing Machine model, proposed by Alan Turing in 1936 [?]. The Turing Machine model has driven the evolution of hardware and software to what it is today, and it has created the scientific paradigm that comprises the majority of the science that surrounds computer systems. Following that, almost 10 years after the definition of the Turing Machine, John Von Neumann wrote the first draft of a report on the EDVAC computer [?] in which he described for the first time the Von Neumann Model. These two models (i.e. the Turing Machine Model and the Von Neumann Architecture) are unquestionably the fundamental basis of all computer systems that exist today.

With the wide adoption of the Von Neumann model, sequential computation became the mainstream programming model for defining instructions for computers. Until recently, sequential computation had a long and successful history. While it is not possible to mention all the
reason for its success, two important aspects were: first, The sequential description and execution of programs that ease human reasoning and debug of software. Second, the adoption of well-defined Instruction Set Architectures. An Instruction Set Architectures (ISA) allows a clear distinction between the architectural model of a computer and its implementation. An ISA defines a vocabulary and an expected behavior of the machine without exposing its implementation details. The adoption of ISAs as a fundamental part of architecture designs permitted a parallel (yet synergistic) evolution of both software and hardware infrastructures. ISAs allowed smooth portability of software across hardware generations, while benefiting from the performance evolution of the hardware. On the hardware side, sequential single core architectures became faster and more optimized without modifying the ISA. While on the software side, a complex ecosystem emerged (e.g. programming languages, algorithms, compilers, runtimes, operating systems and libraries) that targeted these ISAs.

However, hardware evolution rate for single core architectures declined. By the beginning of the 2000s hardware architects started to face physical limitations in their designs. During the last decade, both Dennard’s scaling and Moore’s law have significantly slowed down. Consequently, parallelism became widely adopted and it is now seen as a feasible solution to overcome sequential architecture limitations. Virtually all general purpose computer systems that are found in the market today are parallel machines. At the same time HPC systems are equipped with nodes containing several CPU cores, and even acceleration devices such as GPGPUs, FPGAs or similar. Currently, we are going through what Qi et al, refer to as the “Second Spring” of parallel computing.

Parallel computing research is not a new field. Parallel architectures have been proposed for almost as long as sequential architectures, but its wide adoption only happened until recently. Starting on the 60s and extending until the beginning of the 90s, multiple parallel architectures were proposed. Despite this, interest in parallel computing decreased considerably in the 90s. Several limitations, including programmability and wide adaptation of parallel systems, lead to a cold era in the parallel computing field, mostly pronounced by Ken Kennedy in his article "Is parallel computing dead?".[11]

On the other hand, software infrastructure as well as software developers have been required to adapt to this widely available hardware parallelism. Several approaches have been proposed, but there has not been a convincing and widely adopted parallel programming model, and there is a general perspective that parallel programming is hard. Perhaps the major limitation has been adapting and leveraging the newly available features exposed by parallel systems into a complex and large software infrastructure that was already in place for sequential computation. Nevertheless, parallel computing brings a paradigm changed that even challenges the stability of the aforementioned essential Turing Machine model and Von Neumann models.

Hence, there has been a divergence in the evolution of parallel hardware and parallel soft-

\[1\] Still, multiple ISAs were connected by the same underlying Turing and Von Neumann models providing a smooth transition between ISAs.

\[2\] Here we are excluding Instruction Level Parallelism. We are referring to worker level parallelism (e.g. hardware threads and multi-core processors)
ware. On one side, hardware designers have focused on increasing the number of threads in the hardware. Parallel architectures are mainly dominated by simultaneous multithreading technology (SMT), multi-core technology, and SIMD-like accelerators (e.g. GPGPUs)\(^3\), as well as deepen the memory hierarchy. Yet, each thread still follows a sequential execution of code that uses the aforementioned pillar models. On the other side, software efforts have been focused on how to coordinate and assign work to the different execution threads available in the hardware, while integrating these changes to the state of the art high level programming languages. To this end, several parallel programming models have been proposed such as: Message passing, Fork-Joint, Bulk synchronous parallel (BSP), and tasking (among many others)\([12]\)^4. However, few programming models have been able to be ported into hardware directly, and the ones that have been porter does not seem to be general enough to cover a wide range of programs.

This work considers the need for defining a model of computation across hardware and software for parallel machines that goes beyond sequential computation, as also mentioned by Gao \([5]\), Dennis \([13]\), Chisnall \([14]\) and others. We use the concept of Program Execution Model (PXM), as defined by Gao. The program execution model (PXM) is the basic low-level abstraction of the underlying system architecture upon which our programming model, compilation strategy, runtime system, and other software components are developed. The PXM (and its API) serves as an interface between the architecture and the software. Unlike an ISA, the PXM refers to machine organization at a higher level for a whole class of high-end machines as viewed by the users. It is important to clarify that this proposal does not intend to define yet another tasking scheme to be implemented by a runtime system. Instead it defines a machine architecture and how code should be described and executed for it, while leaving out the problem of higher level programming language abstractions. This is similar to how low level execution of code has no knowledge of, for example, classes of a program that written in the Object Oriented Programming paradigm.

**TODO:** Add synopsis

## 2 Background

### 2.1 The Universal Turing Machine

A description of the The Universal Turing Machine was provided by Alan Turing himself \([?]\) and it is simple and complete enough that it is worth reading and revisiting directly form his words.

**TODO:** Probably add a picture of the Turing Machine here.

We may compare a man in the process of computing a real number to a machine

\(^3\)Interest in other technologies (e.g. FPGAs, Google TPU, and Intel CSA) have also raised, but these architectures have not been widely adapted and will be left out of this discussion.

\(^4\)This work does not intend to survey the different programming models, nor to answer the question of which one is better.
which is only capable of a finite number of conditions $q_1 : q_2 ... q_R$; which will be called "m-configurations". The machine is supplied with a "tape" (the analogue of paper) running through it, and divided into sections (called "squares") each capable of bearing a "symbol". At any moment there is just one square, say the $\tau_{th}$, bearing the symbol $S(r)$ which is "in the machine". We may call this square the "scanned square". The symbol on the scanned square may be called the "scanned symbol". The "scanned symbol" is the only one of which the machine is, so to speak, "directly aware". However, by altering its m-configuration the machine can effectively remember some of the symbols which it has "seen" (scanned) previously. The possible behaviour of the machine at any moment is determined by the m-configuration $q_n$ and the scanned symbol $S(r)$. This pair $q_n, S(r)$ will be called the "configuration": thus the configuration determines the possible behaviour of the machine. In some of the configurations in which the scanned square is blank (i.e. bears no symbol) the machine writes down a new symbol on the scanned square: in other configurations it erases the scanned symbol. The machine may also change the square which is being scanned, but only by shifting it one place to right or left. In addition to any of these operations the m-configuration may be changed. Some of the symbols written down will form the sequence of figures which is the decimal of the real number which is being computed. The others are just rough notes to "assist the memory". It will only be these rough notes which will be liable to erasure.

2.2 Von Neumann Architecture

TODO: Probably add a picture of the Von Neumann Architecture here.

On the other hand, Von Neumann architecture's original description was drafted as part of the EDVAC computer design, and its description is convoluted and hard to follow. Hence it is better to summarize it than to textually quote it.

The Von Neumann Machine describes an organization of a system that implements the Turing Machine. The Von Neumann machine is comprised of three major parts: A Central Processing Unit (CPU), a memory storage, and an input/output interfaces to external devices. I/O interfaces allow the system to communicate with the exterior world. This is, it allows the system to obtain input configurations (programs and data), and provide output results. The memory may contain either numeric values (e.g. positive and negative integers), or encoded instructions (hence the name stored-program computer). Some mechanism in memory (e.g. bit flag) exists to distinguish one from another. On the other hand, the CPU has two parts: 1) an Arithmetic Logic Unit (or Central Arithmetic) that is capable of performing mathematical operations (e.g. $+$, $\times$, $\div$ and $-$), and 2) a Control Unit (or Central Control) which interprets instructions that are fetch from memory and orchestrates the execution of such instruction.

The Von Neumann machine differentiated between the role of the Central Control, and the higher level user defined program. The former discusses the interpretation and orchestration
of the execution of a single instruction within the CPU. The latter is defined as a sequence of instructions in memory, that obey a sequential order, unless the executed instruction ordered the Control Unit to "jump" to a different location in memory and continue executing instructions from there.

2.3 Instruction Set Architectures

From the hardware perspective, the Instruction Set Architecture is a software-hardware interface that defines the vocabulary to be used by the software to program the hardware.

Such description is easy for humans beings to reason about given that it is possible to follow the execution of a code and the different state transitions one by one. Sequential computation has been around for more than 50 years. In spite of wide availability of hardware and programming models for parallelism, sequential computation is still the programming model of choice when it comes to writing (and executing) programs. For this reason, there are many efforts that aim to define what is the best way of introducing parallelism on top of the sequential model.

One additional key model 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 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).

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 [15]. Second, on the software side, a rapid evolution of software infrastructure and algorithms which defined different programming models, programming languages and programming paradigms. All these have built upon each other into multiple layers of complexity, isolating the programmer (i.e. user) from the actual low level execution model. Growth of software technology also has been accompanied by the evolution of compiler technology, libraries and runtime systems that has allowed users to focus on modularized application development, moving them away from understanding the underlying ISA and machine model of the system.

2.4 Program Execution Models

It consists of three parts: a) the activity model which defines the work to be performed; 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. Such PXM needs to explicitly define a model of computation, abstract machine architecture and API definition.
3 Computer Systems, a myriad of models

Figure 1: Conceptual view of computer system infrastructure

To illustrate this evolution figure 1 shows a conceptual view of what current computer systems are in terms of the models and the different interacting parts. At the top we have the users and programmers, and at the bottom we have the executing hardware. As mentioned before, hardware has been accompanied by machine architectural models. Currently most of the computer architectures are highly influenced by the Von Neumann model. Hardware has also 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 user 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 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.
4 Sequential Codelet Model

The Sequential Codelet Model was inspired by the definition of the Codelet Model as explained in [16], in conjunction with the evolution of sequential computing as described in section ???.

Following are the major key observations that inspired this work:

- Sequential single core architectures are currently parallel machines. Aggressive optimizations such as pipelining, superscalar, out of order execution, and register renaming has allowed the Instructions Per Cycle (IPC) count to increase beyond 1. Parallelism is only exploited locally at a window surrounding the program counter, while the compiler optimization techniques (e.g. peephole register allocation and modulo scheduling) allows to increase take further advantage of such parallelism.

- Despite such available parallelism, sequential programming does not require the user to reason about it. While it is possible to code in a way that could potentially improve parallelism, the programmer is not required to think of it neither for correctness, predictability or determinism.

- It is possible for a compiler to optimize and generate quality code for a target ISA due to the known latency of operations, and well defined behavior. Such latency is guaranteed because the access time to registers is neglctable with respect to the computation time.

- Supported operations by hardware architecture and its ISA are application dependant. For example, DSPs architectures provide support for more complex trigonometric operations, while general purpose computers seek to better utilize the die area in basic arithmetic operations, and use the remaining area for local memory.

The size of a Codelet (task) operation is determined by: 1) the required inner state (namely stack and used registers), 2) the instructions memory to describe its computation, and 3) the area of memory that reads or write through load and store operations. Such parameters can be adapted and limited through the code description. Additionally, it is possible to split or merge a Codelet in terms of memory requirements or computational requirements. It is important to guarantee bounded latency operations in Codelets.

4.1 A hierarchical Turing Machine

When Allan Turing described his Turing Machine, he assumed no memory latency nor computation time of each state transition. However, the experience accumulated throughout the years has shown us the importance of low latency memory operations with respect to the computation time. Applications are constantly trying to achieve higher data locality that lead to performance improvements, however, the sequential execution model provided by the original Turing Machine does not consider memory locality.
A hierarchical Turing Machine as depicted in figure 2 is defined as a Turing Machine that relies on a lower level Turing machine to evaluate the operation and transition of a state based on the input symbol read from the infinite memory tape in the level above. Hence, we hierarchically define a state machine where each state’s behavior is itself defined as a state machine in the level below. Each level has an infinite memory tape that is used to evaluate the lower level’s state machine. At some point of the hierarchy of states machines there must exist a constant state that breaks the recursion.

The proposed Turing Machine allows to hierarchically define an operation in terms of lower level operations, which could be contained and well defined, and where the concept of memory locality, and size of an operation are correlated by the level at which the operation is defined. This is, larger memory requirements and more complex operations reside in the higher levels of the hierarchy. Therefore, the orders of magnitude for operation and its memory is also determined by the level that operation is defined.

4.2 The Sequential Codelet Abstract Machine

Based on the hierarchical Turing Machine, we define the Sequential Codelet Abstract Machine. As we mentioned before, we have decided to borrow from the long experience in sequential computation. For this reason, at each level the machine is defined as a Von Neumann machine and described as the well know core architecture with 5 stages pipeline: 1) Fetch, 2) Decode, 3) Execute, 4) Memory, and 5) Write Back.

Figure 3 shows a 3 level implementation of the Sequential Codelet Abstract Machine. The bottom level L0 is just a regular sequential 5 stages pipelined architecture without any modifications. Level L1 has the same pipelined structure, however it uses L0 as its execution unit. Likewise Level L2 uses L1 as its execution unit. The higher we go in the hierarchy, the more complex the operations become, and the larger the memory is. Operations defined at one level are written using operations defined in the level below. Hence, there is a program description for each level (As can be seen later on in the example). Memory is hierarchical as well. There will be a local memory at each level. This memory has a dual purpose. For a particular level it acts as a register file for the operations defined at that level, but for the level below it acts as global memory of the execution unit. Hence, load and store operations in the level below should refer to memory locations in the level above, this way we guarantee that memory latency does not affect the bounded latency property of Codelets. This behavior is depicted in figure 3 as the doted read line connecting the different levels.

We hence define a Codelet as an operation description at one level of the hierarchy (similar to an instruction of an ISA). Codelets are formally defined as a set of instructions comprised by control flow operations, memory instructions, native operations (e.g. arithmetic operations), and Codelets belonging to the level below.
4.3 A convolution example

Before describing an program example for the proposed SCM abstract machine, it is important to understand that the SCM model defines a low level program description language, to be executed by a combination between extended hardware support (e.g. hardware implemented...
Scheduling units), and low level runtimes. For this reason higher level programming language’s constructs (e.g. class, struct, continue loop, and break loop) are left for a compiler to define (or restrict) in terms of this model.

```plaintext
1  int A[1000], window[10], res[1009];
2
3  for (i = 0; i < 1009; i++) {

```
We use the 1D convolution code in listing 1 for this example. While listings 2, 3 and 4 represent the translated code for each of the levels of the aforementioned abstract machine. The following proposed pseudo-codes make several assumptions that would be implementation specific. While these assumptions are not trivial, they do not compromise the validity of the proposed SCM.

### 4.3.1 Language convention

In order to understand the following pseudo-codes the reader must be familiarized with the following conventions. Since there are multiple register files, we use the $R_{\text{level}}^{\text{num}}$ notation to distinguish a particular register number from a particular level. Similar to a register in an ISA, a register has a size, which is provided explicitly at the beginning of the pseudo-code listing. At each level, a memory address references the level above, for this reason we use the $\text{name}_{\text{level}}$ convention to refer to an address in the level above, and we use the $\text{Addr}(\text{Reg})$ operation to translate a register into the address that can be referenced from the level below. Literal constant values are preceded by the # symbol.

### 4.3.2 Three level program description

Each level of the SCM has its own program description. The mapping between the code in 1 and the different levels is still a subject of study, and requires comprehensive research of proper high level programming languages for these abstraction and the compilation techniques. The following pseudo-codes has been translated by hand.

Starting from the highest level, Listing 2 describes the code that L2’s pipeline would be executing as an anonymous codelet_def. We assume that at this level the registers are large enough to hold the vectors $A$, $\text{window}$, and $\text{result}$. This code uses 3 registers of three different sizes. The only action this level has to do is load the memory from the highest level to the local memory, to be presented to the level below through the 1DConv Codelet which behavior is defined in level L1. Although we are assuming large enough registers, it might be more beneficial to split up this code in smaller size registers, and use a loop around it.

```c
int val = 0, ii = i;
for (j = 0; j < 10; j++, ii --)
    val += (ii >= 0 && ii < 1000) ?
        $A[ii] * \text{window}[j]$ : val;
res[i] = val;
```
Listing 2: L2 pseudocode

Following, listing 2 describes the L1 level code. It defines the codelet `conv1D`. This codelet maps the outer $i$ iteration loop in listing 1. This code introduces also control flow instructions (BR and BREQ) that are interpreted by the L1 Scheduling Unit, ideally implemented as part of the computer system hardware. Additionally, Load and Store operations are targeting L2's memory.
This operation has four parameters

AddrA at L2
AddrWindow at L2
AddrRes at L2
sizeOfRes at L2

R10 has the ii iteration variable
R11 has the i iteration variable
R12 has subA[0:100]
R13 has val
R14 has the window variable
R15 refers to the end variable

```c
# Listing 3: L1 pseudocode

codelet_def conv_1D (AddrA_L2, AddrWindow_L2, AddrRes_L2, SizeOfRes):

ST R10, #0 ; ii = 0
LD R14, AddrWindow_L2

Loop_ii:
  BRTG R10, @sizeOfRes, After_ii
  ADD R17, R10, #100 ; R17 = ii + 100
  ; if R17 > sizeOfRes
  BRTG R15, @sizeOfRes, reminder
  LD R15, #100
  BR after_reminder

reminder: ; ; Else
  MODULO R17, @sizeOfRes, #100
  LD R15, R17

after_reminder:
  LD R12, AddrA_L2 [R10]
  ST R11, #0 ; i = 0

Loop_i:
  ; ; i > end
  BRTG R11, R13, After_i
  ; ; L0 level codelet single_conv
  ; ; Result is stored in R13
  ; ; singleConv(&subA, &window, val, i, end)
  singleConv R3, R14, R13, R11, R17
  ADD R17, R11, R10
  ST AddrRes_L2 [R13], R11 ; subRes[ii+i] = val
  ADD R11, R13, #1 ; i++
  BR Loop_i

After_i:
  ADD R10, R10, #100 ; ii++
  BR Loop_ii

After_ii:
  Commit ; ; To L2
```

Finally, listing 4 L0 should be familiar to the reader since it is defined as a legacy ISA for any of the currently available ISAs. Notice that we have removed the superscript from the register names on purpose due to emphasize this fact. Arithmetic operations such as the MULT that are available in current systems could execute this code. It would also be encourage to use
ISA extensions for SIMD to improve this codelet’s behavior.

A sequential execution for this code should go as follows: First, L2 loads vectors $A$ and $\text{window}$ to its local memory, then it issues the $\text{conv1D}$ codelet which is assigned to L1. In addition addresses of registers in L2 are assigned through the $\text{addr()}$ operation and are pass to L1. L1 then starts the execution of its definition by iterating over the elements of the $\text{Res}$ vector. Loading the necessary values from L2 into local memory, then issuing the $\text{singleConv}$ codelet to L0. We have reached then the bottom of our hierarchy, and at this point there is a hardware unit capable of executing and providing results, breaking the hierarchy and returning a value to the levels above. Once L0 has finished it will commit the result through the $\text{writeback synchronization}$ unit of L1. Same happens once L1 and L2 are done, which commit to the level above them.

```plaintext
1;; This operation has five parameters
2;; AddrSubA at L1
3;; AddrWindow at L1
4;; AddrVal at L1
5;; Current value of i
6;; Value of end
7;; $R_0$ has the $j$ iteration variable
8;; $R_1$ has the value of $i - 9$
9;; $R_2$ is used for $A$ values
10;; $R_3$ is used for $\text{window}$ values
11
codelet_def singleConv(AddrSubA_\text{L1}, AddrWindow_\text{L1}, AddrVal_\text{L1}, i, end) :
12ST R_0, #9 ;; $j = 9$
13SUB R_1, @i, #9
14loop_j:
15  BRLT R_0, 0, after
16  ADD R_4, R_1, R_0 ;; $i - 9 + j$
17  BRLT R_4, 0, skip_op
18  BRGE R_4, @end, skip_op
19  LD R_2, AddrA_\text{L1}[R_4] ;; $A[i-9+j]$
20  LD R_3, AddrWindow_\text{L1}, R_0 ;; window[j]
21  MULT R_5, R_2, R_3 ;; $A[i-9+j] \times \text{window}[j]$
22  ST R_5, AddrVal_\text{L1}
23skip_op:
24  SUB R_0, 1, R_0 ;; $j --$
25  BR loop_j
26after:
27  Commit ;; To L1
```

Listing 4: L0 pseudocode

Notice that we do not need to limit the hierarchy to just 3 levels. We could envision deepening the hierarchy even further. In fact, this hierarchy should be mapped to the current hierarchy in computer systems. From the cores at the bottom level, then going up to group of cores, sockets, and even a whole cluster of machines, providing that there is a hardware/software solution that implements the pipeline.
5 Achieving Parallelism

So far we have described a sequential execution model where a program is organized in a hierarchy of instructions. However, the potential of this model comes from all the experience that computer architectures have achieved throughout decades on how to improve Instructions Per Cycle of sequential computation through ILP techniques. Following we describe a set of optimizations that could be applied to improve codelets per cycles at each level. We believe that there is still a lot of potential left to be explored.

5.1 Superscalar

Superscalar machines increase the number of available executing units (ALUs and FPs) available per core. While the issuing of instruction is still sequential, it is possible to avoid structural hazzards in the pipeline and increase parallelism by assigning instructions to different execution units.

For the SCM this is achieved by increasing the execution units at each level. An L0 core can be itself a custom super scalar architecture, but L1 should count with many cores for itself, and L2 should have itself multiple L1 execution units as well. This way we could issue multiple Codelets at multiple levels at the same time, achieving parallelism. Additionally, registers are natural data dependencies that guarantee synchronization among the different cores, without requiring to explicitly assign them. The number of groups that is efficient is still a subject of study.

5.2 Out of order execution

OoO engines are micro dataflow machines. For a given window of code, the system determines the data dependencies between the instructions, allowing execution of instructions in a different order to which they are issued. Each level should have an out of order execution engine associated to itself. This could be achieved by extending the synchronization unit to have an instruction buffer, and a memory reorder buffer at the end of the pipeline.

5.3 Register renaming

Register renaming could allow the program to exploit further parallelism, specially in the case of loops. This technique should be applied at each level to reduce the number of anti dependencies and flow dependencies, increasing utilization of the overall machine.

5.4 Heterogeneous architectures

A single machine should count with cores that have different capabilities. Each focusing on a certain strength. Currently, the benefits of GPU technology has been proven. However, GPUs
still suffer from large costs for complex control flow structures. Additionally, CPU cores and GPU cores are usually in different chips separated by large memory latencies, exacerbated by distinct memory address spaces that requiring explicit memory copying mechanisms. Instead, we propose to have heterogeneous multicores in the superscalar architecture at the lower level. Hence each architecture could provide its strengths to the execution of the program, while the programmer does not have to be burden with programming for different architectures. An example of this are floating point units, which at some point used to be also acceleration devices, but as technology advanced, they were integrated into the core architectures and available to the programmer as ISA instructions. This applies also for FPGAs, Nvidia Tensor Cores, Google TPUs, and other current novel architectures for different applications.

5.5 compiler techniques

Another vast advantage of the proposed approach is to be able to use all the experience obtained in compiler techniques, specially in the back end of compilers. It is then possible to optimize execution of code given a particular machine description that is somehow provided to the compiler at each level. An important restriction is that Codelets should have a latency that the compiler is able to reason about. For this reason, execution of Codelets cannot depend on latency operations that are larger than the execution of code itself and that the compiler cannot know. The hierarchical memory allows this by always using the memory that is the closest to the execution unit, while maintaining the relationship between computation and memory latency similar to register in current architectures.

However, this also implies that if the computation is dependent on a parameter provided by the user, a mapping mechanism, which possibly requires JIT compilation techniques would be necessary.

6 Conclusions

In this position paper we present the Sequential Codelet Model. A Model of computation that heavily borrows from sequential execution models of computation and the experience achieved in single core architectures. However, this model allows parallelism without burdening the programmer to think about it, in the same way that Instruction Level Parallelism conceptually works. We provide the definition of a hierarchical Turing Machine, followed by a possible implementation defined as an abstract machine. We then provides an example of a code written for such an abstract machine, and a description of how parallelism could be achieved.

References

http://doi.acm.org/10.1145/3282307


