The SuperCodelet Architecture

Jose M Monsalve Diaz
Argonne National Lab
Chicago, IL, USA

Kevin Harms
Argonne National Lab
Chicago, IL, USA

Rafael A. Herrera Guaitero
University of Delaware
Newark, DE, USA

Diego A. Roa Perdomo
University of Delaware
Newark, DE, USA

Kalyan Kumaran
Argonne National Lab
Chicago, IL, USA

Guang R. Gao

Abstract

This paper proposes a hierarchical organization of a distributed, parallel and heterogeneous computer system based on the Sequential Codelet Model. It defines a Hybrid Dataflow/von Neumann architecture which enables parallel execution of sequentially defined Codelets at a multi-hierarchical organization of the system. This architecture takes advantage of instruction level parallelism techniques based on dataflow analysis, allowing implicit parallel execution of code. We present the SuperCodelet Architecture. We describe the program execution model and its corresponding programming model, and provide an evaluation of its program execution model through a prototype emulator based on commodity computer systems.

CCS Concepts

• **Computer systems organization** → **Multicore architectures**; **Heterogeneous (hybrid) systems**.

Keywords

Heterogeneity, parallel architectures, Sequential Codelet Model

ACM Reference Format:
Jose M Monsalve Diaz, Kevin Harms, Rafael A. Herrera Guaitero, Diego A. Roa Perdomo, Kalyan Kumaran, and Guang R. Gao. 2022. The SuperCodelet Architecture. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Computer systems are highly complex machines containing large number of parts interacting with each other. An ideal system architecture should feature high performance, portability, and easily programmable interfaces. When looking back in history [6], it is sequential computing architectures that have been able to exploit these three aspects the most. There are three major events that lead to the sequential computing success. First, the Turing Machine introduced by Allan Turing in 1936 [19] which provided the necessary mathematical model. Second, the definition of the Von Neumann architecture in 1945 [13] that defined the components needed to implement the Turing machine and their interaction. And third, starting in the 1960's [1] by Frederick Brooks and his IBM/360 design team, the introduction of a common Instruction Set Architectures (ISA) as a single interface between hardware and software.

The standardization of the ISA worked as a long lasting contract between software and hardware, allowing them to evolve independently and grow apart. Hardware architects focused on improving instructions per cycle, resulting in complex pipelines and Instruction Level Parallelism (ILP) optimizations. On the other hand, software was relieved of the constant need to adapt programs to new system architectures. As long as new software targeted the same ISA and execution model provided by the hardware (i.e. Turing Machine and Von Neumann model), it could freely evolve and increase in complexity. Unlike sequential computing, trending parallel processing lacks an acceptable common abstraction between software and hardware. Consequently, software (and developers) have been burdened with scheduling tasks such as workers creation, workload distribution, memory organization, and synchronization. As a result, parallel computing is difficult and it heavily relies on software implemented runtimes.

In sequential computing, Out of order (OoO) execution is an ILP optimization that allows for parallel execution of sequential instructions. OoO is a dataflow inspired execution mechanism that, during runtime, enforces dependencies across a window of instructions around the program counter. Independent instructions are allowed to change their original order, as long as communication with the outer world is committed in a coherent order that resembles the original program. ILP allows implicit parallelism, therefore, a programmer can be agnostic of the real execution order of the instructions. However, an avid programmer (or compiler) can still change the source code to exploit these mechanisms and improve performance.

This paper builds upon the idea that, in order to bring back performance, portability and programmability, it is necessary to define a common program execution model (PXM) [15] for parallel, distributed and heterogeneous machines. The PXM model takes advantage of the original sequential abstraction to exploit parallelism through ILP techniques, especially Out of Order execution. This work relies on a hierarchically organized machine model and its memory infrastructure that closely maps current computer systems. By using the models presented by Monsalve Diaz et al. in [10] [11], we define the SuperCodelet architecture that uses sequential imperative programming semantics similar to assembly code. Unlike current assembly, the supported operations are tasks, called Codelets, that are also user defined. A Codelet is a set of instructions, a program is defined as a Codelet Graph, where codelets are nodes and data and control dependencies are represented as edges. A Codelet is dormant when the dependences have not been met, it becomes enabled once these dependences are fulfilled and it is fired once the

Conference'17, July 2017, Washington, DC, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

hardware resources required for his execution are available, becoming active. In comparison to previous Dataflow/von Neumann hybrid abstractions, the proposed organization is structured hierarchically, and it can be extended beyond a single level of computational elements. Furthermore, it is based on an abstract machine model that is different to regular task scheduling solutions [4] [12] [21]. In this paper we will focus our attention on a two level abstraction composed of heterogeneous architectures. The objectives of this paper are as follow:

- define the SuperCodelet architecture based on the Sequential Codelet Program Execution Model [10],
- create an emulation runtime that maps the Sequential Codelet Model to current commodity hardware to emulate the behavior of the SuperCodelet architecture, and
- present results of this emulation runtime that validates parallel and heterogeneous execution of sequential Codelet programs.

2 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, it presents a machine organization, a description on how programs execute on this machine, and a description on how programs should be written.

Perhaps the most valuable aspects of the Sequential Codelet Model are: 1) it presents a hierarchical organization of Turing Complete machines that enables computation to occur at any level of the abstraction. 2) At each level, programs are defined sequentially in an imperative style programming model. The SCM machine uses 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 it relies on sequential consistency across each level. 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. This work focuses on multi-core heterogeneous systems.

2.1 Machine Model: Hierarchical Von Neumann

Based on the Hierarchical Turing Machine presented in [10], we define the Sequential Codelet Model abstract machine, called the Hierarchical Von Neumann (HVN) Model. Figure 1 shows a simplified diagram of the HVN abstract machine. The main difference between the original Von Neumann Model and the HVN model relies in the Arithmetic Logic Unit (ALU) that forms the hierarchy. The ALU at each level is seen as a complete Von Neumann machine that represents the level below. Therefore, operations (i.e. Codelets) supported by each level's ALU is programmable in terms of instructions in the lower level. At a given level of the hierarchy N, programs are sequentially described and composed of three type of instructions: Codelets, memory, and control flow.

A Codelet is similar to a task. A Codelet is a collection of instructions that are scheduled atomically and non-preemptive, defined by

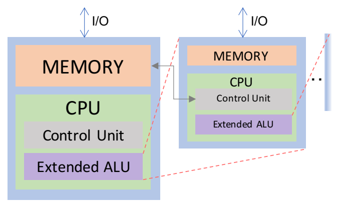

Figure 1: Hierarchical Von Neumann Architecture.

its event and data dependencies. In the Sequential Codelet Model, Codelets are the basic unit of operation of a given level. The functionality of a Codelet in level N is defined in the level below $N - 1$ as a sequential Codelet program. Additionally, each Codelet has a well defined API and is side-effect free (similar to ISA instructions). Control flow instructions determines Codelets scheduling. Regular arithmetic operations (e.g. ADD, SUB, and MULT) are intrinsic Codelets for all levels that do not require a definition in the level below. The level of complexity of each levels' ISA is up to the design of the machine. This flexibility allows for in-memory computation to be described as a program in the upper levels of the hierarchy.

As a result of the structure of the HVN machine, memory is hierarchically organized. When a Codelet program is executing, load and store operations of level $N - 1$ will fall into memory of level N. Thus, limiting the memory latency range in between two consecutive levels. Simple memory operations copy consecutive data from one level to the level below. However, more complex memory operations can be defined to perform intelligent memory fetching (e.g. scatter-gather). On the other hand, memory instructions inside of the Codelet definition only reference the Codelet's operands. Therefore, mixed memory consistency models can be used at different levels, while maintaining a relaxed consistency system wide.

A Hierarchical Von Neuman machine can use a combination of other architectural models. This is analogous to floating point arithmetic units in current architectures, or Nvidia's Tensor cores [14]. In our hierarchical organization heterogeneity computation is supported by including architectures with other execution models (e.g. GPU cores), or application-specific silicon (e.g. Neuromorphic chips [2][8][22]) to the ALU of any given level of the HVN model. A Codelet mapping to these architectures is written in the corresponding programming model.

2.2 The Sequential Codelet Model

Figure 2 shows a 3 level HVN extended abstract machine. This system uses a five stages pipeline at each level of the HVN machine: Fetch, Decode, Execute, Memory and Write back. At the bottom, level L0 corresponds to any architecture that is commonly found nowadays (e.g. a core implementing any of the commodity architectures: RISC-V, ARM, x86, or POWER PC). Going up in Figure 2, the execution stage (Extended ALU) of the L1 pipeline corresponds to L0. Likewise, L1 is the execution stage for L2. Therefore, L1 Codelets are scheduled to L0 and L2 Codelets to L1.

Codelet operands are stored in a register file. The register file in L1 is also seen as memory of L0, and the register file of L2 is also seen as memory of L1. Therefore, memory instructions inside a Codelet will fetch data from the register file of the level above. The higher the level of the hierarchy, the larger the memory capacity is. Therefore, the aggregated size of the register file as well as the

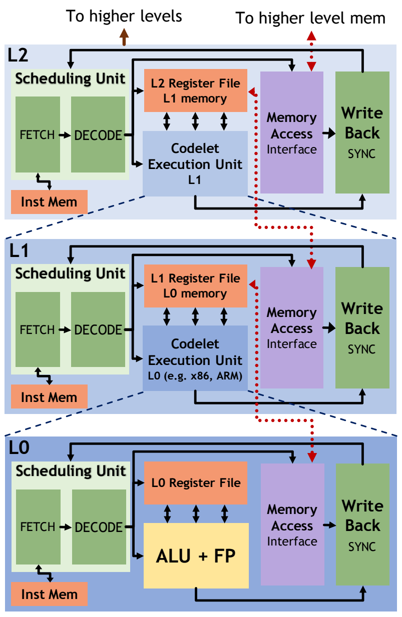

Figure 2: A 3 level abstract machine of the Sequential Codelet Model that implements the Hierarchical Von Neuman Model.

maximum size of each register is expected to be higher as we go up the hierarchy. We opted to use registers since it provides a fixed limit for memory complexity of Codelets, allowing for compiler optimizations and analysis. Furthermore, registers allows hardware dependency handling mechanisms such as those present in ILP.

3 SuperCodelet Architecture

This architecture uses all the elements presented in section 2, and it allows a construction of a parallel and heterogeneous system. We use Tomasulo's original architecture [16] as base, and we extend it for the upper levels of the hierarchical abstract machine of Figure 2. The system organization presented in here is just one possible organization of a system that implements the Sequential Codelet Model. Our intention in this paper is to show how to achieve parallelism on heterogeneous systems using the SuperCodelet architecture, demonstrating that the proposed architecture does not go against the progress made until now in heterogeneous architectures, but instead builds upon it.

As mentioned before, parallel execution is achieved through the use of techniques that derive from instruction level parallelism. At a given level, the system must maintain the order of true-dependent instructions (i.e. Read after write), while anti dependencies and output dependencies can be eliminated through Tomasulo's algorithm and register renaming techniques. Independent instructions are allowed to execute in parallel. Heterogeneity is achieved through using different architectures in Level L0 as execution units of the SCM machine.

Figure 3 shows the diagram of the SuperCodelet architecture for up to 3 levels. Level L2 (light gray background) is partially shown. Level L0 (dark gray background at the bottom) is seem as a black box that implements architectures that should already be familiar to the reader such as CPU cores, GPUs streaming multiprocessors or FPGAs. Finally multiple instances of Level L1 (mid-gray background) are displayed, but only one is expanded and labeled.

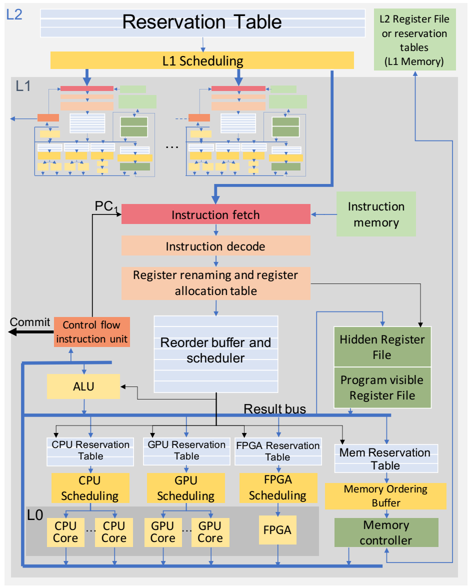

Figure 3: Diagram of the SuperCodelet architecture

Centering our attention in L1, it is easy to notice the similarities of this diagram to diagrams used in variations of Tomasulo's algorithm in current single core architectures. Starting at the bottom, the usual FPU and ALU units have been replaced with CPU, GPU and FPGA cores. Each type containing its own reservation table and scheduler. A traditional ALU is added to the left to perform regular arithmetic operations at level L1. Continuing, on the bottom right there is the memory interface with its corresponding memory ordering buffer to guarantee in order commit of instructions. This memory is connected to L2's register file and reservation tables (top right), which acts as L1's memory. In the middle we have instruction fetch (connected to the instruction memory), instruction decode, register renaming and allocation logic, and the reordering buffer. To the right of this section there is the hidden and program visible register files. This aforementioned structure repeats twice on the top part of the L1 box, thus representing the potential for multiple of these units which act as L2 execution units, enabling parallelism across L1.

3.1 Execution order

Assume we are describing an L2 Codelet that is currently mapped to an L1 execution unit, as seen in Figure 3. Execution stages are:

Fetch and Decode The instruction fetch unit starts reading instructions of the current L2 Codelet from the Instruction Memory and gives it to the decode stage. Following, register renaming and allocation removes possible anti and output dependencies. Then, depending on the instruction type, it is pushed into a reordering buffer of the corresponding L0 execution unit, memory controller or the L1 ALU.

Codelets are defined to be executed in the level below. A given Codelet may be represented in different variants of the L0 architectures. For example, a Codelet that performs a Matrix Multiplication may be implemented in GPU and CPU. Execution goals or system state could determine what variant to use. For example, the GPU may by busy, or there may be different power goals for the overall execution of the program that disable some units [9]. Codelets are not bounded to a specific core, but they have an identifier in the reservation table that will be used to fetch its instructions.

Execute Reservation tables contain a CodeletID and its operands. These Codelets may be waiting for values to be provided by other execution units. Once all the operands of a Codelet are available, the corresponding Scheduling unit will assign it to one of the available execution units. Once assigned, Codelets must execute until completion in a non-preemptive fashion. When a Codelet finishes, the execution unit will signal the scheduler. The Scheduler will now signal all the reservation tables for instructions waiting for this Codelet to complete, and the entry in the reservation table is freed for a new instruction to come in.

Memory Units Memory operations of a level will result in access to the level above. Memory operations require its own infrastructure for re-ordering according to the chosen memory consistency model. Access will flow through the bus of level L2 into the corresponding location: a register in the register file, or a reservation station for the consumer Codelet. There are two register files. One that is accessible by the user, corresponding to all the different registers available to the user for the description of programs. A second register file is used for register renaming opportunities.

Commit L2 Codelet has finished execution, a commit operation signals the following instructions in L2 that are waiting. When the final instruction finishes, a Commit operation will guarantee that the pipeline is empty, and signal L2 for completion.

4 Programming model

A program written for the SuperCodelet model maps to the hierarchical organization of the machine. For each level, the user must declare the collection of available Codelets. A Codelet is define by using Codelets and instructions of the level below. For example, if a Codelet is to be used in L2, that Codelet needs to be defined in a sequential Codelet program in level L1. At all levels above L0, the user has access to a set of operations that are common. We refer to these as the SuperCodelet ISA. Consequently, Codelet programs use the Codelets of the level below, plus supported SuperCodelet ISA operations. We divide the instructions in four groups:

- **Arithmetic instructions:** `ADD`, `MULT`, `SUB`, and Shift right and left (`SHFR/SHFL`). Their syntax have the format `OP` $R_D,\ R_S,\ R_T$
- **Control:** `JMPLBL` (to label) and `JMPPC` (to offset) for unconditional branching. `BREQ`, `BGT/BLT`, and `BGET/BLET` for conditional branching.
- **Memory:** `LDIMM` for immediate values. `LDADDR` and `STADDR` for load and store operations. `LDOFF` and `STOFF` for load and store with offsets.
- **Codelet Control:** `COD <name>` and `COMMIT`

These instructions form the bare minimum for our design, but they do not limit possible extensions in the future. The Codelet control operations coordinate the communication across levels for initiating and finishing Codelet Execution. 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 results are ready to be used.

Codelets are user defined, therefore it is necessary to provide flexibility in terms of number, type (Register or Immediate value), and read/write direction of operands. Read or write direction allows the out of order engine to discover dependencies during execution.

5 Evaluation

In order to evaluate the feasibility of this architecture and to provide an early prototype, we have developed an emulator of the SuperCodelet architecture: SCMUlate [7]. This emulator resembles the behavior of two levels of the Super Codelet Model, and it runs in current multicore systems with support for integrated GPUs through OpenMP offloading code generation. We show a Matrix Multiplication example.

All tests were performed on an Intel Core i7-8700k processor. There are 6 CPU cores, each with HyperThreading technology (i.e. SMT with 2 threads), for a total of 12 hardware threads. In addition, it contains an Intel UHD Graphics 630 Gen 9.5 architecture integrated GPU. The processor is hosted in a Dell Model Precision 3630 desktop tower. Additionally, it is equipped with 32 GB of DRAM distributed in two 16 GB DDR4 DIMMs, each running at 2666 MHz and a last level of cache (LLC) of 12MB.

The Assembler (Interpreter) It allows to interpret the set of SuperCodelet ISA operations as mentioned in section 4. Assembly code is translated into executable instructions at runtime. The input of the assembler is a text file containing the code of level L1. The output is a list of objects representing the Instruction.

The Runtime It maps the SuperCodelet Abstract Machine and its execution model to the available resources. A software thread is created per hardware thread in the system and each thread is assigned a role. There are three roles in the Abstract Machine: SU or Scheduling Unit, CU or Computation Unit, and MU or Memory Units. One single thread is the Scheduling Unit. This thread does not participate in the computation, but it is in charge of the arithmetic and control operations of the SuperCodelet ISA. The rest of the threads are all Computational Units and Memory Units executed concurrently one role at a time depending on the instruction. The Scheduler Unit performs fetch and decode operations. If the instruction is an arithmetic or control instruction, then it is executed immediately. Otherwise, it needs to be scheduled to a CU or an MU. There are three modes of operation: *sequential*, *Superscalar*, and *Out of Order*.

L0 Codelet definition These are declared and defined in C++ through the use of classes and macros. A Codelet is automatically registered to a lookup table. When the assembler sees a `COD` instruction, it uses the lookup table to find the implementation, creates an executable Codelet containing the arguments, and assigns it to one of the computational Unit.

Register File To emulate the behavior of the register file, SCMUlate uses the LLC. A segment of memory in DRAM is allocated as big as the LLC. A register name keeps a reference to this location in memory, allowing to maintain locality between producer and consumer Codelet. The expectation is that any memory access that occurs within a Codelet scheduled in a CU (GPU or CPU) will have almost zero cache misses in the LLC. Following, the simulated L1 register file is partitioned into registers of different sizes that are multiples of the cache line. Memory instructions, such as LD and ST, perform copies from a memory location of L2 (i.e. DRAM) to the segment of memory that maps the L1 register file.

5.1 Matrix multiplication

Dense Matrix multiplication is one of the most widely used kernels in scientific computation and data sciences. 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 for this example is a tiled Matrix Multiplication, using an L1 Codelet per tile with L1 registers as operands. This Codelet is called `MatMult_2048L`. The matrix multiplication performed inside this Codelet is fixed in size. The operation to be performed is $C = C\ A * B$.

The `MatMult_2048L` uses registers of size 2048L (i.e. 2048x64 = 131072 bytes). These registers can fit a square tile of dimension $\sqrt{\frac{2048*64}{8}} = 128$. When copying data between memory and registers 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. These Codelets are named `LoadSqTile_2048L` and `StoreSqTile_2048L` respectively. Listing 1 shows the matrix multiplication L1 SCM code for a single tile. The complete code can be seen in the GitHub repository [7].

```
LDIMM R64B_1 , 0; // Loading base address A
LDIMM R64B_2 , 131072 ; // Loading base address B
LDIMM R64B_3 , 262144 ; // Loading base address C
// Single tile code
COD LoadSqTile_2048L R2048L_1 , R64B_1 , 128; // Load A tile
COD LoadSqTile_2048L R2048L_2 , R64B_2 , 128; // Load B tile
COD LoadSqTile_2048L R2048L_3 , R64B_3 , 128; // Load C tile
COD MatMult_2048L R2048L_3 , R2048L_1 , R2048L_2 ;
COD StoreSqTile_2048L R2048L_3 , R64B_3 , 128; // Store C tile

COMMIT ;
```

Listing 1: Matrix Multiplication 1 tile: C.

```
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 2: Optimized version of matMult_2048L.

Three different implementations of the MatMult 2048L Codelet were explored: No optimized, user optimized, and Matrix Multiplication with Intel's MKL library. The No-optimized is a naïve Matrix Multiplication implementation. The user optimized implementation swapped two lines of the naïve version, allowing a modern compiler to easily vectorize this code using SIMD extensions. Finally, Listing 2 uses Intel's Math Kernels Library (MKL) to highly optimize the execution of Matrix Multiplication. A GPU implementation is achieved by using an OpenMP `target` region inside of the Codelet's implementation. OpenMP is only used to generate code for the GPU and schedule it, but not to speed up computation outside of the Codelet. Intel's OneAPI compiler is used to generate code for the Intel Gen9 GPU.

5.2 Results

Figure 4 shows the execution time of a single Codelet in all three implementations. It also overlaps the overall execution time for M = N = K = 40 tiles running on all 12 threads. Out of order execution is enabled in SCMulate. This figure shows it is possible to change the implementation of a Codelet that keeps its API in the upper level.

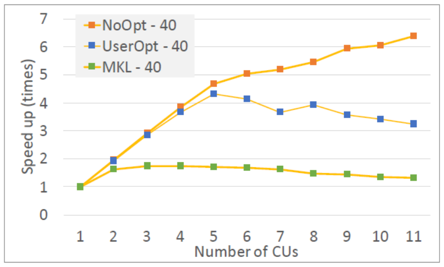

Figure 4: M=N=K=40 tile dimension. Scalability comparison for different implementations

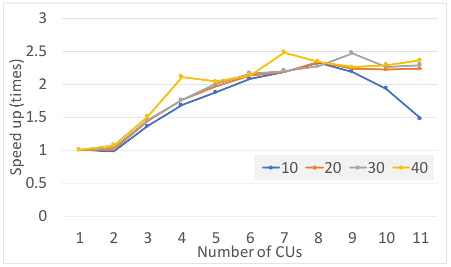

Figure 5: M=N=K=40 tile dimension. Scalability for GPU computation with the number of GPU computational units

Figure 4 shows a scalability comparison of all three Codelet versions. NoOpt and UserOpt show an almost perfect scaling until 5 CUs. After 6 CUs scalability is affected by resource sharing in HyperThreading. Given that all the Codelets are computational units and memory units, memory bandwidth is saturated after 6 cores are used. On the other hand, Scalability in the MKL version is affected by a large scheduling time in the emulator, in comparison to the execution time of the Codelet. These results influence the formulation of section 5.3. Figure 5 shows the execution time of the Matrix Multiplication microbenchmark when running on the Intel Gen 9.5 Integrated GPU. The LLC is shared between the CPU and the GPU, allowing for access to share values between CPU and GPU. Similar to the case of MKL, scalability is poor due to the high performance of the GPU Codelets. Larger Codelet sizes are needed to fully take advantage of the SCM machine.

5.3 Defining the appropriate size of Codelets

The size of a Codelet is determined in terms of execution time. Figure 6 shows how the size of Codelets have a direct impact on the execution time of the application. Figure 7 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 a constant time T_s to schedule an instruction (Codelet) into a CU and that 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 grows, proportional to the number of CUs. Assuming no synchronization cost, and no extra overhead in the system, the time between one allocation and the next one, for the same CU is equal to $T_s X N_{CU}$ 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

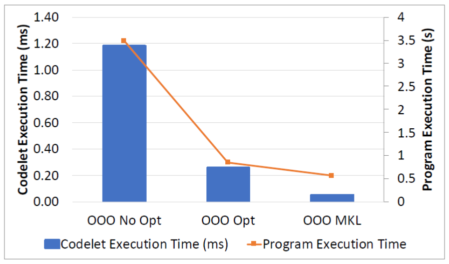

Figure 6: 5 CUs and M=N=K=20. Bars: Codelet execution time (left axis). Line: Program Execution Time (right axis).

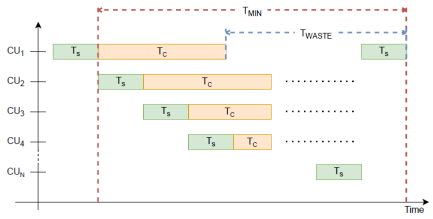

Figure 7: Sequential bottleneck. T_s Scheduling time for Codelet. T_c Compute time. T_{MIN} minimum compute time to avoid T_{waste} sub-utilization

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. The size of the Codelet has a direct impact on scalability, as seen in section 5. One must increase the Codelet size (i.e. T_C), or reduce scheduling time (i.e. T_s) to allow to take advantage of the Sequential Codelet Model.

6 Related Work

This work extends the original Codelet Model proposed by Suetterlein et al. in [18], we have borrowed the definition of Codelet, change the machine model and program execution model. The Sequential Codelet Model is a hybrid dataflow/Von Neumann architecture. Historically, several hybrid approaches have been proposed and can be reviewed in surveys such as [20] [17]. Perhaps the ones that share the most properties with this work are [4][12] [5] [3]. They propose a similar description of sequential execution of tasks that are executed in parallel by means of out of order execution engines. Furthermore, some prior work proposes the use of hardware mechanisms for handling data dependencies across tasks. Although we use a similar sequential abstraction and out of order execution schemes, the Sequential Codelet Model proposes a hierarchical organization of both scheduling and memory. Such organization result in an execution model that may be used beyond multiple cores, and possibly mapped to multiple nodes and groups of nodes.

7 Conclusions

This paper introduces the SuperScalar architecture. An architecture based on the Sequential Codelet Model. The hierarchical structure of the SuperCodelet architecture aims to utilize hierarchical sequential semantics to be used in parallel, distributed and heterogeneous systems, taking advantage of out of order execution techniques that are inspired by dataflow models of computation. We present a theoretical background, system organization, and programming model for the SuperCodelet Model. We also present an evaluation framework that shows the feasibility of our system in current commodity parallel and heterogeneous hardware.

References

[1] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks. 1964. Architecture of the IBM System/360. *IBM J. Res. Dev.* 8, 2 (April 1964), 87–101. https://doi.org/10.1147/rd.82.0087

[2] Andrew S. Cassidy, Jun Sawada, Paul Merolla, John V. Arthur, Rodrigo Alvarez-Icaza, Filipp Akopyan, Bryan L. Jackson, and Dharmendra S. Modha. 2016. TrueNorth: A High-Performance, Low-Power Neurosynaptic Processor for Multi-Sensory Perception, Action, and Cognition.

[3] E. Castillo, L. Alvarez, M. Moreto, M. Casas, E. Vallejo, J. L. Bosque, R. Beivide, and M. Valero. 2018. Architectural Support for Task Dependence Management with Flexible Software Scheduling. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 283–295.

[4] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M. Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. 2010. Task Superscalar: An Out-of-Order Task Pipeline. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, USA, 89–100. https://doi.org/10.1109/MICRO.2010.13

[5] G. Gupta and G. S. Sohi. 2011. Dataflow execution of sequential imperative programs on multicore architectures. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 59–70.

[6] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (Jan. 2019), 48–60. https://doi.org/10.1145/3282307

[7] Jose M Monsalve Diaz. [n.d.]. SCMUlate Emulator Source Code. https://github.com/josemonsalve2/SCM.

[8] Norman P. Jouppi et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[9] Aaron Landwehr. 2016. An experimental exploration of self-aware systems for exascale architectures.

[10] J. Monsalve, K. Harms, K. Kalyan, and G. Gao. 2019. 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)*. 1–8.

[11] J. Monsalve Diaz. 2021. *Sequential codelet model: a supercodelet program execution model and architecture.* Ph.D. Dissertation. University of Delaware.

[12] Lucas Morais, Vitor Silva, Alfredo Goldman, Carlos Alvarez, Jaume Bosch, Michael Frank, and Guido Araujo. 2019. Adding Tightly-Integrated Task Scheduling Acceleration to a RISC-V Multi-Core Processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 861–872. https://doi.org/10.1145/3352460.3358271

[13] John von Neumann. 1945. *First Draft of a Report on the EDVAC.* Technical Report.

[14] NVIDIA. 2017. 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.

[15] Peng Qu, Jin Yan, You-Hui Zhang, and Guang R. Gao. 2017. Parallel Turing Machine, a Proposal. *Journal of Computer Science and Technology* 32, 2 (01 Mar 2017), 269–285. https://doi.org/10.1007/s11390-017-1721-3

[16] R. M. Robert Tomasulo. 1967. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development* 11, 1 (1967), 25–33.

[17] Borut Robic, Jurij Silc, and Theo Ungerer. 2004. Beyond Dataflow.

[18] Joshua Suettlerlein, Stéphane Zuckerman, and Guang R. Gao. 2013. An Implementation of the Codelet Model. In *Euro-Par 2013 Parallel Processing*, Felix Wolf, Bernd Mohr, and Dieter an Mey (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 633–644.

[19] Alan M. Turing. 1936. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* 2, 42 (1936), 230–265. http://www.cs.helsinki.fi/u/gionis/cc05/OnComputableNumbers.pdf

[20] Fahimeh Yazdanpanah, Carlos Alvarez-Martinez, Daniel Jimenez-Gonzalez, and Yoav Etsion. 2014. Hybrid dataflow/von-Neumann architectures. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1489–1509. https://doi.org/10.1109/TPDS.2013.125

[21] Fahimeh Yazdanpanah, Daniel Jimenez-Gonzalez, Carlos Alvarez-Martinez, Yoav Etsion, and Rosa M. Badia. 2013. Analysis of the task superscalar architecture hardware design. *Procedia Computer Science* 18 (2013), 339–348. https://doi.org/10.1016/j.procs.2013.05.197

[22] Wenqiang Zhang, Bin Gao, Jianshi Tang, Peng Yao, Shimeng Yu, Meng-Fan Chang, Hoi-Jun Yoo, He Qian, and Huaqiang Wu. 2020. Neuro-inspired computing chips. *Nature Electronics* 3 (07 2020), 371–382. https://doi.org/10.1038/s41928-020-0435-7