Thèse n° 9941

Compilation and Design Space Exploration of Dataflow Programs for Heterogeneous CPU-GPU Platforms

Présentée le 9 juin 2023

Faculté des sciences et techniques de l'ingénieur
Groupe SCI STI MM
Programme doctoral en génie électrique

pour l'obtention du grade de Docteur ès Sciences

par

Aurélien François Gilbert BLOCH

Acceptée sur proposition du jury

Dr J.-M. Sallese, président du jury
Dr M. Mattavelli, directeur de thèse
Prof. J.-F. Nezan, rapporteur
Dr M. Raulet, rapporteur
Prof. A. Burg, rapporteur

École
polytechnique
fédérale
de Lausanne

2023

Acknowledgements

I would like to start by expressing my gratitude to my thesis advisor, Dr. Marco Mattavelli, for the opportunity to pursue my own research topic and for his support throughout this thesis.

Additionally, I extend my thanks to the members of my thesis committee, Dr. Jean-Michel Sallese, Prof. Andreas Burg, Prof. Jean-François Nezan, and Dr. Mickaël Raulet for their feedback, constructive criticism, and expert evaluation of my work.

I would also like to extend my appreciation to both Dr. Endri Bezati and Dr. Simone Casale Brunet for their valuable technical guidance at various stages of my thesis.

I would like to thank all past and current members of the SCI-STI-MM group, especially my colleagues Ana Hernandez-Lopez, Ünsal Öztürk, and Mustafa Beşirli for these long debate and discussion fueled with coffee that made this long journey more interesting. A warm thank you goes to our lab secretary, Daniela Vallat, for bringing the lab together.

I am very lucky to have great circles of friends both in France and in Switzerland and I would like to thank them all for their affection along this difficult endeavor. A truly special recognition goes to my closest friends Léa Prost and Chiara Ercolani for their tremendous and unwavering moral support without which this thesis would not have been completed.

I would like to express loving thanks to my partner Lucy Fitzgerald for her support during the final stretch of this thesis, which allowed me to gain perspective.

Last but not least, I have to sincerely express my love to all members of "La famille proche" but of course more specifically to my parents Florence and Jean-Sébastien Bloch without whom I wouldn't be there today both literally and figuratively for their love, and infinite support all along my studies.

Lausanne, April 28, 2023 Aurelien Bloch

Abstract

Today's continued increase in demand for processing power, despite the slowdown of Moore's law, has led to an increase in processor count, which has resulted in energy consumption and distribution problems. To address this, there is a growing trend toward creating more complex heterogeneous systems where multicore, many-core, GPU, FPGA, and DSPs are combined in a single system. This poses challenges in terms of how to take advantage of such systems and how to efficiently program, evaluate, and profile applications where sub-components run on different hardware. Dataflow programming languages like RVC-CAL have proven to be an appropriate methodology for achieving such a complex goal due to their intrinsic portability and the ability to easily decompose a network of actors on different processing units, matching the heterogeneous hardware. Previous research has shown the efficacy of this methodology for systems combining multicore, many-core, CPU, FPGAs, and others. It has also been shown that the performance of programs executed on heterogeneous parallel platforms largely depends on the design choices regarding how to partition the computation on the various processing units. In other words, it depends on the parameters that define the partitioning, mapping, scheduling, and allocation of data exchanges among the various processing elements of the platform executing the program. The advantage of programs written in languages using the dataflow model of computation is that executing the program with different configurations and parameter settings does not require rewriting the application software for each configuration but only requires launching a new generation of the execution code corresponding to the parameters, using automatic generation tools. Another competitive advantage of dataflow software methodologies is that they are well-suited to support designs on heterogeneous parallel systems as they are inherently free of memory access contention issues and naturally expose the available intrinsic parallelism. However, it is still an open research question whether dataflow programming languages such as RVC-CAL can fit with massively parallel SIMD architecture such as GPUs. Recent GPU architectures make available numbers of parallel processing units that exceed by orders of magnitude the ones offered by CPU architectures. While programs written using dataflow programming languages are well-suited for programming parallel heterogeneous systems, they may not offer sufficient parallel degrees to efficiently exploit the resources available on today's GPUs. Furthermore, the dynamic nature of the RVC-CAL model may conflict with the very rigid SIMD pipeline. The objective of this thesis is to develop a full suite of tools using the dataflow programming language RVC-CAL to provide an automated design flow for programming, analyzing, and optimizing application programs running on CPU/GPU heterogeneous systems. The main contribu-

tions of this thesis are the development of a high-level compiler infrastructure that targets CPU/GPU heterogeneous processing platforms and supports the full specification of the RVC-CAL dataflow programming language, facilities for generating instrumented applications for profiling purposes, and a set of design space exploration pipelines to automatically optimize the resulting application by suggesting performant partition and mapping configurations.

Key words: dynamic dataflow, source-to-source compiler, RVC-CAL, heterogeneous CPU-GPU systems, SIMD, parallel computing, profiling, performance estimation, design space exploration.

Résumé

L'augmentation continue de la demande en puissance de calcul, et ce malgré le ralentissement de la loi de Moore, a conduit à une augmentation du nombre de processeurs, ce qui a entraîné des problèmes de consommation et de distribution d'énergie. Pour remédier à cela, il y a une tendance croissante à la création de systèmes hétérogènes plus complexes où les processeurs multicore, many-core, GPU, FPGA et DSP sont combinés dans un seul système. Cela pose des défis en termes d'utilisation de ces systèmes, de programmation efficace, d'évaluation et de profilage des applications dans lesquelles les sous-composants s'exécutent sur différents matériels. Les langages de programmation de flux de données tels que RVC-CAL se sont avérés être une méthodologie appropriée pour atteindre un tel objectif en raison de leur portabilité intrinsèque et de leur capacité à décomposer facilement un réseau d'acteurs sur différentes unités de traitement, correspondant aux systemes hétérogènes. Des recherches antérieures ont montré l'efficacité de cette méthodologie pour des systèmes combinant des processeurs multicore, many-core, FPGA et autres. Il a également été démontré que les performances des programmes exécutés sur des plates-formes parallèles hétérogènes dépendent largement des choix de conception concernant la manière de partitionner le traitement sur les différentes unités de calculs. En d'autres termes, cela dépend des paramètres qui définissent le partitionnement, l'assignement, l'ordonnancement et l'allocation des échanges de données entre les différents éléments de traitement de la plate-forme exécutant le programme. L'avantage des programmes écrits dans des langages utilisant le modèle de calcul en flux de données est que l'exécution du programme avec différentes configurations et paramètres ne nécessite pas de réécrire le logiciel pour chaque configuration, mais nécessite simplement de lancer une nouvelle génération du code d'exécution correspondant aux paramètres, en utilisant des outils de génération automatique. Un autre avantage concurrentiel des méthodologies de logiciels de flux de données est qu'elles conviennent bien à la prise en charge de conceptions sur des systèmes parallèles hétérogènes car elles sont exemptes de problèmes de contention d'accès mémoire et exposent naturellement le parallélisme intrinsèque disponible. Cependant, il s'agit encore d'une question de recherche ouverte de savoir si les langages de programmation de flux de données tels que RVC-CAL peuvent s'adapter aux architectures SIMD massivement parallèles constituent les GPU. En effet, les architectures GPU récentes mettent à disposition des nombres d'unités de traitement parallèles qui dépassent de plusieurs ordres de grandeur ceux offertes par les architectures CPU. Bien que les programmes écrits à l'aide de langages de programmation de flux de données soient bien adaptés à la programmation de systèmes hétérogènes parallèles, ils peuvent ne pas offrir suffisamment de degrés de parallélisme pour

exploiter efficacement les ressources disponibles sur les GPU actuels. De plus, la nature dynamique du modèle RVC-CAL peut entrer en conflit avec la rigidité du pipeline SIMD. L'objectif de cette thèse est de développer une suite complète d'outils utilisant le langage de programmation de flux de données RVC-CAL pour fournir un processus de développement automatisé pour la programmation, l'analyse et l'optimisation de programmes s'exécutant sur des systèmes hétérogènes CPU/GPU. Les principales contributions de cette thèse sont le développement d'une infrastructure de compilateur de haut niveau qui cible les plates-formes de traitement hétérogènes CPU/GPU et prend en charge la spécification complète du langage de programmation de flux de données RVC-CAL, des fonctionnalités pour générer des applications instrumentées à des fins de profilage, et un ensemble d'options pour l'exploration de l'espace de design pour optimiser automatiquement l'application résultante en suggérant des configurations de partitionnement et de d'assignement performantes.

Mots clefs : flux de données dynamique, compilateur de source à source, RVC-CAL, systèmes hétérogènes CPU-GPU, SIMD, calcul parallèle, profilage, estimation de performance, exploration de l'espace de conception.

Contents

Acknowledgements i

List of Figures xi

List of Tables xv

List of Listings xvii

1 Introduction 1

1.1 Programming Heterogeneous Systems 1
1.2 Problem Statement and Motivation 2
1.3 Challenges of Dataflow Synthesis on GPUs 4
1.4 Research Contributions 5
1.5 Thesis Organization 6

2 State of the Art and Background 9

2.1 Introduction 9
2.2 Heterogeneous Platforms 11
2.2.1 Systems Overview 11
2.2.1.1 Embedded System 11
2.2.1.2 General Computing System 12
2.2.1.3 Data Center 13
2.2.2 GPU Systems 13
2.2.3 NVIDIA Architecture 15
2.3 Software Development Framework 16
2.3.1 Development Framework for CPU/GPU Programming 16
2.3.2 Dataflow Development Framework for CPU/GPU Programming 17
2.4 Design Space Exploration 20
2.5 Conclusion 20

3 Dataflow Programming 23

3.1 Introduction 23
3.2 Dataflow Model of Computation 24
3.2.1 Kahn Process Networks 24
3.2.1.1 Kahn Process 25

3.2.1.2 Monotonicity and Continuity . . . 25
3.2.2 Dataflow Process Network . . . 26
3.2.2.1 Actor with Firings . . . 26
3.2.3 Actor Transition System and Composition . . . 27
3.2.3.1 Enabled Transition and Step of an Actor . . . 27
3.2.3.2 Actors Composition . . . 28
3.3 CAL Actor Language . . . 28
3.3.1 Formal Definition . . . 30
3.3.2 Execution Model . . . 30
3.3.3 Syntax and Semantic . . . 32
3.3.3.1 Lexical Tokens . . . 32
3.3.3.2 Actors . . . 32
3.3.3.3 Actions . . . 33
3.3.3.4 Guards . . . 33
3.3.3.5 FSM . . . 34
3.3.3.6 Priorities . . . 34
3.3.4 Complete Example of a Program . . . 35
3.4 Open RVC-CAL Compiler . . . 38
3.5 Design Space Exploration Framework . . . 41
3.5.1 Execution Trace Graph . . . 42
3.5.2 Performance Metrics . . . 43
3.5.3 Performance Estimation . . . 45
3.5.4 Design Space Exploration . . . 47
3.6 Conclusion . . . 50

4 High-Level Synthesis of RVC-CAL Dataflow Programs on GPU 51
4.1 Introduction . . . 51
4.2 CUDA Generation Tool Flow . . . 51
4.2.1 CUDA Design Pipelines . . . 51
4.2.2 CUDA Programming Model . . . 52
4.3 Model for CPU/GPU Co-Processing . . . 56
4.3.1 CUDA Code Generation . . . 56
4.3.2 Independent GPU Actors . . . 59
4.3.3 GPU Partitions . . . 59
4.3.4 Data Communications . . . 60
4.3.4.1 RVC-CAL FIR Filter . . . 61
4.3.4.2 RVC-CAL JPEG Decoder . . . 62
4.4 Generation Features . . . 64
4.4.1 SIMD Parallel Execution . . . 65
4.4.1.1 Design Methodology . . . 65
4.4.1.2 IDCT Application . . . 68
4.4.1.3 RVC-CAL JPEG Decoder . . . 68

4.4.2 Inter-Actions Parallel Execution . . . 69
4.4.2.1 Design Methodology . . . 69
4.4.2.2 Implementation of the Inter-Actions Parallel Execution . . . 70
4.4.2.3 Efficient Parallel FIFO . . . 72
4.4.2.4 IDCT Application . . . 72
4.4.3 Dynamic Heterogeneous Actors . . . 74
4.4.4 Dynamic SIMD Parallelization . . . 76
4.5 Conclusion . . . 77

5 Automatic Design Space Exploration on GPU 79
5.1 Introduction . . . 79
5.2 Performance Estimation . . . 79
5.2.1 Clock-Accurate Profiling . . . 79
5.2.2 Static Heterogeneous Estimation . . . 81
5.2.2.1 RVC-CAL JPEG Decoder . . . 82
5.2.2.2 RVC-CAL Smith-Waterman Aligner . . . 85
5.2.3 SIMD Parallel Estimation . . . 86
5.2.3.1 RVC-CAL IDCT . . . 88
5.2.4 Dynamic Methodology Estimation . . . 88
5.3 Design Space Exploration . . . 91
5.3.1 Tabu Search . . . 91
5.3.2 Neighborhood Move Generator . . . 92
5.3.3 Design Point Evaluations . . . 93
5.3.3.1 Static Evaluation . . . 94
5.3.3.2 Dynamic Evaluation . . . 94
5.3.3.3 Measured Evaluation . . . 95
5.3.4 Results . . . 96
5.3.4.1 IDCT Example . . . 96
5.4 Conclusion . . . 100

6 Additional Experimental Results 101
6.1 Introduction . . . 101
6.2 Experimental Setup . . . 101
6.3 Experimental Applications . . . 102
6.3.1 HEVC Decoder . . . 102
6.4 Model Validation . . . 102
6.5 Conclusion . . . 104

7 Conclusions and Future Work 105
7.1 Conclusion . . . 105
7.2 Future Work . . . 106
7.2.1 High-Level Synthesis of RVC-CAL Dataflow Programs . . . 107
7.2.1.1 Dataflow Compiler Extensions . . . 107

7.2.1.2 Features for Performance Improvement 108
7.2.1.3 Auto-Tuning Programs 109
7.2.1.4 Profiled Application Generation 109
7.2.2 Design space exploration 109

Bibliography **119**

Curriculum Vitae **121**

List of Figures

2.1 Illustration of the computational scalability trends in both industry and academia, showcasing scale-up, scale-out, and heterogeneous approaches. 10

2.2 Comprehensive high-level overview of the Jetson Nano architecture [1] 11

2.3 Comprehensive high-level overview of the Qualcomm QCA4020 architecture [2] 12

2.4 Comprehensive high-level overview of the Apple M1 Max architecture [3] 12

2.5 Comprehensive high-level overview of the Microsoft Catapult architecture [4] . 13

2.6 Comprehensive high-level overview of the NVIDIA Grace Hopper architecture [5] 14

2.7 Comprehensive high-level overview of a GPU architecture [6] 14

2.8 Comprehensive high-level overview of the NVIDIA GPU architecture [7] 15

3.1 Representation of a CAL application network comprising five actors and five FIFO buffers, along with a detailed view of actor internals featuring finite state machine, actions and internal variables. 29

3.2 Visual representation of the actor's execution model considered in this thesis. . 31

3.3 An example of a dataflow network with five actors (i.e. *Prod*, *CopyTokenA*, *CopyTokenB*, *PingPong* and *Merger*). 36

3.4 Execution of the CAL example with all actors running in separate threads on a CPU. 38

3.5 Overview of the code generation process within the ORCC compiler framework. 41

3.6 Overview of the parameter optimization process within the TURNUS design space exploration framework. 43

3.7 Execution trace graph obtained after the execution of the RVC-CAL program described in Figure 3.3 if the guard of the counter of the Prod actor was set to 2. The firing set S is summarized in Table 3.1, and the dependencies set D is summarized in Table 3.3. 44

4.1 Overview of the code generation process within the ORCC compiler framework combined with the Exelixi CUDA backend. 52

4.2 Structures of the folders and files generated by the Exelixi CUDA backend. . . . 53

4.3 Illustration of the automatic scalability allowed by the CUDA programming model [8]. 54

4.4 Illustration of the memory hierarchy used in CUDA compatible hardware [8]. . 54

4.5 Illustration of how a typical CUDA codes is organized with sequential CPU section and parallel GPU code [8]. 55
4.6 Program partitioning over CPU and GPU. Four types of FIFOs are used (1-CPU FIFO, 2-Legacy-HostFifo, 3-HostFifo, 4-Device FIFO. 58
4.7 Example of the execution of the dataflow program depicted in Figure 3.3. In this example, four tokens are generated by the producer actor, and actors are mapped over both a CPU and a GPU, using multiple CUDA streams to showcase multiple independent actor executions. 59
4.8 RVC-CAL FIR filter dataflow network: actors and communication FIFO buffers. 61
4.9 Speedup results for the RVC-CAL FIR filter. On the x-axis the different mapping configurations and on the y-axis the speedup value with the application implemented using the new FIFO inter-partition methodology. 62
4.10 Network for the RVC-CAL implementation of a JPEG decoder. 64
4.11 Speedup results for the RVC-CAL JPEG decoder. On the x-axis the different mapping configurations and on the y-axis the speedup value with the application implemented using the new FIFO inter-partition methodology. 64
4.12 Illustration of the test RVC-CAL IDCT dataflow network. 68
4.13 Example of the execution of the dataflow program depicted in Figure 3.3 with all actors mapped to the GPU and utilizing the SIMD Parallel Execution feature. . 70
4.14 Example of the execution of the dataflow program depicted in Figure 3.3 with all actors mapped to the GPU and utilizing both the SIMD Parallel Execution feature and the inter-action parallelization optimization. 71
4.15 Parallel schema showcasing the parallel execution of two GPU actors and four actions by using separated cudaStream. 71
4.16 Illustration of the parallel reading process implementation in the FIFO buffer, with the writing process functioning similarly. 74
4.17 Illustration comparing the dynamic and static implementations of a dataflow network. 76
4.18 Representation of the SIMD parallel read/write of a FIFO buffer. 77

5.1 Illustration of the breakdown of an action's generated code: inputs are read sequentially, followed by the action body executing the processing, and concluding with the writing of outputs. 81
5.2 Illustration of the four static configurations required during profiling. 83
5.3 Normalized comparison between the estimated and measured total runtime of the JPEG application with height inputs and two FIFO buffer configurations (512 and 1024 tokens). The identity line in orange corresponds to the 1:1 line for visual reference. 84
5.4 Normalized comparison between the estimated and measured total runtime of the JPEG application with all 16 possible partitions and two FIFO buffer configurations (512 and 1024 tokens). The identity line in orange corresponds to the 1:1 line for visual reference. 85

5.5 Graphical representation for the Smith-Waterman aligner application in RVC-CAL. . . . 85
5.6 Normalized comparison between the estimated and measured total runtime of the Smith-Waterman aligner application with four inputs and two FIFO buffer configurations (256 and 1024 tokens). The identity line in orange corresponds to the 1:1 line for visual reference. . . . 87
5.7 Normalized comparison between the estimated and measured total runtime of the Smith-Waterman aligner application with all 1024 possible partitions and two FIFO buffer configurations (256 and 1024 tokens). The identity line in orange corresponds to the 1:1 line for visual reference. . . . 87
5.8 Comparison between the measured and estimated normalized total elapsed execution time with varying numbers of SIMD threads when executing the IDCT application. . . . 89
5.9 Design flow with dynamic networks. Deep blue elements represent the implementation path of the tool, while light blue the profiling and design space exploration path of the tool. . . . 90
5.10 Dynamic heterogeneous actor methodology injected with instrumentation code so as to generate performance weights. . . . 90
5.11 Generic design flow when doing design space exploration using Tabu search. . . 93
5.12 Design flow when doing design space exploration using TS and the static evaluation strategy. . . . 95
5.13 Design flow when doing design space exploration using TS and the dynamic evaluation strategy. . . . 96
5.14 Design flow when doing design space exploration using TS and the measured evaluation strategy. . . . 97
5.15 Illustration of the test RVC-CAL IDCT dataflow network. . . . 98

6.1 Illustration of the test RVC-CAL HEVC decoder dataflow network. . . . 103
6.2 Performance comparison of the RVC-CAL HEVC decoder implementation across three hardware platforms for 33 different partitions, with each partition having a single different actor executing on the GPU and the remaining actors on the CPU. 103

List of Tables

3.1 Firing of the RVC-CAL program described in Figure 3.3 if the counter of the Prod actor was set to 2. 44
3.2 Dependencies kinds, directions, parameters and additional attributes. 45
3.3 Dependencies set S of the execution trace graph depicted in Figure 3.7. 46

4.1 Statistics of the results depicted in Figure 4.9a using 10 executions. 63
4.2 Statistics of the results depicted in Figure 4.9b using 10 executions. 63
4.3 The different mappings settings of the FIR implementation that are used in the results. 63
4.4 Statistics of the results depicted in Figure 4.11a using 10 executions. 65
4.5 Statistics of the results depicted in Figure 4.11b using 10 executions. 65
4.6 The different mappings settings of the JPEG decoder implementation that are used in the results. 65
4.7 Speedup results with statistics for the the RVC-CAL IDCT application using 10 executions. 68
4.8 Frame rate and speedup results for the RVC-CAL JPEG decoder. 69
4.9 Performance results of the IDCT application example, showcasing the impact of increasing the number of inter-action parallel executions from one to sixteen using multiple CUDA streams. 74

5.1 The different resolutions and quality factors of the input images used for the JPEG Decoder results. 84
5.2 Summary table of results with three evaluation methods (Static, Dynamic, Measured), and four neighboring move generators. 99
5.3 Mapping of the bests/worst partitions between CPU and GPU and there corresponding performances. 99

Listings

3.1 Example of the actor's syntax 32
3.2 Example of the action's syntax 33
3.3 Example of the guard's syntax 33
3.4 Example of the FSM's syntax 34
3.5 Example of the priority's syntax 35
3.6 Description of the dataflow network using the XML Dataflow Format (XDF) . . 36
3.7 CAL implementation of the Producer actor. 37
3.8 CAL implementation of the CopyTokens actor. 37
3.9 CAL implementation of the PingPong actor. 37
3.10 CAL implementation of the Merger actor. 38
3.11 Example of an XCF file specifying the partitioning and mapping configuration for a dataflow program. 39
3.12 Example of a BXDF file specifying the sizes of all FIFO buffer connections in a dataflow program. 40
3.13 Example of an EXDF file specifying the execution weights extracted from a dataflow program. 47
3.14 Example of an SXDF file specifying the scheduling weights extracted from a dataflow program. 48
3.15 Example of an CXDF file specifying the communication weights extracted from a dataflow program. 49
4.1 Striped down example of the *CUDA* implementation of the action selection of the *Producer* actor 58
4.2 Simplified implementation of a *CUDA* action selection function for an actor of the *idct* design illustrated in Figure 4.12 67
4.3 Stripped-down example of the *CUDA* implementation of the action selection of the *Producer* actor 73
5.1 Simplified example of the utilization of NVIDIA's assembly language (i.e. *SASS*) was required for reading the GPU's stable autoincrementing register. 81
5.2 Simplified example of the utilization of NVIDIA's assembly language (i.e. *SASS*) was required for reading the GPU's stable autoincrementing register in parallel mode. 88

1 Introduction

This thesis presents research aimed at developing a comprehensive high-level compiler design flow that is based entirely on the dataflow model of computation. The tool flow encompasses capabilities and optimizations specifically designed for CPU/GPU heterogeneous processing systems. Despite advancements in silicon technology, individual sequential processors are not advancing fast enough to keep up with the demand for processing power. The traditional approach of scaling up and scaling out by increasing the number of cores per chip and the total number of chips in a data center is no longer sufficient. Instead, the trend is towards more complex parallel heterogeneous architectures where sub-elements are specialized to perform specific processing tasks more efficiently, whether it be in terms of latency, throughput, or energy consumption. One of the biggest challenges in effectively utilizing these platforms is that traditional sequential specification formalism and software, developed for sequential processor architectures, are not well suited for programming these parallel platforms. Furthermore, these specifications are not appropriate for unified specifications targeting any type of hardware component, such as conventional processors and SIMD parallel architectures. With the precise number, nature, and configuration of possible sub-systems varying greatly when targeting a full heterogeneous system, the portability of applications across different platforms is becoming increasingly important, but is not adequately addressed by traditional sequential behavioral descriptions and methodologies.

1.1 Programming Heterogeneous Systems

The increasing demand for computing power can partially be addressed by the use of heterogeneous systems. However, to fully tap into their potential, design flows must be able to support these heterogeneous architectures. The challenge lies in the fact that each sub-architecture has typically been developed for specific use cases, and comes with its own software stack, programming language, frameworks, and development methodology. As a result, engineers often specialize in developing for just one or a few platforms. This makes it extremely challenging to create a coherent development strategy that integrates all these architectures to work together towards a common goal.

Another critical challenge in the field of heterogeneous computing is to effectively choose the most appropriate sub-architecture to compose a system, given a behavioral description of the algorithm. This decision has a direct impact on the performance of the system, and thus, it is of utmost importance to make a well-informed choice. In addition to choosing the right sub-architecture, it is also crucial to determine which processing units will execute the different sub-parts of the overall application software. This decision must be made carefully, taking into account the characteristics of the processing units, such as their processing power, memory, and energy consumption, and the requirements of the sub-parts of the application software. Moreover, it involves the use of optimization techniques and tools to make informed decisions, taking into account the specific requirements of each application.

Portability is another crucial aspect in programming heterogeneous systems as the hardware available for the same application can vary greatly based on factors such as resource availability, client needs, power consumption, and costs. This means that developers may need to write multiple versions of an application to ensure that all potential platform configurations are utilized efficiently and effectively. This can result in significant modifications or rewriting of the code for each case, making portability important in saving time, effort, and resources. Additionally, portability helps ensure that code is consistent and reliable across different systems, and remains relevant and usable in the future as improvements in hardware can result in significant changes to the overall architecture, requiring redesigns.

Finally, each architectural sub-element is a parallel processing unit, and some hardware architectures are even inherently parallel in nature. This presents the challenge of extracting the parallelism opportunities from the behavioral description of the target application in order to effectively and efficiently utilize all the processing elements available and improve the overall performances. This is particularly important in CPU/GPU heterogeneous systems, where the number of parallel resources is enormous.

1.2 Problem Statement and Motivation

The challenges in programing heterogeneous CPU/GPU processing platform presented in the previous section can be formalized as the problems of *Abstraction*, *Concurrency*, *Modularity*, *Analyzability* and *Portability*. This thesis as the purpose to demonstrate that the RVC-CAL dataflow programming language is a good candidate for building tools solving these issues.

The RVC-CAL programming language is a high-level programming language that provides software architects with the necessary mechanisms to represent algorithms in terms of parallel constructs directly extracted from the data. Additionally, the language's modular and hierar-

chical structure facilitates the decomposition and analysis of complex systems. The language also provides formal semantics that enable static analysis and verification of programs.

The RVC-CAL methodology addresses the following categories in the following ways:

- **Abstraction:** The RVC-CAL programming language provides a high level of abstraction for designing and implementing concurrent signal processing applications. It allows users to express their algorithms using a dataflow programming model, abstracting away low-level details such as memory allocation and synchronization.
- **Concurrency:** The RVC-CAL programming language supports concurrency through the use of actors that are independent and parallel computing nodes that communicate with each other through channels without the need for synchronization.
- **Modularity:** The RVC-CAL programming language provides modularity at its core as programs are created through hierarchical composition of actors, which allows for the creation of complex systems from simpler sub-components. These actors can be reused across different projects and systems, improving code reuse and reducing development time.
- **Analyzability:** RVC-CAL achieves the ability to reason about the correctness and behavior of a program by providing a formal semantics that defines the behavior of the language constructs. This semantics enables formal verification tools to be used to verify the correctness of RVC-CAL programs. Additionally, RVC-CAL supports static type checking, which helps to detect type errors at compile time, and provides runtime monitoring, debugging facilities and design space exploration frameworks for performance tuning.
- **Portability:** RVC-CAL programs are described in a platform-independent manner, making them easily portable across diverse hardware and software platforms. This portability is achieved through the use of a compiler backend that automatically generates low-level implementations optimized for specific targeted architectures.

Thesis problem statement: *The dataflow programming model of computation possesses the essential characteristics required for CPU/GPU heterogeneous computing as it offers abstraction, concurrency, modularity, analyzability and portability.*

To support the problem statement of this thesis, the following contributions are presented:

- **Abstraction:** This thesis ensures that the details of the CPU or GPU platform are fully abstracted away so that the same actor can run on both platforms without any changes to the code.
- **Concurrency:** In addition to supporting multicore processing, this thesis enhances the concurrency of the system by fully supporting the SIMD nature of GPU architecture.

Furthermore, the potential for concurrent processing tasks is increased by developing SIMD and inter-action parallel execution methodologies within the actors.

- **Modularity:** The methodology resulting from this thesis fully support the composition of actors targeted to the CPU or GPU architecture. It strictly maintains the modular model even for actor with internal parallelization.
- **Analyzability:** The performance estimation and design space exploration methodologies are extended to support GPU systems.
- **Portability:** This thesis enhances portability by including the GPU platform as one of the available sub-elements for composing a heterogeneous system.

1.3 Challenges of Dataflow Synthesis on GPUs

Compared to CPU architecture, synthesizing code for GPUs presents a variety of significant challenges.

In this thesis, several challenges related to the synthesis of code for RVC-CAL actors on GPU platforms are set to be resolved. First and foremost, the goal is to provide full support for the dynamic dataflow model of computation in order to enhance expressiveness and enable easier representation of complex applications. This task is not trivial, as GPUs possess more rigid architectures and programming interfaces, introducing limitations that must be overcome. One notable example of such a limitation is the handling of memory, which needs to be managed efficiently.

Secondly, for performance reasons, the goal is to surpass the traditional OpenCL model of computation, where the CPU schedules tasks and waits for the results. This can be achieved by enabling GPU actors to fully execute on the GPU platform, including both scheduling and action execution. By doing so, we can eliminate the need to transfer data back and forth for making scheduling decisions, which frees up the CPU to execute other actors that do not benefit from GPU execution.

It is crucial to execute multiple actors in parallel, as a single actor running on a GPU is insufficient to efficiently utilize the GPU resources. The execution of both CPU and GPU actors must be overlapped while reducing or minimizing the use of expensive synchronization mechanisms. Additionally, achieving an overlap between GPU processing and memory transfers from CPU to GPU, as well as from GPU to CPU, is essential to maintain continuous computation of the dataflow graph and achieve competitive performance.

Another significant challenge is to research and develop techniques that maximize resource utilization, as the dataflow model may not inherently provide a level of parallelism that matches the available hardware resources.

Ultimately, this study aims to propose a methodology for effectively implementing the First-

In-First-Out (FIFO) mechanism, which ensures consistency and enhances performance in the dataflow model of computation.

1.4 Research Contributions

The principal contributions and their corresponding publications in the thesis can be summarized as follows:

- Development of a high-level synthesis compiler infrastructure that targets CPU/GPU heterogeneous processing platforms and supports the full specification of the RVC-CAL dataflow programming language. The compiler infrastructure is based on the open-source ORCC compiler and the Exelixi backend, with the addition of a new CUDA backend [9, 10].

- Implementation of a set of compiler optimizations within the newly designed Exelixi CUDA backend to enhance the performance and functionality of the generated low-level implementations. The features include:

    1. Improvement of performances by minimizing data transfer between actors by using through the utilization of composite data structures [11].
    2. Utilization of Single Instruction Multiple Data (SIMD) parallelization techniques to speed up the runtime of dataflow actions, resulting in improved overall execution performance. This is accomplished through parallel execution of multiple instances of the same action using CUDA [12].
    3. Utilization of the dynamic programming feature provided by the CUDA API to implement inter-action parallel execution techniques to speed up the runtime of dataflow actions, leading to improved overall execution performance [13].
    4. Introduction of a methodology for generating dynamic RVC-CAL networks. With this technique a single executable binary is created that allows for the specification of the partitioning and mapping at runtime during the application's startup process [14].
    5. Development of a dynamic SIMD parallelization optimization that involves generating multiple SIMD parallel executions of the same action. The number of threads used could change dynamically throughout the runtime of the application, with the objective of maximizing performance and maximizing utilization of GPU resources [15].

- Improvement over the Exelixi CUDA backend for generating instrumented code for clock-accurate profiling and outputting the corresponding performance metrics, that can be utilized in the TURNUS post-processor to estimate the overall application execution time accurately [16, 14].

- Development around the design space exploration tool TURNUS to adapt the methodology to the CPU/GPU model of this thesis [17].

1.5 Thesis Organization

This thesis dissertation is divided into seven chapters that are organized as follows:

Chapter 2 portraits the state of the art and background regarding heterogeneous CPU/GPU platforms. An introduction to different available heterogeneous platforms is depicted. Moreover, a description of the literature regarding the different frameworks and development available for programming CPU/GPU heterogeneous platforms is depicted, for both generic and more specifically dataflow frameworks. Finally, the existing design space exploration tools relevant in this context are shown.

Chapter 3 provides the background regarding dataflow programming necessary for the understanding of the presented thesis. First of all, the dataflow model of computation is defined, and the specific RVC-CAL dataflow programming language used in this thesis is presented. Finally, the two frameworks that this work extends are depicted, namely the ORCC dataflow compiling suite and the TURNUS design space exploration framework.

Chapter 4 is one of the core chapters presenting the developed work. It presents in depth the tool flow and model created to allow dataflow models to properly execute on CPU/GPU heterogeneous processing platforms. Additionally, a list of generation features and optimizations built on top of the model is presented. This includes SIMD parallel execution, inter-actor parallel execution, dynamic heterogeneous actors, and dynamic SIMD parallelization.

Chapter 5 is the second core chapter, dedicated to improving design space exploration tools applied to CPU/GPU heterogeneous processing platforms. In the first part, the methodology for obtaining clock-accurate profiling metrics is presented. Secondly, three important methodologies for automated performance estimation are presented. Finally, the aforementioned methodologies are used in a Tabu-Search algorithm to perform computer-assisted design space exploration with the purpose of automatically finding actor partitions and mapping configurations that would yield promising performances.

Chapter 6 presents the experimental results performed using the combination of the innovative components developed in this thesis work. The goal is to demonstrate how a developer can benefit from the tools created or extended in this work to increase their productivity regarding CPU/GPU co-development.

Chapter 7 concludes the manuscript, summarizes the achievements of this work, highlights possible future developments, and illustrates the open problems that would benefit from further research.

2 State of the Art and Background

2.1 Introduction

The growing needs of today's applications is still driving the search for higher computational capacity of processing platforms. As moore's law is getting more difficult to satisfy in terms of new process node technology and as the increase in logic gates switching frequency is getting smaller over time, the processing power needs is not being met. This facts pushes the roll-out of heterogeneous processing platforms. Indeed, exploiting deeper levels of domain-specific hardware specialization is considered as a solution to the fact that the straightforward multiplication of similar and off-the-shelf processing elements is not sufficient anymore to deliver the necessary increase in processing power. More recently, industries are shifting to increasingly complex heterogeneous hardware for custom usage (i.e, Microsoft ARM [18], NVIDIA Grace [19], Apple silicon [20]). They are clear attempts to answering these increased processing needs, however, they introduce new challenges and the need to cope with unresolved problems in how to efficiently program them.

The development of heterogeneous system to help with providing more efficiency and processing capabilities as a field of research can be broken down in three different parts:

- **Complex architectures**: Processing architecture is getting more complex in all scales. Whether it is in embedded system, general computing or data centers, heterogeneous processing elements are use to improve power consumption and raw performances. Such architecture can be compose with a lot of different processing element such as FPGA, CPU, GPU, tensor accelerator or other type of hardware accelerator.
- **Software platforms**: As the complexity of these architecture increase, heterogeneous system are getting more difficult to program, analyze and optimize. This fact brings the necessity for higher level of abstraction using advanced software framework to help leveraging this specialized processing capabilities.
- **Design exploration tools**: The pic performance could only be achieved if the processing

problems at hand are decomposed into suitable sub-tasks that are then assign to the proper processing element of the heterogeneous hardware. The space of design grow extremely quickly with the complexity of the task and the complexity of the heterogeneous hardware. This leads to the need for suitable framework helping with such task decomposition and assignment.

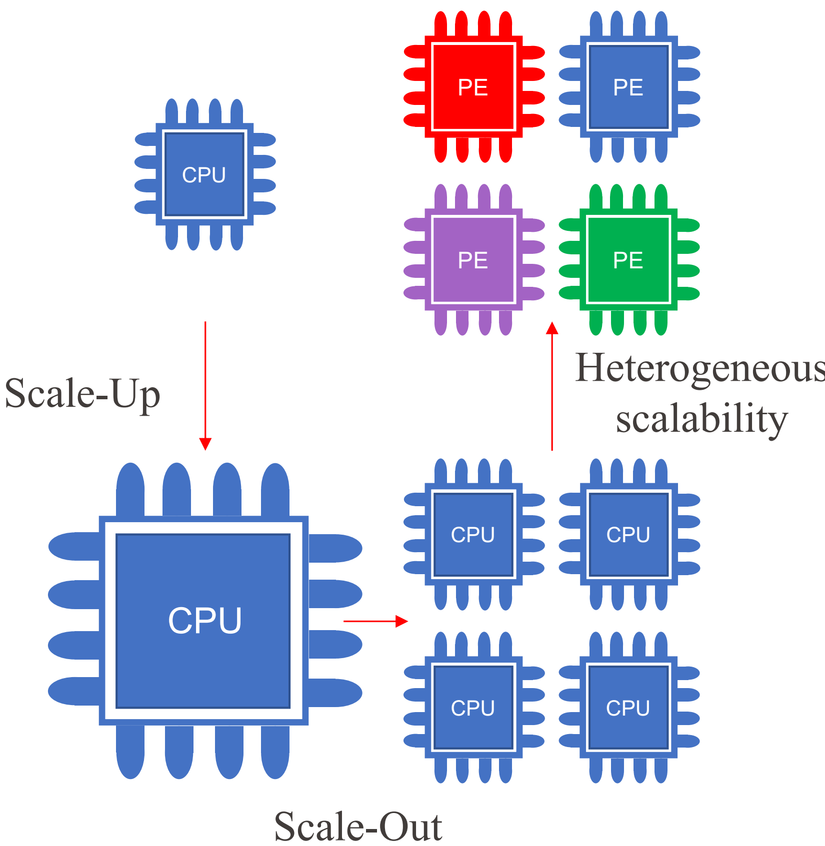


Figure 2.1: Illustration of the computational scalability trends in both industry and academia, showcasing scale-up, scale-out, and heterogeneous approaches.

The rest of this chapter presents a summary of the state-of-the art related to heterogeneous system related to the context of this thesis that partially solves the stated problems. In the following, the different heterogeneous system architectures that are used to solve the performance gap are presented. In addition, the programing languages, software stacks and development framework developed in the intent of leveraging these complex architecture are depicted. This chapter concludes on the work done toward tool providing automated guidance in the adequate distribution of tasks to the different subsystem of heterogeneous architecture.

2.2 Heterogeneous Platforms

2.2.1 Systems Overview

The industry is developing many different and more complex heterogeneous system. These processing system are composed of different well known processing elements. These elements can encompass but are not limited to CPU core, FPGA, GPU, DSP, tensor accelerator, NoC, Modem, Fixed-function unit, or other type of hardware accelerators. With there popularization, these processing system are used in all product category an can be divided in three application types such as Embedded system, general purpose computing and data center. The following subsection is going to show a few example of such system for illustration.

2.2.1.1 Embedded System

In this category the heterogeneous aspect is there to either get really low power consumption or to get powerful embedded system with the idea that power consumption is a major point of design. Bellow are two example of such architecture. Figure 2.2 shows the architecture of NVIDIA Jetson Nano, a powerful embedded system specialize for AI application that embed in part, a quad core ARM CPU and a 128 CUDA tensor cores GPU. On the other end of the spectrum, Figure 2.3 shows the architecture of the Qualcomm QCA4020, a very low power embedded system for Internet-of-Things device with low power Arm Cortex-M4F CPU with maximum 128 MHz, security accelerators for encryption and co-processor for communications.

Figure 2.2: Comprehensive high-level overview of the Jetson Nano architecture [1]

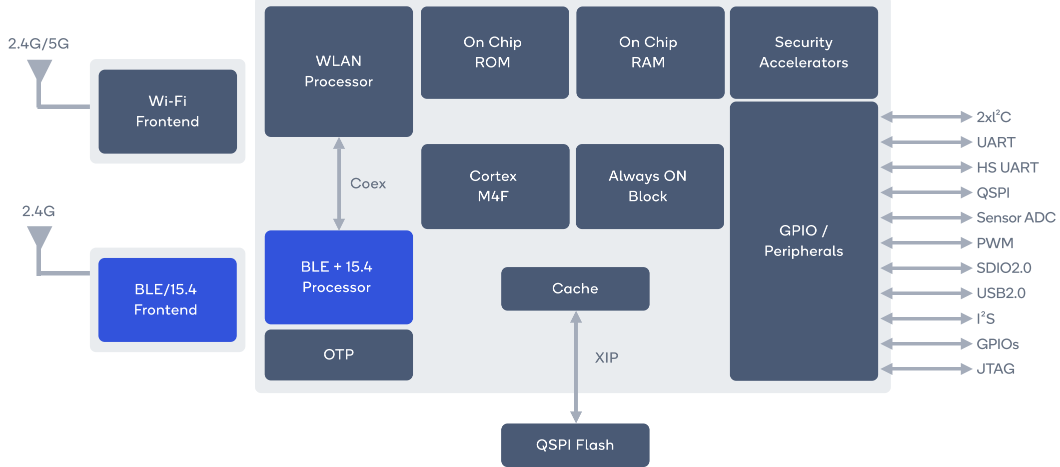


Figure 2.3: Comprehensive high-level overview of the Qualcomm QCA4020 architecture [2]

2.2.1.2 General Computing System

In this category the heterogeneous aspect is there to either get a good power consumption to performance ratio for portable device running out of battery or to get processing power that would not otherwise be possible in these form-factor. Figure 2.4, shows the architecture of the Apple M1 Max architecture that contains many different processing elements such as high-performance and high efficiency CPU cores, GPU cores, Neural engine and media engine.

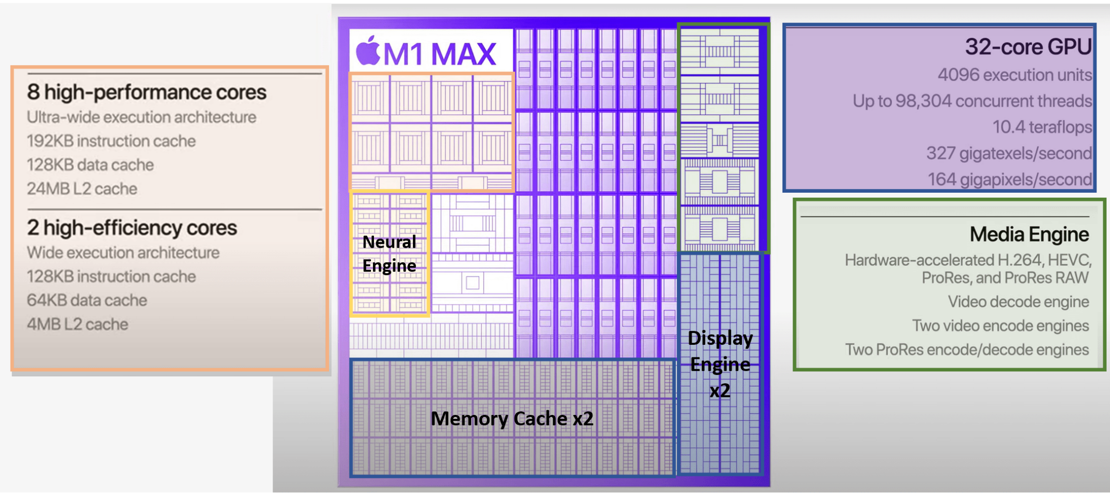


Figure 2.4: Comprehensive high-level overview of the Apple M1 Max architecture [3]

2.2.1.3 Data Center

In this category the heterogeneous aspect is there is two fold. On once side it provides the possibility to keep increasing the overall processing power of the data center without increasing the overall power consumption. On the other side it reduce the fraction of the energy spend in power dissipation cooling these device leaving more energy budget for processing. Figure 2.5, shows the architecture of the Microsoft Catapult dual socket server blade that also encompass a on-board FPGAm while Figure 2.6 shows the architecture of the new NVIDIA Grace Hopper Superchip with up to 72 Arm CPU core and a GPU with 18432 FP32 CUDA Cores and 576 Tensor cores.

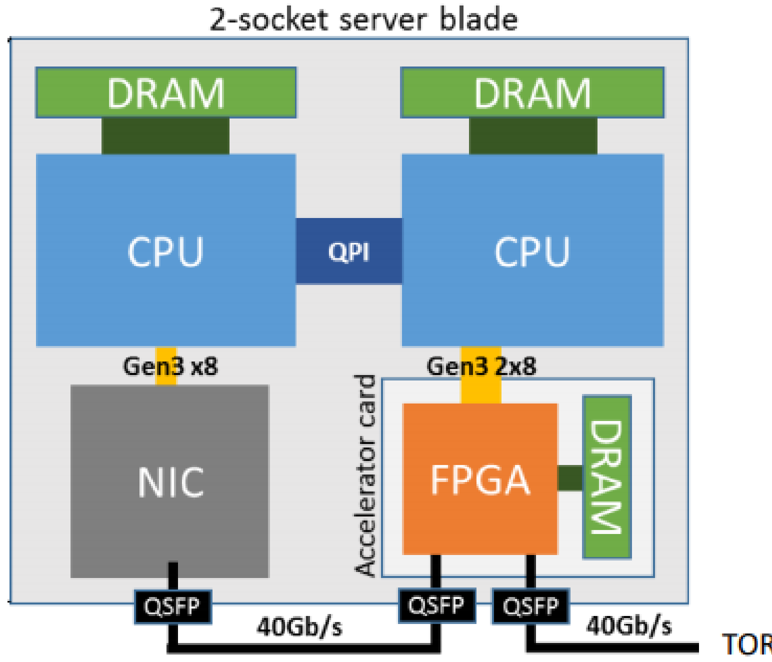


Figure 2.5: Comprehensive high-level overview of the Microsoft Catapult architecture [4]

2.2.2 GPU Systems

In this thesis we focus on specific heterogeneous combination, CPU/GPU co-processing platforms. A Graphics Processor Unit (GPU) is primarily known for its usage in applications that massively rely on graphics processing such a video game or 3D modeling. Today, GPGPU's (General Purpose GPU) are the choice of platform to accelerate parallel workloads in modern High Performance Computing (HPC) architecture.

Figure 2.7, shows how the architecture of CPUs and GPUs differs. CPUs are optimize to be as quick as possible as finishing tasks, while keeping the ability to quickly switch between operations. They are low latency optimized. They usually contains a small amount of cores that each have their own control units, L1 data, and instruction cache. On the other hand,

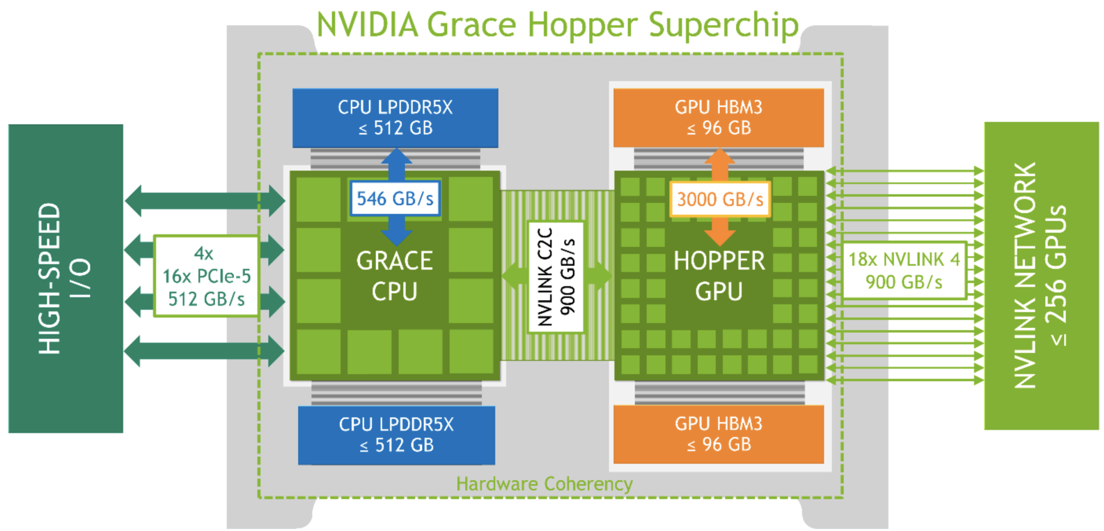


Figure 2.6: Comprehensive high-level overview of the NVIDIA Grace Hopper architecture [5]

GPUs are optimized for throughput allowing to push as many as possible tasks through their internals at once. They are usually composed of magnitudes more number of cores that are divided in groups each sharing the same instruction and data level one cache. Relatively to the number of cores they contains less caches than a CPU, as the memory latency is less of a priority as long as enough bandwidth is available to keep the core busy with processing tasks.

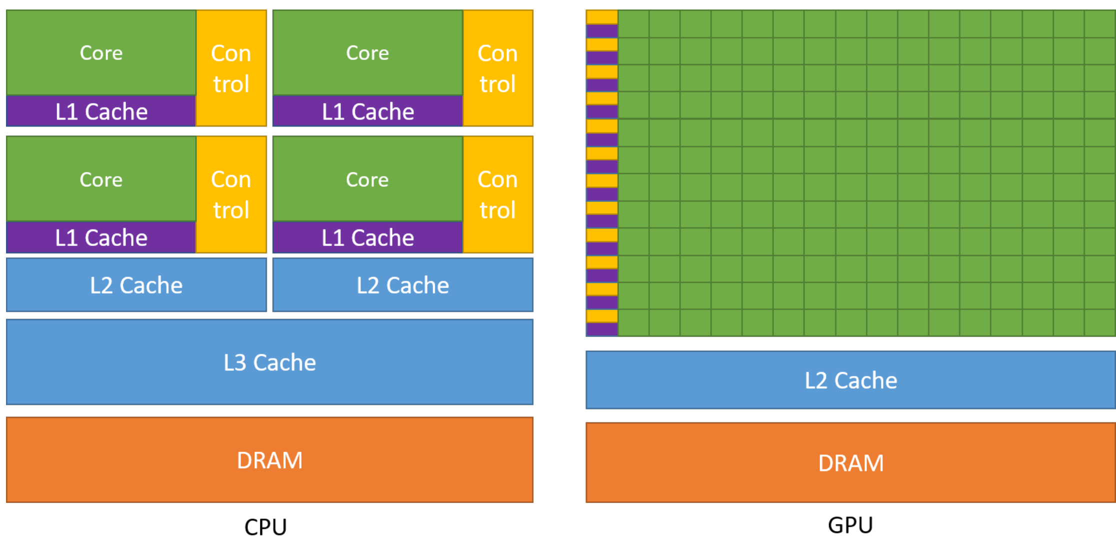


Figure 2.7: Comprehensive high-level overview of a GPU architecture [6]

2.2.3 NVIDIA Architecture

In the practical part of this thesis we focus on NVIDIA architecture as we are using the CUDA APIs. In this section we present a little bit the architecture and more importantly introduce the vocabulary that will be used in this work. Figure 2.8 Show an high-level overview of the architecture of an NVIDIA GPU. NVIDIA GPUs develop an architecture that we call SIMD for Single Instruction Multiple Data or interchangeably we can find in the literature, SIMT for Single Instruction Multiple Thread. This means that a collection of cores will be used to execute the exact same task in parallel using different data as input. These core are called CUDA cores. CUDA cores are organized as groups called Streaming Multiprocessors (SM). Depending of the micro-architecture (generation model) the number of CUDA cores per SM varies and depending of individual model inside a generation, the number of SM per GPU varies. Each SM contains in addition, warps scheduler (responsible of scheduling instructions to groups or up to 32 cores), L1 data cache / shared memory and a register file. The onboard GPU DRAM is called Global Memory.

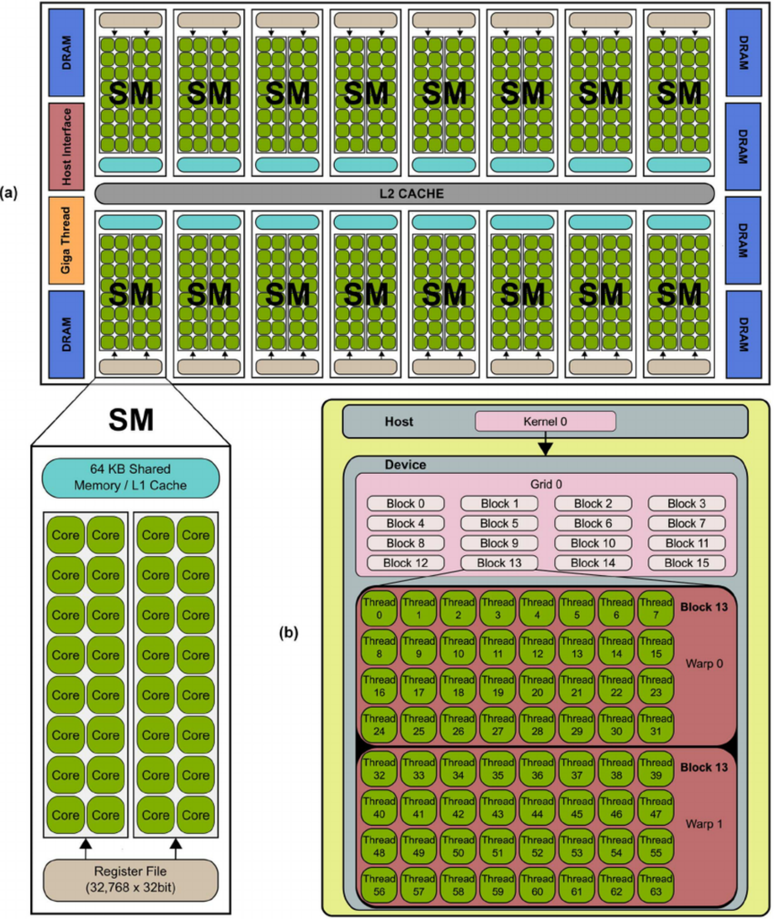


Figure 2.8: Comprehensive high-level overview of the NVIDIA GPU architecture [7]

2.3 Software Development Framework

2.3.1 Development Framework for CPU/GPU Programming

There exist a plethora of cross-platform CPU/GPU development framework this section will introduce some of the major one use in the literature. The one that as been selected for this work is the NVIDIA CUDA framework and is described in more details in Section 4.2.2.

OpenCL: (Open Computing Language) is a royalty-free open standard for parallel programming across heterogeneous platforms. OpenCL works with CPU, GPU, DSP, FPGA and other processors or hardware accelerators. It is based on C/C++ language, with third-party wrappers also available for Python, Java, R, GO, JavaScript, and many others [21].

SYCL: is a royalty-free, higher-level, single-source embedded domain-specific programming model that improve programming productivity for various heterogeneous processors. It allows developers to support various CPUs, GPUs, and field-programmable gate arrays (FPGAs). SYCL achieves this by using single-source code written in standard ISO C++, that does not require C++ extensions and rely on pure runtime rather than a particular compiler [22].

DPC++: (data parallel C++) is a programming language based on C++ that is designed for parallel computing. It allows developers to write code that can be run on a variety of hardware platforms, including CPUs, GPUs, and other specialized devices. DPC++ includes a number of features and libraries that make it easy to write efficient parallel code, such as support for parallel algorithms, thread-safe memory management, and data-parallel execution models. It is intended for use in high-performance computing applications such as machine learning, scientific simulations, and image processing. It is an open source project of Intel to introduce SYCL for LLVM and oneAPI. C++17 and parts of C++20 with SYCL 2020 are base of this Compiler framework (part of oneAPI) [23].

OpenACC: is a user-driven directive-based parallel programming standard designed to simplify the process of programming heterogeneous CPU/GPU systems and architectures with less effort than required with a low-level model. Developers can annotate C, C++, and Fortran source code to accelerate certain areas and the goal is to provide a model for accelerator programming that is portable across operating systems and various types of host CPUs and accelerators. OpenACC is intended for scientists and engineers interested in porting their codes to a wide variety of heterogeneous high-performance computing (HPC) hardware platforms [24].

OpenMP: (Open Multi-Processing) is a set of compiler directives, library routines, and environment variables that enable shared memory parallelism in C, C++, and Fortran programs. It is a portable, scalable model that provides a simple and flexible interface for developing parallel applications on platforms that range from embedded systems to supercomputers.

The main advantage of OpenMP is that it is relatively easy to use, particularly for developers who are already familiar with the language in which they are programming. It also has a large user community and is supported by a wide range of compilers and tools. However, OpenMP may not be the best choice for all types of parallel applications, as it is primarily designed for shared memory systems and may not perform well on distributed memory systems, even thought latest versions does offer support for SIMD architecture [25].

Halide: is a programming language designed for writing high-performance image and array processing code on modern machines. It is particularly well-suited for tasks that can be parallelized, such as applying the same operation to every element of an image or array. Halide is implemented as a domain-specific language (DSL) C++ library and includes a compiler that can generate optimized code for a variety of platforms, including CPUs, GPUs, and specialized processors such as DSPs. One of the key features is th separation between the algorithm and its execution schedule, which enables the programmer to easily experiment with different scheduling approaches without having to modify the algorithm itself [26].

HIP: is a programming language and runtime system that allows developers to write code that can run on both AMD and NVIDIA graphics processing units (GPUs). It is designed to be a portable and high-performance alternative to the CUDA programming model, which is specific to NVIDIA GPUs. HIP is implemented as a C++ library and includes a compiler that can translate HIP code into the native programming languages of AMD and NVIDIA GPUs, such as OpenCL and CUDA, respectively. This allows developers to write code that can be compiled and run on either type of GPU without having to rewrite it for each platform. HIP is intended to make it easier for developers to take advantage of the parallel processing capabilities of GPUs without being tied to a specific vendor or platform [27].

2.3.2 Dataflow Development Framework for CPU/GPU Programming

In the context of programming CPU/GPU heterogeneous systems using the CAL programming language, several approaches can be found in the literature. The works reported in [28] and [29] offer two different approaches to using OpenCL to program GPU platforms from RVC-CAL application programs. In the first work, OpenCL is generated from the ORCC intermediate representation (IR), whereas in the second work, the distributed application layer IR called DAL [30] is used for the final synthesis.

Both works first classify each actor of the dataflow program according to its more restricted MoC in terms of dynamic degree, and map only SDF MoC actors [31] to be executed on the GPU platform. Dynamic actors (DPN) are mapped to the available CPU cores.

The authors of [32] propose to generate SYCL from RVC-CAL dataflow programs. SYCL is a cross-platform abstraction layer based on C++ that enables code for heterogeneous processors to be written in a single-source with the advantages of OpenCL. Their approach consists of

using the Single Instruction Multiple Data (SIMD) nature of GPUs to run in parallel multiple consecutive firings of the same action when there is available data for firing it. In their work, they describe the conditions under which such optimization is possible.

The principal differences of the solution described in this work from other dataflow-to-GPU methodologies that have appeared in the literature are that the GPU execution is not limited to actions, but also the action selection computation is fully executed on the GPU. Furthermore, the mapping of actors on GPUs applies to any type of actor, including DPN, compare to the proposed solutions that only support actors with SDF MoC. Such an approach enables any sub-network partition of a dataflow application program to be mapped and executed on a GPU platform, regardless of the dynamic behavior of its actors.

Besides approaches specifically targeting CAL compilation, other research article using the dataflow methodology to program CPU-GPU processing platform can be found.

The article [33] discusses the XKaapi runtime system for programming heterogeneous high-performance computing (HPC) platforms with multicore CPUs and accelerators, such as GPUs. XKaapi supports a data-flow task model and a locality-aware work stealing scheduler that enables multi-implementation on CPU or GPU and multi-level parallelism with different grain sizes. The authors report performance results on dense linear algebra kernels, matrix product, and Cholesky factorization, on a heterogeneous architecture composed of two hexa-core CPUs and eight NVIDIA Fermi GPUs. The results show that dynamic scheduling and fine-grained parallelism achieve performance results as good as static strategies, and in most cases outperform them. The multi-level parallelism on multiple CPUs and GPUs enabled by XKaapi led to a highly efficient Cholesky factorization. The authors report that this is the best performance in double precision measured on a heterogeneous architecture with up to eight GPUs. ath that time future works was planned to include new experimental evaluations on other linear algebra problems, such as LU and QR factorizations with accelerators such as the future Intel Xeon Phi or the Kepler GT110 GPU.

The article [34] describes a high-level programming framework to efficiently execute applications specified as synchronous dataflow graphs (SDF) on heterogeneous systems using OpenCL. The framework automatically embeds actors into OpenCL kernels and instantiates data channels to improve memory access latencies and end-to-end performance. The framework exploits multilevel parallelism offered by heterogeneous systems by using pipeline and task parallelism to distribute the application to different compute devices and data-parallelism to process independent actor firings or output tokens concurrently. The proposed framework enables application designers to efficiently exploit the parallelism of heterogeneous systems without writing low-level architecture dependent code. The viability of the approach is demonstrated by running synthetic and real-world applications achieving speed-ups of up to 41x compared to execution on a single core. In the future, the goal is for the proposed design flow will be applied to other OpenCL-capable platforms.

The article [35] introduces DIF-GPU, an automated, dataflow-based design framework for application mapping and software synthesis on heterogeneous CPU-GPU platforms. DIF-GPU is designed to overcome the complex design issues of developing optimized implementations for CPU-GPU platforms, including task scheduling, interprocessor communication, memory management, and different forms of parallelism. DIF-GPU is based on novel extensions to the dataflow interchange format (DIF) package, which deeply incorporates efficient vectorization and scheduling techniques for synchronous dataflow specifications and provides software synthesis capabilities. The article shows that DIF-GPU outperforms conventional CPU-GPU mappings and enhances application performance through optimized management of interprocessor communication for given scheduling and vectorization configurations. Additionally, DIF-GPU can explore complex design spaces in the mapping of applications onto CPU-GPU platforms.

This article [36] describes a mapping of a high-level data-flow programming model, Concurrent Collections (CnC), onto heterogeneous platforms to achieve high performance and low energy consumption while maintaining the ease of use of data-flow programming. The authors designed a software flow to convert CnC programs to the Habanero-C language, extended the Habanero-C runtime system to support work-stealing across heterogeneous computing devices, and introduced task affinity for these components. They demonstrated the effectiveness of their proposed approach by mapping a pipeline of medical image-processing algorithms onto a prototype heterogeneous platform, showing up to 17.72× speedup and an estimated usage of 0.52× of the power used by CPUs alone when using accelerators and CPUs. The authors also discussed future research opportunities, including the use of tag functions for static analysis, the use of ranges for data-parallel computations, and the exploration of affinity value assignment at runtime.

TensorFlow [37], a machine learning system that operates at a large scale and in heterogeneous environments. It uses dataflow graphs to represent computation, shared state, and operations that mutate that state, which allows the nodes of the dataflow graph to be mapped across many machines in a cluster, and within a machine across multiple computational devices, including CPU, GPU, and ASICs called Tensor Processing Units (TPUs). TensorFlow offers a set of uniform abstractions that allow users to harness large-scale heterogeneous systems for production tasks and experimenting with new approaches. TensorFlow has become widely used for machine learning research as the TensorFlow programming model facilitates experimentation and it as been demonstrated that the resulting implementations are performant and scalable. TensorFlow is always in progress and as the goal to bridge the gap in automatic optimization, system level development, and providing a system that transparently and efficiently uses available distributed resources in the context of machine learning.

2.4 Design Space Exploration

Design space exploration is an essential aspect of the dataflow methodology. Simply having a high-performing low-level implementation generation tool is not sufficient to obtain efficient application software. This is because many parameters can be tuned independently of the implementation, such as actor partitioning and mapping, buffer dimension, action scheduling, actor fusion or composition, and more. Over the years, researchers have developed ways to automatically evaluate and generate configuration parameters. This section, presents some of the existing methodologies.

There exist a plethora of design space exploration tools such as CAL Design Suite [38, 39], COMPA [40], Daedalus [41, 42, 43], MAPS [44, 45, 46, 47], Mescal [48, 49] , Metropolis [50], PeaCE [51], Preesm [52, 53], Ptolemy [54, 55] , SDF^3 [56], Sesame [57], Space Codesign [58, 59], SPADE [60, 61], SynDEx [62], SystemCoDesigner [63, 64, 65]. However, in the best of the writer's knowledge such tools in the context of design space exploration for dataflow mythology targeting heterogeneous CPU/GPU co-processing platform is not really developed. The work that approach this field the most is presented in [66] that discusses the use of heterogeneous system architectures and the OpenCL paradigm for optimizing the performance per Watt in embedded and mobile markets. The paper proposes a runtime controller integrated into the Linux Operating System for optimizing the power efficiency of OpenCL applications. The controller autonomously adapts by acting on the mapping and the dynamic voltage and frequency scaling (DVFS) of processing units to optimize the performance/power consumption trade-off. The paper presents experimental results that demonstrate the efficiency of the controller to quickly converge to the optimal solution with less than 10% of error. The conclusion suggests future work in adopting further actuation knobs for resource usage and improving the policy and controller to support the concurrent execution of several applications. Overall, the paper proposes a promising solution for optimizing power efficiency in OpenCL applications.

2.5 Conclusion

In this chapter, the state-of-the-art in heterogeneous systems has been presented, encompassing both hardware and software perspectives.

First and foremost, an overview of the three main categories of hardware targeted by heterogeneous platforms is provided. These include embedded systems, general computing platforms, and data centers. This presentation aims to demonstrate the widespread nature of this heterogeneous trend and underscores the importance of having efficient and user-friendly methods for programming these systems.

Next, a more detailed examination of heterogeneous GPU architecture systems was conducted, as they are the focus of this thesis. A description of the various architecture families developed by NVIDIA was provided, as their products are utilized to present the results of this research.

Subsequently, the existing literature on software systems capable of programming CPU/GPU co-processing platforms was presented, with a focus on two main areas. First, a list of popular general development frameworks was provided, followed by a discussion of works that specifically employ the dataflow methodology.

Lastly, the current state-of-the-art in design space exploration for dataflow methodology was presented.

3 Dataflow Programming

3.1 Introduction

The increasing demand for high computational power in processing platforms is driven by the growing needs of modern application programs. The limitations of Moore's Law in creating smaller circuit components and the challenges posed by rising logic gate frequencies have led to the growing adoption of heterogeneous processing platforms. In order to effectively leverage the available computational power of these platforms, advanced levels of domain-specific hardware specialization need to be explored as opposed to simply scaling up existing processing elements. This is why heterogeneous system are increasing in popularity from embedded system, personal computing to data center as demonstrated in Section 2.2.

Models of computation (MoC) are a crucial aspect of heterogeneous system-level design. MoCs are the semantics of the interactions between modules and are used to specify the principles of a design. They are independent of the implementation technology and include classes such as Synchronous Languages, Discrete Event, Finite State Machine, Imperative, and Dataflow.

The Dataflow MoC is depicted as a directed graph, with nodes, known as actors, representing computational units and edges symbolizing channels of communication where tokens are transmitted. Dataflow models are predominantly used to characterize data-driven systems, such as signal-processing applications. Employing the Dataflow MoC in specific application domains often results in more accurate behavioral descriptions, compared to using imperative MoCs, as the Dataflow MoC aligns closer to the original intention of the algorithms. Dataflow programming has been demonstrated to be an effective method for managing large and parallel applications, addressing portability concerns across different platforms, and exploring parallelism opportunities. In fact, dataflow languages are designed to expose the parallelism inherent in the process of executing tasks on data, which enables rapid evaluation of various settings such as mapping software kernels to hardware processing elements without

incurring the costly redesigns required in traditional imperative software programming. These redesigns often require manual rewriting and can consume significant amounts of developer time. Dataflow-based computing models have been effectively utilized in a plethora of industries, including but not limited to digital codec implementation, high-frequency financial applications, and genomic analysis. Due to their expressiveness, mathematical rigor in the models, and independence from a specific architecture.

RVC-CAL is a promising solution for heterogeneous system-level design. It is a dataflow programming language that leverages the Dataflow MoC and has the specificity of expressing applications as network processes. RVC-CAL provides essential properties for heterogeneous platform design, including parallel scalability, modularity, finite state machine scheduling, portability, and adaptability. The MoC used in the dataflow networks expressed using the RVC-CAL language is inspired from the Dataflow Process Network (DPN) model. This language has the potential to change the way we approach programming parallel platforms, making it more efficient and less tedious.

This chapter provides the background necessary to fully understand the Dataflow MoC and the unique features of the RVC-CAL programming language. Additionally, the tools used to generate implementations and optimize programs from the Dataflow representation used in this work will be presented.

3.2 Dataflow Model of Computation

This Section describes the necessary foundation for explaining the Model of Computation (MoC) on which the CAL dataflow programming language is based on. Indeed, CAL combines Kahn Process Network and Dataflow Process Network MoC and extends it with the Actor Transition System.

3.2.1 Kahn Process Networks

The Kahn Process Network (KPN) [67] is a model where network of *processes* communicate exclusively through unidirectional and unbounded First-In-First-Out (FIFO) buffers. Each buffer holds a potentially infinite sequence of tokens. In the notation formally defined in [68] each token sequence is represented as $X = [x_1, x_2, x_3, \ldots]$, where x_i is a token picked from a defined set. Tokens are atomic data elements that are produced and consumed only once. Writing to the FIFO buffers is immediate and *non-blocking*. On the other hand, reading from the buffers is *blocking*, meaning that if a process tries to read a token from an empty buffer, it will wait (stalls) until there are sufficient tokens to fulfill the request. Hence, it is not possible to check if a buffer has any tokens.

3.2.1.1 Kahn Process

Let S^p represent the set of p-tuples of sequences, described as $X = X_1, X_2, \ldots, X_p \in S^p$. A Kahn process can then be depicted as a mapping from a set of input sequences to a set of output sequences, given by:

$$F : S^p \to S^q \tag{3.1}$$

Compared to other MoC that operates on state semantics, KPN process F functions on an *event semantic*. Furthermore, the only constraint is that F must be a continuous mapping function.

3.2.1.2 Monotonicity and Continuity

The sequence X is considered to come before the sequence Y (denoted as $X \sqsubseteq Y$) in a prefix ordering of sequences when X is a prefix of, or equal to, Y. For instance, if $X = [x_1, x_2]$ and $Y = [x_1, x_2, x_3]$, then $X \sqsubseteq Y$ and X is often referred to as approximating Y as it holds partial information about Y. The empty sequence, symbolized with $\perp$, is considered a prefix of any other sequence.

A sequence of increasing chains $\chi = X_0, X_1, \ldots$ is defined as $X_1 \sqsubseteq X_2 \sqsubseteq \ldots$. This increasing chain of sequences has one or more upper bounds Y such that $X_i \sqsubseteq Y$ for all $X_i \in \chi$. The Least Upper Bound (LUB) of the chain, denoted $\sqcup\chi$, is an upper bound that satisfies $\sqcup\chi \sqsubseteq Y$ for any other upper bound Y. Remark that the LUB may be an infinite sequence.

The functional process F, as defined in Equation (3.1), maps an increasing chain of sets of sequences χ to another set of sequences that may or may not form an increasing chain. The least upper bound (LUB) of χ, denoted as $\sqcup\chi$, is then mapped by F. A process F is referred to as Scott-continuous [69] if, for any increasing chain of sequences χ, the LUB of the chain $\sqcup F(\chi)$ exists and satisfies:

$$F(\sqcup\chi) = \sqcup F(\chi) \tag{3.2}$$

Networks composed of Scott-continuous processes have a property known as monotonicity, which can be considered as a type of causality that doesn't involve time. This mean that *future input* as only inference on *future output*. A process F is considered monotonic if:

$$X \sqsubseteq Y \implies F(X) \sqsubseteq F(Y) \tag{3.3}$$

where X and Y are sequences.

A continuous process is guaranteed to be monotonic, but a monotonic process is not necessarily continuous. The property of monotonicity allows for incremental computation [70], meaning that given partial information about the input sequences, it is possible to compute a portion of the output sequences. A monotonic process does not require its inputs to be complete before producing outputs and will not wait indefinitely for inputs. Networks composed of monotonic processes are known to be *determinate*.

3.2.2 Dataflow Process Network

Dataflow Process Networks (DPN) [68] are a specific type of KPN, where the processing units are referred to as *actors*. Like KPNs, DPNs only allow for communication between actors through unidirectional and potentially unbounded buffers that can transfer infinite sequences of tokens. In DPNs, *writes* to buffers are non-blocking, but *reads* from buffers are non-blocking as well, meaning that an actor can test the presence of input tokens before attempting to read them. If there are insufficient input tokens, the read operation returns immediately, and the actor is not forced to wait or be suspended. This characteristic of DPNs may lead to *non-determinism*, even though the actors themselves are deterministic.

3.2.2.1 Actor with Firings

DPN networks are a subclass of KPN where processes are represented by repeated **firings** of actors [71]. An actor firing refers to an indivisible unit of computation. These firings can be modeled as functions and their activation is guided by certain firing rules. The sequence of firings constitutes a continuous Kahn process, which is mathematically defined as the lest-fixed-point of a functional mapping. This formally establishes DPN as a special case of KPN [72].

An actor is described as a tuple (f, R), with m inputs and n outputs, where:

- $f : S^m \to S^n$ is known as the *firing function*.
- $R \subseteq S^m$ represents a set of finite sequences, referred to as the *firing rules*.
- The output of $f(r_i)$ must be finite for every $r_i \in R$.
- No two unique $r_i, r_j \in R$ can be joined, meaning they don't have a least upper bound.

The Kahn process F outlined in Equation (3.1) for the actor f, R can be interpreted as the least-fixed-point of the functional $\phi : (S^m \to S^n) \to (S^n \to S^m)$ defined as follows:

$$(\phi(F))(s) = \begin{cases} f(r) \oplus F(s') & \text{if there exists } r \in R \text{ and } s' \text{ such that } s = r \oplus s' \text{ and } s \sqsubseteq s' \\ \bot & \text{otherwise} \end{cases} \tag{3.4}$$

where $\oplus$ is the concatenation operator and $(S^m \to S^n)$ denotes the set of mappings from S^m to S^n. It can be demonstrated that ϕ is both monotonic and continuous. The firing function f doesn't have to be continuous, nor does it have to be monotonic, but it must be a finite function for each of the specified firing rules [72].

3.2.3 Actor Transition System and Composition

In Actor Transition Systems (ATS) [73], actors are described using Labeled Transition Systems (LTS). ATS extends the concept of firings in actors by incorporating the ideas of **atomic step**, **internal state**, and **priority**. In this framework, a step mark the transitions between one state to the next. Actors manage and modify their internal variables, which are not token sequences but individual internal values that cannot be shared among actors. The introduction of priorities gives actors the ability to detect and respond to the lack of tokens, but this can lead to increased difficulty in analysis and potentially bring unintended non-determinism into a dataflow application.

An actor can be described as a labeled transition system LTS $(\sigma_0, \tau, \succ)$ with, a non-empty state space Σ, the universe of tokens that can be exchanged between actors u, and a finite, partially-ordered sequence of n tokens over u, represented as U^n where:

- $\sigma_0 \in \Sigma$, the initial state of the actor.
- $\tau \subset \Sigma \times U^n \times U^m \times \Sigma$, the transition relation.
- $\succ \subset \tau \times \tau$, the strict partial order over τ.

A *transition* is the tuple $(\sigma, s, s', \sigma') \in \tau$, where $\sigma \in \Sigma$ represents the starting state, $s \in S^n$ the input, $\sigma' \in \Sigma$ the ending state, and $s' \in U^m$ the output of the transition. The relation $\succ$ on τ, also called **priority** defines a partial order relation which is non-reflexive, anti-symmetric, and transitive. The transition (σ, s, s', σ') can also be written as $\sigma \xrightarrow{s \to s'} \sigma'$. In an ATS, same as for every LTS, each transition can be labeled and called *action* λ in such a way that:

$$\lambda : \sigma \xrightarrow{s \to s'} \sigma' \tag{3.5}$$

In essence, the process of transitioning from one state to another is referred to as taking a step. This step can be identified by labeling it as an action, and its execution defined as a firing. During firing, tokens may be consumed and produced, and any changes in the internal variables may occur.

3.2.3.1 Enabled Transition and Step of an Actor

The priority relation ensures that a transition can only occur if there are no other transitions that can be made. This concept outlines the criteria for a valid step by an actor, which involves satisfying two conditions:

- Availability of the necessary input tokens.

- No transition with higher priority is available.

An n-to-m actor $(\sigma_0, \tau, >)$ has a transition $\sigma \xrightarrow{s \to s'} \sigma'$ considered **enabled** when the following conditions are met:

$$\begin{cases} v \sqsubseteq s \\ \nexists \sigma \xrightarrow{r \to r'} \sigma'' \in \tau \ : \ r \sqsubseteq v \wedge \sigma \xrightarrow{s \to s'} \sigma' > \sigma \xrightarrow{r \to r'} \sigma'' \end{cases} \tag{3.6}$$

By definition, given a state $\sigma \in \Sigma$ and an input tuple $v \in S^n$, a **step** from this state and input is depicted as any enabled transition $\sigma \xrightarrow{s \to s'} \sigma'$.

3.2.3.2 Actors Composition

The sets of *input ports*, P_τ^{in}, and *output ports*, P_τ^{out}, for a transition relation τ are defined as the ports involved in either consuming input or producing output:

$$\begin{cases} P_\tau^{in} = \{p \in P \mid \exists \sigma \xrightarrow{s \to s'} \sigma' \in \tau \ : \ \sigma(p) \neq \perp\} \\ P_\tau^{out} = \{p \in P \mid \exists \sigma \xrightarrow{s \to s'} \sigma' \in \tau \ : \ \sigma'(p) \neq \perp\} \end{cases} \tag{3.7}$$

In this, the set P contains names of both input and output ports. It is assumed that an input port named p and an output port with the same name have no relationship. To express intricate functionality, actors are combined into a **dataflow network**. For instance, the illustration in Figure 3.1 shows a dataflow network consisting of five actors that are interconnected by five buffers. The architecture of a network can be depicted by a partial function which maps each input port in its domain to its corresponding output port. It is worth mentioning that this assumption implies the absence of fan-in connections and permits open (unconnected) input and output ports.

3.3 CAL Actor Language

The Cal Actor Language (CAL) [74] is a domain-specific language that provides abstractions for using actors in a dataflow programming paradigm. Figure 3.1 shows a graphical representation of a CAL program and what it consists of. A dataflow program consists of a hierarchical graph called a *network*, which connects processing nodes through directed edges. Each node is an abstract processing element called an *actor*, and each edge represents a communication channel called a *FIFO buffer* (First In First Out), which connects the output and input ports of actors and allows them to exchange data packets called *tokens*. Different dataflow models of computation (MoCs) are reported in the literature, but they all have the common characteristic that actors base their computations on the consumed tokens and their internal state. Actors

are independent computing elements that can communicate only through the buffers, which enables interesting properties such as composability of network elements and the ability to partition a dataflow network into sub-network elements that can be mapped and executed on independent but interconnected processing elements.

The computation is done through the execution of special functions called *actions*, which are considered atomic kernels of execution that cannot be stopped. Each action can be associated with a *guard* statement, a boolean function that must be satisfied for the action to be executed. The CAL Actor language follows the ATS model, resulting in a class of dataflow MoC that provides high expressiveness where the consumption and production of data tokens can depend on the values of input data tokens and the internal state of the actor, which can include state variables and a finite-state machine (FSM). The FSM drives the selection and ordering of action firings, and only an action associated with the current state can be fired, resulting in a state transition.

CAL's expressiveness is well-suited for designing complex and intuitive dynamic algorithms, but it also leads to more difficult analysis and optimization problems when generating efficient implementations of the network.

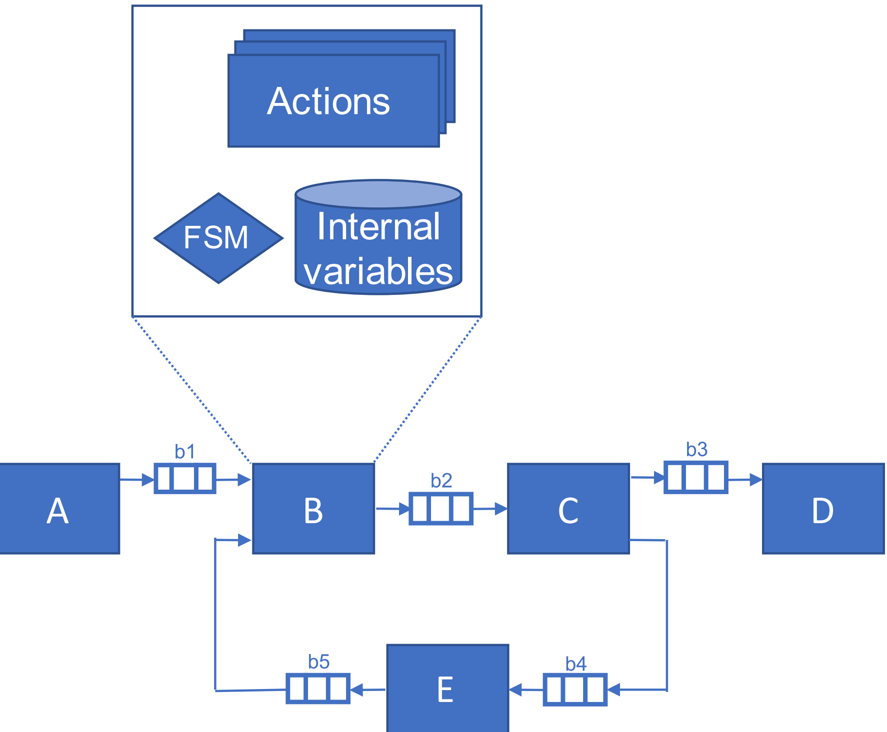


Figure 3.1: Representation of a CAL application network comprising five actors and five FIFO buffers, along with a detailed view of actor internals featuring finite state machine, actions and internal variables.

3.3.1 Formal Definition

A CAL program named *network* and define as N is the composition of actors represented as a tuple (K,A,B) where:

- K is a finite set of actor classes and is represented as $K = \{\kappa_1, \kappa_2, ..., \kappa_{n_K}\}$.
- A is a finite set of actors and is denoted as $A = \{a_1, a_2, ..., a_{n_A}\}$.
- B is a finite set of queues and is represented as $B = \{b_1, b_2, ..., b_{n_B}\}$.

An *actor class* κ in a CAL program refers to the program code template and behavior implementation of an actor (i.e. the source code). Multiple actors can be instantiated from the same class, but each actor is distinct with its own internal state that cannot be shared.

An *actor* in a CAL program is represented as a tuple $(\kappa, P^{in}, P^{out}, \Lambda, V, FSM)$ where:

- κ is the actor class.
- P^{in} is the finite set of input ports and is denoted as $P^{in} = \{p^{in}1, p^{in}2, ..., p^{in}n_I\}$.
- P^{out} is the finite set of output ports and is represented as $P^{out} = \{p^{out}1, p^{out}2, ..., p^{out}n_O\}$.
- Λ is the finite set of actions and is denoted as $\Lambda = \{\lambda_1, \lambda_2, ..., \lambda n\Lambda\}$.
- V is the finite set of internal variables and is represented as $V = \{v_1, v_2, ...v_{n_V}\}$.
- FSM is the internal finite state machine.

A *queue* in a CAL program is defined as a tuple (a_s, p_s, a_t, p_t) where:

- $a_s \in A$ is the source actor (i.e. the one that produces the tokens)
- $p_s \in P^{out}_{a_s}$ is the output port of the source actor
- $a_t \in A$ is the target actor (i.e. the one that consumes the tokens from the queue)
- $p_t \in P^{in}_{a_t}$ is the input port of the target actor

3.3.2 Execution Model

In this work the following model for the selection of the action to be fired is considered. Figure 3.2 illustrates the 9 steps involved in executing an action, which are:

- **Test States**: Test is the FSM is in the appropriate state.
- **Test Priority**: Test that no action with higher priority can be fired.
- **Test Input**: Test that there are enough tokens available in the input buffers.
- **Test Output**: Test that there is enough free space in the output buffer to write the resulting data.
- **Execute Guard**: Execute the action's guard.
- **Test guards**: Test if the guard is satisfied
- **Read Input**: Read all data from the input buffer necessary for the execution of the action
- **Execute Action**: Execute the action's code.
- **Write Output**: Write the resulting data computed during action execution to the output buffers.

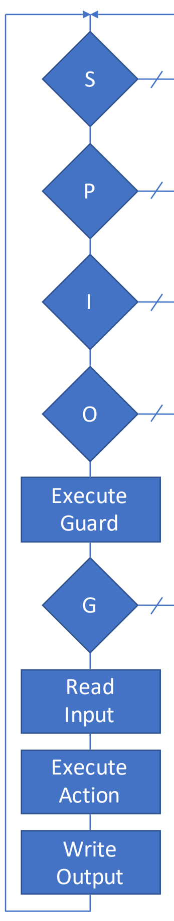


Figure 3.2: Visual representation of the actor's execution model considered in this thesis.

3.3.3 Syntax and Semantic

3.3.3.1 Lexical Tokens

Lexical tokens, also known as lexemes, are groups of indivisible characters with a collective meaning. They serve as building blocks for structuring algorithms and using the functionalities provided by a programming languages. The CAL lexical tokens can be categorized as follows:

- **Keywords**: special type of identifier designated by a programming language for specific purposes. They cannot be used as user-defined identifiers in any other context. Some reserved keywords are *while*, *if*, *else false*, *true*, *begin*, *action*, and *actor*.
- **Operators**: depict mathematical, algebraic or logical operations. Operators are written as special characters such as !, &, ^, %, =, <, >, ?, ∗, /, +, −, | and ~.
- **Delimiters**: are used to indicate the start or the end of a syntactical element in the CAL code. The following delimiters are defined: [,], (,), { and }.
- **Comments**: comments constituting of a single-line can be represented by starting with // while comment ranging multiple-lines start with /∗ and end with ∗/.

3.3.3.2 Actors

Actors are the building blocks of a dataflow program and serve as processing elements. The CAL syntax to declare an actor is shown in Listing 3.1. There are three important parts to note. First, actors can have parameters (i.e. the *name* of type *String*), which can be defined during instantiation and help to specify a specific instantiation of an actor when it is used multiple times within the same network. Second, actors have a fixed communication interface defined by a list of input and output ports, represented here by *I* as an input port and *O* as an output port, both of type *int*, separated by the ==> symbol. Finally, actors can encapsulate state in the form of internal values, variables, arrays, etc., which can be initialized during instantiation, such as in the example the *counter* variable of type *int*, which is initialized to the value zero.

Listing 3.1: Example of the actor's syntax

```
1  actor MyActor(String name) int I ==> int O :
2
3    int counter := 0;
4
5    [...]
6
7  end
```

3.3.3.3 Actions

The action is an atomic execution step that takes the form of a function. The actor in Listing 3.2 contains a single action labeled *myaction* that demonstrates how the consumption and production of tokens are specified. The input pattern, which is before the ==>, describes the number of tokens read from input ports and the name of the variable that references these tokens for the rest of the action code. In this example, a single token is read from the input I and is referred to as v. The output pattern, which is after the ==> defines the number of tokens produced and written to the output FIFO buffers each time the action is executed. In this example, a single token is written to the O port, and its value is twice the value of v.

Listing 3.2: Example of the action's syntax

```
actor MyActor(String name) int I ==> int O :

  int counter := 0;

  myaction: action I : [ v ] ==> O : [ 2 * v ] end

end
```

3.3.3.4 Guards

A guard is an additional firing condition that can be specified for each action and that is evaluated before an action is considered for firing. Listing 3.3 shows that *myaction* has a guard that states that it can only be fired if the *counter* variable is less than 10. It is important to note that the guard expressions can be based on the values read from the inputs. In fact, the FIFO buffer implementations connecting actors need to be able to provide pick functionality, meaning that input values can be tested without being consumed. As a side note, compared to the previous example where only input and output patterns were used, this action contains a body where the *counter* variable is incremented.

Listing 3.3: Example of the guard's syntax

```
actor MyActor(String name) int I ==> int O :

  int counter := 0;

  myaction: action I : [ v ] ==> O : [ 2 * v ]
  guard counter < 10
    counter := counter + 1;
  end

end
```

3.3.3.5 FSM

The Finite State Machine (FSM) is an additional construct that an actor can contain to organize and control the firing order of actions.

Listing 3.4 shows an example of an FSM that contains two states: *stateOne* and *stateTwo*, and two functions: *one* and *two*. Line 7 indicates that the FSM is initialized to the *stateOne* state. The next two lines specify that when the FSM is in the *stateOne* state, only the action *one* can fire and when it does, the FSM transitions to the *stateTwo* state. Similarly, when the FSM is in the *stateTwo* state, only the action *two* can fire, causing the FSM to transition back to the *stateOne* state.

Listing 3.4: Example of the FSM's syntax

```
actor MyActor(String name) int I ==> int O :

  one: action I : [ v ] ==> O : [ v ] end

  two: action I : [ v ] ==> O : [ v ] end

  schedule fsm stateOne:
    stateOne(one) --> stateTwo
    stateOne(two) --> stateOne
  end
end
```

3.3.3.6 Priorities

Priorities, as the name implies, allow for specifying the firing priority between actions and play a role in the action selection algorithm. If all scheduling rules permit two actions to fire at the same time, the priority will determine which one will be executed first, instead of allowing an arbitrary, implementation-specific, order to take place. Listing 3.5 shows an example with two actions: *one* and *two*. The action *one* has priority over the action *two* as indicated by the symbol > in the listing.

Listing 3.5: Example of the priority's syntax

```
actor MyActor(String name) int I ==> int O :

  one: action I : [ v ] ==> O : [ v ] end

  two: action I : [ v ] ==> O : [ v ] end

  priority
    one > two
  end
end
```

3.3.4 Complete Example of a Program

Figure 3.3 depicts the graphical representation of the top-level network of a CAL dataflow program. This network can be programmed using the XML Dataflow Format (XDF) based on the eXtensible Markup Language (XML), as demonstrated in Listing 3.6. This example will be used in subsequent chapters to illustrate the various features and optimizations discussed in this thesis. The network consists of five instances of actors (*Prod*, *CopyTokensA*, *CopyTokensB*, *PingPong*, and *Merger*). Note that *CopyTokensA* and *CopyTokensB* are two instances of the same actor (*CopyTokens*), as shown in Listing 3.8. The CAL implementation of the *Producer* actor is presented in Listing 3.7. It consists of a single action that produces one token per firing, which is incremented each time. A guard prevents the action from firing more than six times.

In Listing 3.9, the schedule statement acts as a finite-state machine (FSM), and the transition from one state to another is accomplished by the firing of an action, as discussed in Section 3.3.2, the FSM acts has an extra parameter for selecting the next action to fire, and in this case it makes the two actions to fire alternatively.

Finally, the *Merger* actor, shown in Listing 3.10, reads a token from each of its two input ports and prints them along with the firing number (*counter*). It is important to note that the output port of the *Prod* instance is connected to the input ports of two different instances (*CopyTokensA* and *PingPong*), so the corresponding FIFO implementation must be able to handle this. If sufficient hardware resources are available to execute all actors in parallel and only considering actual dependencies, Figure 3.4 depicts the execution of the program by showing when the different actions will fire.

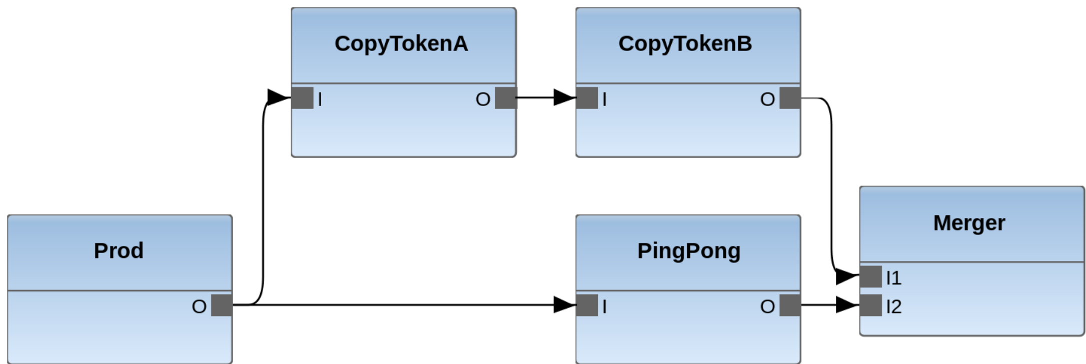


Figure 3.3: An example of a dataflow network with five actors (i.e. *Prod*, *CopyTokenA*, *CopyTokenB*, *PingPong* and *Merger*).

Listing 3.6: Description of the dataflow network using the XML Dataflow Format (XDF)

```
<?xml version="1.0" encoding="UTF-8"?>
<XDF name="Example">
    <Instance id="Prod">
        <Class name="cal.Producer"/>
    </Instance>
    <Instance id="CopyTokenA">
        <Class name="cal.CopyTokens"/>
        <Parameter name="name">
            <Expr kind="Literal" literal-kind="String" value="first"/>
        </Parameter>
    </Instance>
    <Instance id="CopyTokenB">
        <Class name="cal.CopyTokens"/>
        <Parameter name="name">
            <Expr kind="Literal" literal-kind="String" value="second"/>
        </Parameter>
    </Instance>
    <Instance id="PingPong">
        <Class name="cal.PingPong"/>
    </Instance>
    <Instance id="Merger">
        <Class name="cal.Merger"/>
    </Instance>
    <Connection dst="CopyTokenA" dst-port="I" src="Prod" src-port="O"/>
    <Connection dst="CopyTokenB" dst-port="I" src="CopyTokenA" src-port="
      O"/>
    <Connection dst="PingPong" dst-port="I" src="Prod" src-port="O"/>
    <Connection dst="Merger" dst-port="I1" src="CopyTokenB" src-port="O"/
      >
    <Connection dst="Merger" dst-port="I2" src="PingPong" src-port="O"/>
</XDF>
```

Listing 3.7: CAL implementation of the Producer actor.

```
actor Producer () ==> int O:

  uint counter := 0;

  p: action  ==> O:[counter]
  guard
    counter < 6
  do
    counter := counter + 1;
  end

end
```

Listing 3.8: CAL implementation of the CopyTokens actor.

```
actor CopyTokens (String name) int I ==> int O:

  c: action  I:[val] ==> O:[val] end

end
```

Listing 3.9: CAL implementation of the PingPong actor.

```
actor PingPong () int I ==> int O:

  pp1: action I:[val] ==> O:[val]
  do
    println("PingPong[pp1]:" + val);
  end

  pp2: action I:[val] ==> O:[-val]
  do
    println("PingPong[pp2]:" + val);
  end

  schedule fsm a_pp1:
    a_pp1(pp1) --> a_pp2;
    a_pp2(pp2) --> a_pp1;
  end

end
```

Listing 3.10: CAL implementation of the Merger actor.

```
actor Merger () int I1, int I2 ==> :

  uint counter := 0;

  m: action I1:[ v1 ], I2:[ v2 ] ==>
  do
    println("Merger("+counter+"):"+ v1 +";"+ v2);
    counter := counter + 1;
  end

end
```

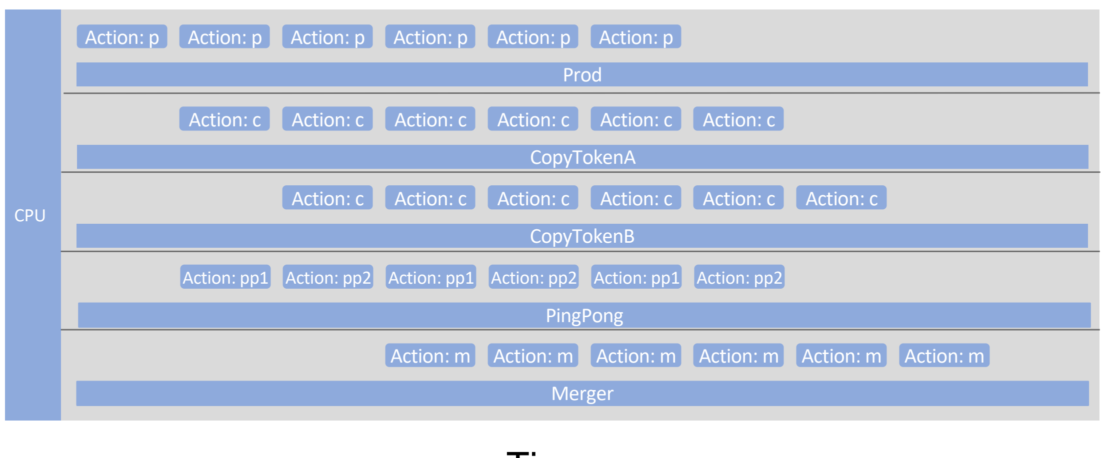


Figure 3.4: Execution of the CAL example with all actors running in separate threads on a CPU.

3.4 Open RVC-CAL Compiler

In the present work, the RVC-CAL dataflow language, a subset of the CAL language standardized by the MPEG committee (referenced in [75, 76, 77, 78]), is used to express the algorithm of the application program. This subset limits the types of data, operators, and features that can be utilized when describing a CAL actor. Within the MPEG community, RVC-CAL is utilized as a reference software language for specifying MPEG video-coding technology through a library of components (actors) that are configured and instantiated into networks to produce standard MPEG video decoders (e.g. MPEG4-SP, AVC, HEVC).

Several open-source compilers have been developed for the RVC-CAL programming language, including Caltopia [79], Tycho [80], Cal2Many [81, 82], DAL [83], or Streamblock [84]. In this thesis, ORCC [85, 86], the Open RVC-CAL Compiler, is a source-to-source compiler framework that provides the tools for designing, simulating, and generating code for different software

runtime or hardware architectures is used. It take the form of a fully featured Integrated Development Environment (IDE) built as an Eclipse IDE [87] project. Figure 3.5 shows how the tool flow is setup. The compiler processes the RVC-CAL description along with configuration files (XDF, XCF, and BXDF) to drive the implementation. After various stages of processing in the compiler pipeline (presented in the next subsection), the backend generates platform-specific source code (referred to as "Program Code" in the diagram). This code is then compiled using a platform-specific compiler to produce the final binary that will run on the hardware platform.

The advantage of using a source-to-source compiler instead of directly generating a binary is that it simplifies the process of supporting many different hardware architectures and automatically benefits from improvements made by vendor-specific or open-source compilers. Additionally, this methodology is highly efficient for working with heterogeneous systems, as the different parts can be compiled by compilers designated for each element's architecture while maintaining a single location for the application program's design.

In addition to the RVC-CAL and network description (XDF) discussed in Section 3.3, the compiler requires additional configuration files. The *BXDF* file specifies the fixed size of all FIFO buffers in the network. The *XCF* file provides the *partition* and *mapping* of the network. A partition is a group of sets of actors, where each actor in the network belongs to exactly one set and follows the same scheduling strategy. For example, a non-preemptive set is scheduled in such a way that each actor is executed sequentially and runs until no action can be taken due to conditions on their input data. The mapping is the assignment of these sets to a specific computational hardware resource, such as CPU cores. Examples of parameter files that are compatible with the example in Figure 3.3 can be seen in Listing 3.12 for the BXDF file and Listing 3.11 for the XCF file.

Listing 3.11: Example of an XCF file specifying the partitioning and mapping configuration for a dataflow program.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Configuration network="Example">
3    <Partitioning>
4      <Partition id="1" scheduling="NON_PREEMPTIVE">
5        <Instance id="Prod"/>
6        <Instance id="CopyTokenA"/>
7        <Instance id="CopyTokenB"/>
8        <Instance id="PingPong"/>
9        <Instance id="Merger"/>
10     </Partition>
11   </Partitioning>
12 </Configuration>
```

Listing 3.12: Example of a BXDF file specifying the sizes of all FIFO buffer connections in a dataflow program.

```
<?xml version="1.0" ?>
<bxdf network="Example" default-size="1024">
  <connection source="CopyTokenA" source-port="O"
              target="CopyTokenB" target-port="I"
              size="1024"/>

  <connection source="CopyTokenB" source-port="O"
              target="Merger" target-port="I1"
              size="1024"/>

  <connection source="PingPong" source-port="O"
              target="Merger" target-port="I2"
              size="1024"/>

  <connection source="Prod" source-port="O"
              target="CopyTokenA" target-port="I"
              size="1024"/>

  <connection source="Prod" source-port="O"
              target="PingPong" target-port="I"
              size="1024"/>
</bxdf>
```

The ORCC's compiler pipeline is composed of the following parts:

- **Front-end**: this stage is responsible for parsing the RVC-CAL code and creating the Abstract Syntax Tree (AST). This stage is implemented using Xtext [88], a textual modeling framework designed for developing programming languages and Domain Specific Languages (DSL), to automatically create a parser, linker, and editor from the grammar description. The AST is then converted into an Intermediate Representation (IR), which enables further code manipulation and optimization. This stage also performs semantic validation, type inference, and expression evaluation.

- **Core**: this part contains the IR definitions and provides the machinery for the development of optimization stages. It uses the Eclipse Modeling Framework (EMF) [89] to automatically generate functions for manipulating the data structures and to provide runtime support for the model. The Core also provides automatic serialization of the IR, enabling incremental optimization.

- **Interpreter**: this stage provides the necessary code so that the IR can be directly interpreted and emulated. The simulation is type-accurate and grants the possibility to verify the correct functionality and behavior of the RVC-CAL program before any final implementation is performed.

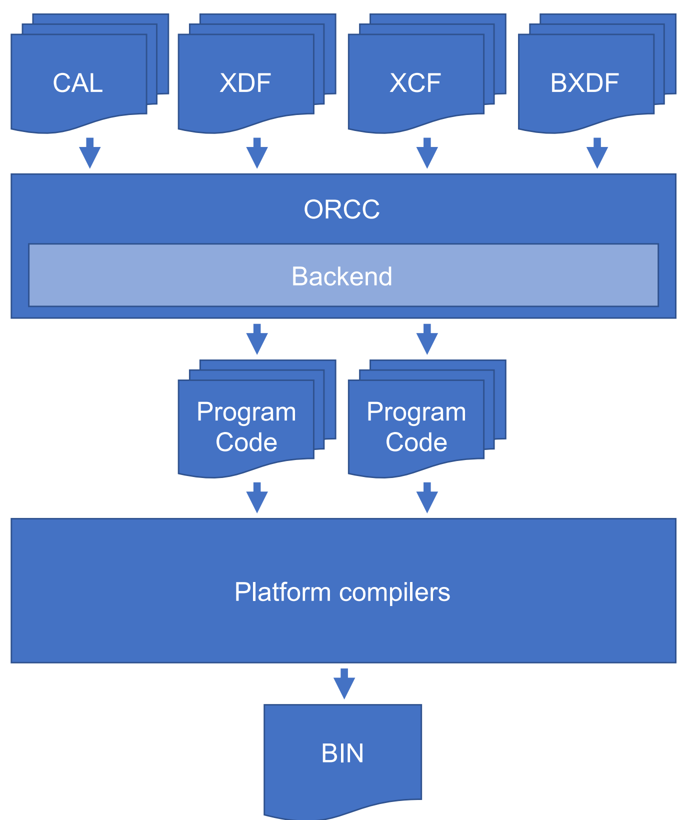


Figure 3.5: Overview of the code generation process within the ORCC compiler framework.

- **Back-end**: this is the final stage of the compiler infrastructure and is responsible for converting the generic IR into the target environment. It starts with IR optimizations that are specific to the target platform, and then generates the final output of the compiler. ORCC has multiple back-ends, including a new one developed in this thesis. Each back-end is used for different purposes, although typical ORCC back-ends generate code in general-purpose programming languages such as C/C++, Java, Python, etc., targeting specific hardware platforms. The code generation is done using Xtend [88], a flexible and user-friendly template-based code generation framework. The generated code then needs to be compiled or interpreted by a platform-specific compiler to produce an executable program. This thesis focuses mainly on this stage to integrate GPUs as a target hardware platform.

3.5 Design Space Exploration Framework

Dataflow programs can be freely partitioned and mapped, resulting in correct executions without the need for software rewriting. However, this creates challenges in terms of finding the most efficient configurations for partitioning, mapping, and scheduling. The number of possible design points is so large that it would be too time-consuming or even impossible for

a developer to find the best configuration through trial and error. This requires automatic and systematic methods for identifying and evaluating good design point configurations. TURNUS [90, 91] is a design space exploration framework that has been developed for this purpose. It is based on a high-level abstract model of computation, which includes the dataflow network structure and the actor execution model, and is enhanced with profiling measures of each atomic execution obtained on a specific heterogeneous processing platform. The execution model analysis can then explore the configuration design space and find efficient configurations without having to actually execute each one on the hardware platform.

Figure 3.6 illustrates the flow of the design space exploration tool when working with TURNUS to optimize dataflow programs in the ORCC framework. The RVC-CAL representation of the application program, along with the different configuration files (network, partition, buffer sizes), is first fed into the compiler. An optimization loop is initiated, which continues until the user is satisfied with the performance achieved. The optimization objectives can vary, such as critical path reduction, minimizing overall execution time, maximizing resource utilization, and so on. In the particular context of this thesis, the loop runs for a fixed amount of time predetermined by the user at the start of the process. The loop begins with the compiler generating a platform-specific source code implementation for the targeted architecture, which includes performance evaluation code. This code is then compiled or synthesized into an executable using platform-specific compilers. Once executed, the performance metric is extracted from the platform and the Execution Trace Graph (ETG) is labeled. The performance estimation of the application can then be evaluated using different configurations without the need to recompile and execute each time. The system analyzes the ETG to perform various evaluations, such as the critical path evaluation or buffer dimensions, and proposes new configurations to be tested on the platform.

In this section, the four elements that make up the TURNUS design space exploration methodology are introduced. Firstly, the ETG is defined, then its combined use with the performance metric extracted from the concrete implementation is discussed. Next, the post-processing method for the performance estimation of the application is described. Finally, the last part demonstrates how all these elements are combined to perform the design space exploration.

3.5.1 Execution Trace Graph

The ETG (Execution Time Graph) is a graph-structured representation of the execution of a dataflow program. Each node in the ETG represents a single action firing, and each directed arc represents an execution constraint between two different action firings. These constraints can be caused by internal variables, finite state machines, guards, ports, or tokens. Table 3.2 illustrates the different types of dependencies and how they are represented in the ETG.

To provide an example, Figure 3.7 shows a graphical representation of the ETG obtained when executing the RVC-CAL program described in Figure 3.3 with the guard on the counter of the Prod actor set to 2. In this example, the firing set S contains ten action firings $S = S_0, S_2, \ldots, S_9$,

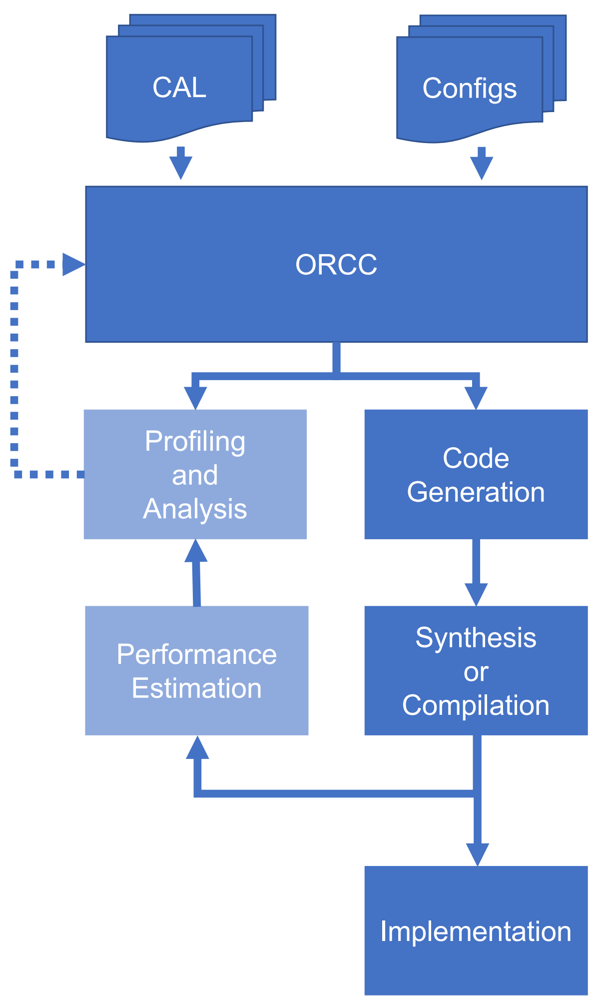


Figure 3.6: Overview of the parameter optimization process within the TURNUS design space exploration framework.

which are summarized in Table 3.1. The table includes the actor, actor-class, and action responsible for each firing. The dependency set D contains twenty-one dependencies $D = e_0, e_2, \ldots, e_{20}$, which are summarized in Table 3.3.

3.5.2 Performance Metrics

Once the ETG is generated, the next step is to create a *Timed execution trace graph* (TETG) where each firing and each dependency is labelled with a corresponding time value called weight. These weights are measured on the target hardware platform through the software instrumentation of the code generated by the ORCC compiler. We denote 3 types of dependencies:

- **Execution**: time of the execution of the action firing. It consist only of the computation

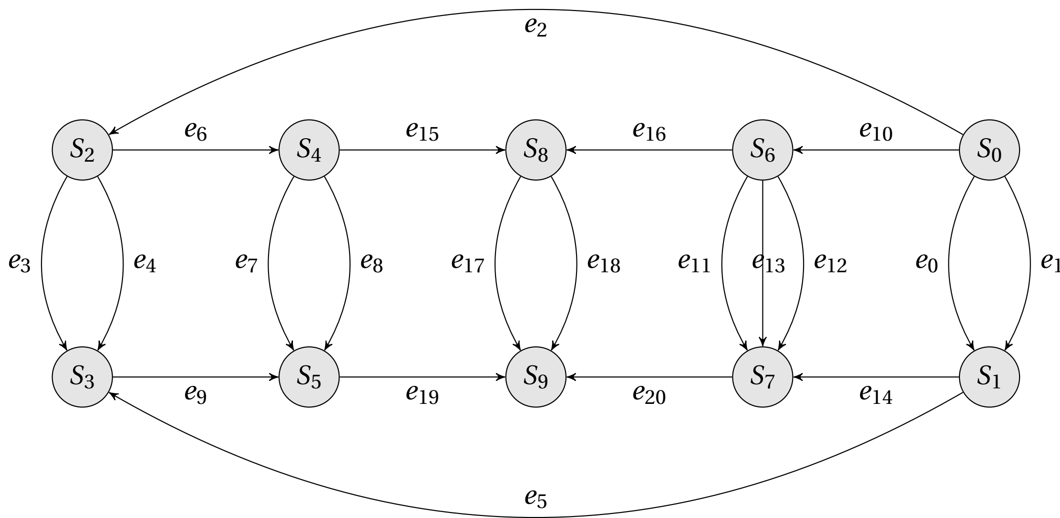


Figure 3.7: Execution trace graph obtained after the execution of the RVC-CAL program described in Figure 3.3 if the guard of the counter of the Prod actor was set to 2. The firing set S is summarized in Table 3.1, and the dependencies set D is summarized in Table 3.3.

<table>
<tr><th>Firing</th><th>Actor</th><th>Actor-class</th><th>Action</th></tr>
<tr><td>S0</td><td rowspan="2">Prod</td><td rowspan="2">Producer</td><td rowspan="2">p</td></tr>
<tr><td>S1</td></tr>
<tr><td>S2</td><td rowspan="2">CopyTokenA</td><td rowspan="4">CopyTokens</td><td rowspan="4">c</td></tr>
<tr><td>S3</td></tr>
<tr><td>S4</td><td rowspan="2">CopyTokenB</td></tr>
<tr><td>S5</td></tr>
<tr><td>S6</td><td rowspan="2">PingPong</td><td rowspan="2">PingPong</td><td>pp1</td></tr>
<tr><td>S7</td><td>pp2</td></tr>
<tr><td>S8</td><td rowspan="2">Merger</td><td rowspan="2">Merger</td><td rowspan="2">m</td></tr>
<tr><td>S9</td></tr>
</table>

Table 3.1: Firing of the RVC-CAL program described in Figure 3.3 if the counter of the Prod actor was set to 2.

of the body of the action and of the computation of the potential output pattern.

- **Communication**: time of transmission of the data tokens. It consist of the reading of the token from the input buffers and the writing of results token in the output buffers.
- **Scheduling**: time of the action selection process inside an actor. To be noted that it doesn't include the context switching between different actors of a same partition.

Listings 3.13, 3.15, and 3.14 are examples of the execution, communication, and scheduling weights, respectively, extracted from the execution of the instrumented code generated by

	Name	Direction	Parameters	Additional attributes
D_v	internal variable	read/read write/write read/write write/read	variable id	initial value final value
D_f	finite state machine			
D_g	guard	enable disable	guard id appearance order	
D_p	port	read/read write/write	port id	
D_t	tokens		output port id number of tokens	token values

Table 3.2: Dependencies kinds, directions, parameters and additional attributes.

the ORCC compiler of the RVC-CAL example shown in Figure 3.3. In Listings 3.13, an entry is presented for each action of each actor, including the name of the action, the average, minimum, maximum, and variance of the number of clock cycles needed for its execution. In Listings 3.15, each FIFO is labeled with the name of the source and destination actors and ports, and for each one, the latency in clock cycles to read and write from and to the FIFO is displayed. Finally, Listings 3.14 show the scheduling weights, including the average, minimum, maximum, and variance of the number of clock cycles needed for the scheduling of each action. Each entry represents the time spent between the end of the execution of the "source" action (the last action executed) and the beginning of the execution of the "target" action (the next action to be executed). Additionally, and for all type of weights, each actor entry shows the clock frequency of the hardware platform executing that actor. This information is important for heterogeneous platforms, as explained in Section 5.2.1.

3.5.3 Performance Estimation

In order to make decisions about the performance of specific design points, an automated way to evaluate the performance of an application program is necessary. To meet this requirement, TURNUS has an estimation performance tool named the ETG post-processor. It uses two inputs: the platform-independent model of the program execution provided by the TETG and the model of the target architecture. The post-processor is based on a Discrete Event System Specification formalism, as described in [92].

Discrete Event System Specification (DEVS) is a formalism for modeling systems with discrete event dynamic behavior. In DEVS, a system is represented as a set of atomic models that are connected to each other and described by their state transition, output, and time advance functions. Communication between the atomic models occurs through signals that are received or sent via port values, which define the types of objects accepted or produced as input or output, respectively.

(S_i, S_j)	Source	Target	Kind	Direction	Parameter	Attribute
e_0	S_0	S_1	Variable	Write/Write	variable=counter	before=1 after=2
e_1	S_0	S_1	Port	Write/Write	port=O	
e_2	S_0	S_2	Tokens	-	count=1 source-port=O target-port=I	value=1
e_3	S_2	S_3	Port	Read/Read	port=I	
e_4	S_2	S_3	Port	Write/Write	port=O	
e_5	S_1	S_3	Tokens	-	count=1 source-port=O target-port=I	value=2
e_6	S_2	S_4	Tokens	-	count=1 source-port=O target-port=I	value=1
e_7	S_4	S_5	Port	Read/Read	port=I	
e_8	S_4	S_5	Port	Write/Write	port=O	
e_9	S_3	S_5	Tokens	-	count=1 source-port=O target-port=I	value=2
e_{10}	S_0	S_6	Tokens	-	count=1 source-port=O target-port=I	value=1
e_{11}	S_6	S_7	FSM	-		
e_{12}	S_6	S_7	Port	Read/Read	port=I	
e_{13}	S_6	S_7	Port	Write/Write	port=O	
e_{14}	S_1	S_7	Tokens	-	count=1 source-port=O target-port=I	value=2
e_{15}	S_4	S_8	Tokens	-	count=1 source-port=O target-port=I1	value=1
e_{16}	S_6	S_8	Tokens	-	count=1 source-port=O target-port=I2	value=1
e_{17}	S_8	S_9	Port	Read/Read	port=I1	
e_{18}	S_8	S_9	Port	Read/Read	port=I2	
e_{19}	S_5	S_9	Tokens	-	count=1 source-port=O target-port=I1	value=2
e_{20}	S_7	S_9	Tokens	-	count=1 source-port=O target-port=I2	value=-2

Table 3.3: Dependencies set S of the execution trace graph depicted in Figure 3.7.

There are four building blocks that can be used to model a dataflow program in DEVS: an actor, a buffer, an actor partition, and a buffer partition. An atomic actor models a dataflow actor that processes all of its firings contained in the ETG. The DEVS time advance function corresponds to the action weights assigned to each firing, obtained through profiling, and defines the next update time (state transition) of an actor model. Executing each firing requires transitioning through several states of an actor.

Listing 3.13: Example of an EXDF file specifying the execution weights extracted from a dataflow program.

```
<network name="Example">
    <actor id="CopyTokenA" frequency="1800000">
        <action id="c" clockcycles="2.000000" clockcycles-min="2.000000"
          clockcycles-max="2.000000" clockcycles-var="0.000000"/>
    </actor>
    <actor id="CopyTokenB" frequency="1800000">
        <action id="c" clockcycles="2.000000" clockcycles-min="2.000000"
          clockcycles-max="2.000000" clockcycles-var="0.000000"/>
    </actor>
    <actor id="Merger" frequency="1800000">
        <action id="m" clockcycles="2.000000" clockcycles-min="2.000000"
          clockcycles-max="2.000000" clockcycles-var="0.000000"/>
    </actor>
    <actor id="PingPong" frequency="1800000">
        <action id="pp1" clockcycles="184200.000000" clockcycles-min="
          184200.000000" clockcycles-max="184200.000000" clockcycles-var=
          "0.000000"/>
        <action id="pp2" clockcycles="86695.000000" clockcycles-min="
          86695.000000" clockcycles-max="86695.000000" clockcycles-var="
          0.000000"/>
    </actor>
    <actor id="Prod" frequency="1800000">
        <action id="p" clockcycles="49413.000000" clockcycles-min="
          219.000000" clockcycles-max="98607.000000" clockcycles-var="
          4840099272.000000"/>
    </actor>
</network>
```

3.5.4 Design Space Exploration

The design space exploration can now be done using the elements described in the previous three sections. Using TURNUS' performance estimation tool, one can perform critical path detection, hotspot analysis, buffer size dimensioning, and partitioning to help the developer achieve their performance target. In the context of design space exploration, this thesis focuses on partitioning, specifically for adding GPUs as a target platform in a heterogeneous system. The goal is to identify which actors in a workflow would benefit from being executed on GPU

hardware. That is why Section 5.3 presents how the Tabu search meta-algorithm can be used for this purpose.

Listing 3.14: Example of an SXDF file specifying the scheduling weights extracted from a dataflow program.

```
1  <network name="Example">
2      <actor id="CopyTokenA" frequency="1800000">
3          <scheduling source="" target="c" clockcycles="735796.000000"
             clockcycles-min="735796.000000" clockcycles-max="735796.000000"
              clockcycles-var="0.000000"/>
4          <scheduling source="c" target="c" clockcycles="69141.000000"
             clockcycles-min="69141.000000" clockcycles-max="69141.000000"
             clockcycles-var="0.000000"/>
5      </actor>
6      <actor id="CopyTokenB" frequency="1800000">
7          <scheduling source="" target="c" clockcycles="1208113.000000"
             clockcycles-min="1208113.000000" clockcycles-max="
             1208113.000000" clockcycles-var="0.000000"/>
8          <scheduling source="c" target="c" clockcycles="14045.000000"
             clockcycles-min="14045.000000" clockcycles-max="14045.000000"
             clockcycles-var="0.000000"/>
9      </actor>
10     <actor id="Merger" frequency="1800000">
11         <scheduling source="" target="m" clockcycles="1621003.000000"
             clockcycles-min="1621003.000000" clockcycles-max="
             1621003.000000" clockcycles-var="0.000000"/>
12         <scheduling source="m" target="m" clockcycles="91602.000000"
             clockcycles-min="91602.000000" clockcycles-max="91602.000000"
             clockcycles-var="0.000000"/>
13     </actor>
14     <actor id="PingPong" frequency="1800000">
15         <scheduling source="" target="pp1" clockcycles="717588.000000"
             clockcycles-min="717588.000000" clockcycles-max="717588.000000"
              clockcycles-var="0.000000"/>
16         <scheduling source="pp1" target="pp2" clockcycles="14744.000000"
             clockcycles-min="14744.000000" clockcycles-max="14744.000000"
             clockcycles-var="0.000000"/>
17     </actor>
18     <actor id="Prod" frequency="1800000">
19         <scheduling source="" target="p" clockcycles="535952.000000"
             clockcycles-min="535952.000000" clockcycles-max="535952.000000"
              clockcycles-var="0.000000"/>
20         <scheduling source="p" target="p" clockcycles="250024.000000"
             clockcycles-min="250024.000000" clockcycles-max="250024.000000"
              clockcycles-var="0.000000"/>
21     </actor>
22 </network>
```

Listing 3.15: Example of an CXDF file specifying the communication weights extracted from a dataflow program.

```
<communication network="Example">
    <buffer source-actor="Prod" source-port="O" target-actor="CopyTokenA"
       target-port="I">
        <memory level="RAM">
            <read type="hit" percentage="1.000000" latency="128622.000000
             " frequency="1800000"/>
            <write type="hit" percentage="1.000000" latency="42703.500000
             " frequency="1800000"/>
        </memory>
    </buffer>
    <buffer source-actor="CopyTokenA" source-port="O" target-actor="
      CopyTokenB" target-port="I">
        <memory level="RAM">
            <read type="hit" percentage="1.000000" latency="73841.500000"
              frequency="1800000"/>
            <write type="hit" percentage="1.000000" latency="58644.000000
             " frequency="1800000"/>
        </memory>
    </buffer>
    <buffer source-actor="CopyTokenB" source-port="O" target-actor="
      Merger" target-port="I1">
        <memory level="RAM">
            <read type="hit" percentage="1.000000" latency="44639.500000"
              frequency="1800000"/>
            <write type="hit" percentage="1.000000" latency="39526.500000
             " frequency="1800000"/>
        </memory>
    </buffer>
    <buffer source-actor="PingPong" source-port="O" target-actor="Merger"
       target-port="I2">
        <memory level="RAM">
            <read type="hit" percentage="1.000000" latency="44184.000000"
              frequency="1800000"/>
            <write type="hit" percentage="1.000000" latency="52786.500000
             " frequency="1800000"/>
        </memory>
    </buffer>
    <buffer source-actor="Prod" source-port="O" target-actor="PingPong"
      target-port="I">
        <memory level="RAM">
            <read type="hit" percentage="1.000000" latency="67035.500000"
              frequency="1800000"/>
            <write type="hit" percentage="1.000000" latency="42703.500000
             " frequency="1800000"/>
        </memory>
    </buffer>
</communication>
```

3.6 Conclusion

This section has provided an overview of dataflow parallel programming concepts. The dataflow computational model was defined and its background explored. CAL, a behavioral language, was subsequently introduced and exemplified with various code snippets. Key concepts such as actors, actions, guards, priorities, and finite state machines were demonstrated.

This chapter has introduced two powerful frameworks that were utilized in this work to achieve the desired results. The first framework, known as the OpenRVC-CAL Compiler, is a compiler framework that was used and extended to synthesize a low-level implementation targeting the hardware architecture from the CAL dataflow programming language representation. The second framework, called TURNUS, is a design space exploration framework that played a important role in this work by enabling automatic parameter tuning and performance optimization.

4 High-Level Synthesis of RVC-CAL Dataflow Programs on GPU

4.1 Introduction

This chapter focuses on the high-level synthesis of RVC-CAL dataflow programs in the context of a CPU/GPU co-processing platform. The first section provides a description of the tool flow that has been developed and the CUDA technology used in this work. Some of the development presented in the follow-up sections have been published in various venues. For example, the CPU/GPU co-processing model [9] has been published, as well as various optimizations such as inter-actions parallel executions [13], and dynamic SIMD parallel executions [12, 15]. These optimizations will be further developed in subsequent sections.

4.2 CUDA Generation Tool Flow

In this section, the synthesis pipeline of the tool developed in this thesis will be presented first. Afterwards, the CUDA programming model used in the synthesis process will be described.

4.2.1 CUDA Design Pipelines

Figure 4.1 is a specific representation of the more general Figure 3.5 for this thesis project. It illustrates how the ORCC compiler uses the Exelixi CUDA backend, developed in this thesis, to generate C++ and CUDA code from the RVC-CAL representation. The XCF file informs the compiler which type of actors should be generated. Actors assigned to a numbered partition are generated as CPU actors in C++, while actors assigned to the "PG" partition (Partition GPU) are generated as GPU actors using C++/CUDA. All the necessary connections and partitions are generated accordingly, as explained in further detail in subsequent sections.

Figure 4.2 shows the files generated during the synthesis phase. The *lib* folder contains all the static files required for the project runtime, including definitions of data structures and classes, and helper functions. These files are identical regardless of the project being synthesized and the compiled binary can be shared between different projects. The *src* folder contains code

specific to the dataflow program being generated. Each CPU or GPU actor gets a source and header file, while *Network.cu* is the main file that instantiates all actors, connects them with the appropriate FIFO buffer, and launches them in the appropriate partitions. The *Network.xcf* is the configuration file that describes the partition and mapping configuration used for the synthesis of this project.

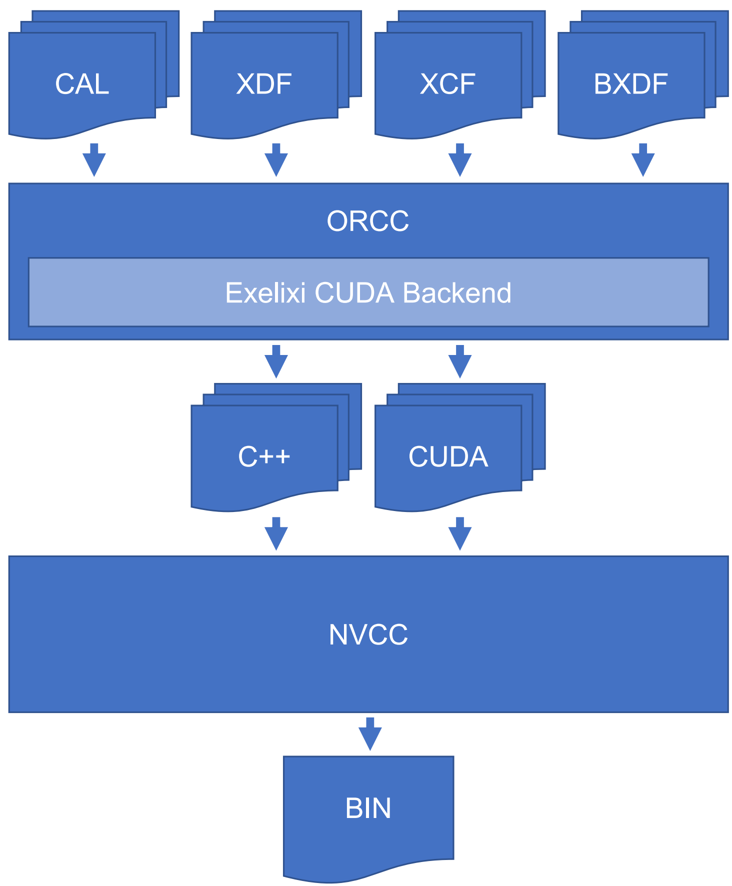


Figure 4.1: Overview of the code generation process within the ORCC compiler framework combined with the Exelixi CUDA backend.

4.2.2 CUDA Programming Model

NVIDIA introduced CUDA (Compute Unified Device Architecture), a general-purpose parallel programming model and application programming interface (API) that leverages the parallel computing capability of NVIDIA GPUs to solve complex computational problems more efficiently than on a CPU.

CUDA offers a low-level API as well as a higher-level API, providing developers with a fine-grained control over the GPU hardware while also offering a productive level of abstraction. It comes with a software environment that enables developers to use C/C++ or FORTRAN as a high-level programming language to jointly develop CPU and GPU processing, as well as

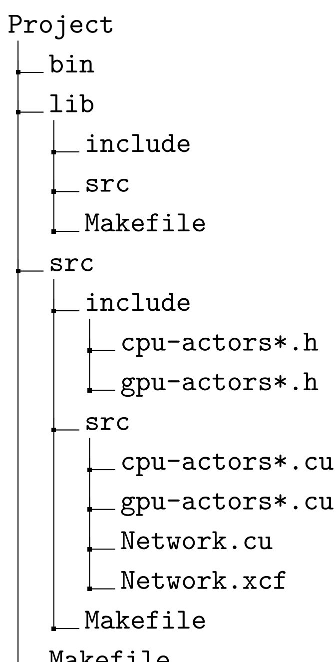


Figure 4.2: Structures of the folders and files generated by the Exelixi CUDA backend.

communication and synchronization between the two.

The challenge in programming GPUs lies in developing application software that transparently scales its parallelism to take advantage of the growing number of processor cores. To address this challenge, CUDA provides abstraction layers based on threads, thread groups, shared memory, and barriers for synchronization. It is important to note that threads are executed in warps, which are collections of 32 threads, and blocks of threads can contain multiple warps.

This decomposition preserves the expressiveness of the language by allowing threads to collaborate when computing subtasks and enables automatic scalability. In fact, each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially. As a result, a single compiled CUDA program can execute on any number of multiprocessors, as illustrated in Figure 4.3. Only the runtime system needs to be aware of the exact hardware layout.

CUDA threads may access data from multiple memory spaces during their execution, as illustrated in Figure 4.4. Each thread has access to its own private local memory. All threads in a thread block have access to a shared memory that is visible to all threads in the block and has the same lifetime as the block. All threads also have access to the same global memory.

The CUDA programming model enables heterogeneous computing, where both the CPU and GPU work together to perform tasks. A CUDA program is composed of both CPU code and GPU code. The CPU code prepares and launches functions, called *kernels*, to be executed on

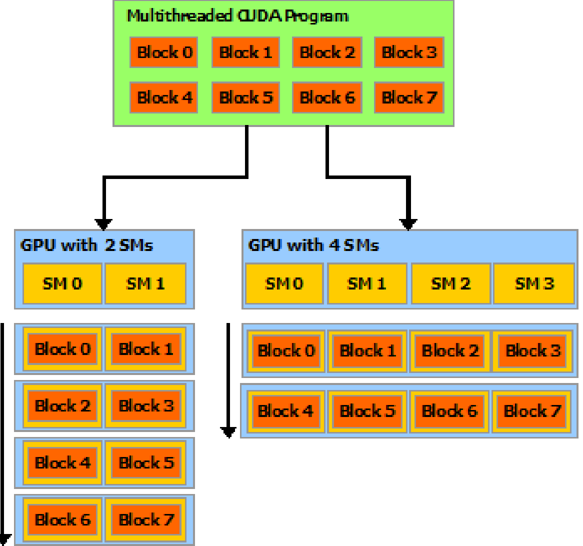


Figure 4.3: Illustration of the automatic scalability allowed by the CUDA programming model [8].

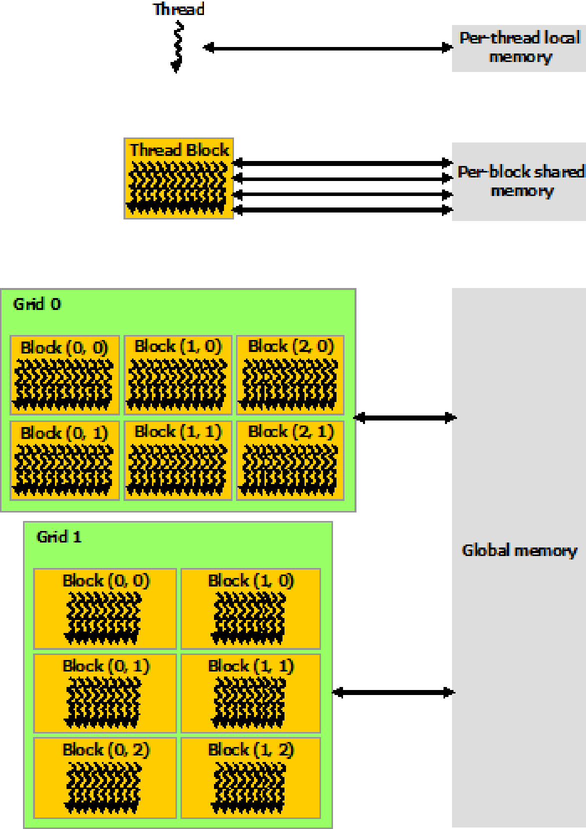


Figure 4.4: Illustration of the memory hierarchy used in CUDA compatible hardware [8].

the GPU.

As demonstrated in Figure 4.5, the CUDA programming model is based on the assumption that CUDA threads run on a physically separate device, such as a GPU, which operates as a co-processor to the host running the C++ program. This scenario is commonly seen, for instance, when the CUDA kernels are executed on a GPU while the rest of the C++ program is executed on a CPU. This setup allows for efficient and effective utilization of the computational resources available in the system.

The CUDA programming model also assumes that the host and device have separate memory spaces in DRAM, referred to as host memory and device memory, respectively. The program manages the memory spaces accessible to kernels via calls to the CUDA runtime system, which includes allocating and deallocating device memory and transferring data between host and device memory. This is described in detail in the CUDA Programming Interface.

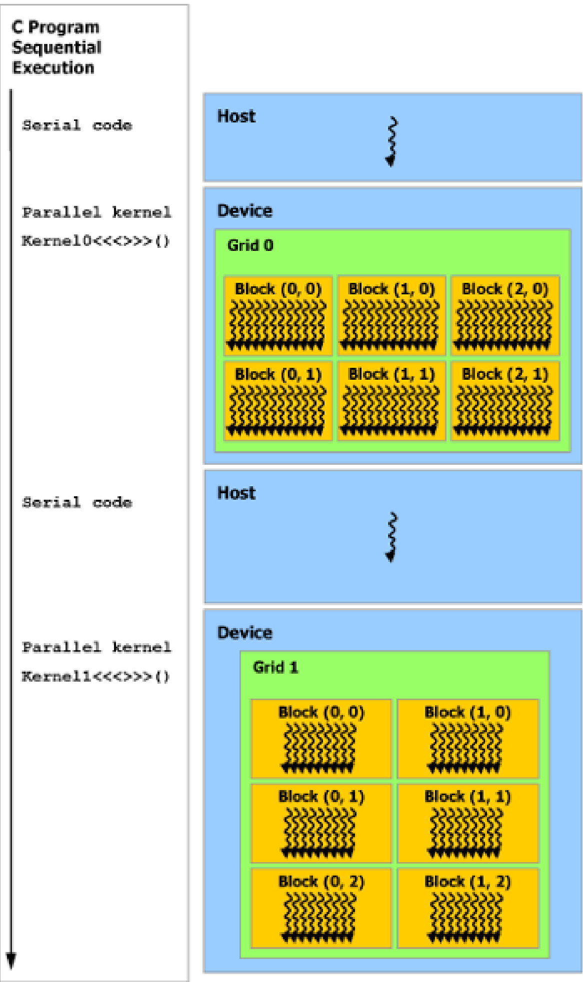


Figure 4.5: Illustration of how a typical CUDA codes is organized with sequential CPU section and parallel GPU code [8].

4.3 Model for CPU/GPU Co-Processing

This section describes the methodology for automatically generating C++/CUDA code for executing DPN dataflow software on heterogeneous CPU/GPU co-processing platforms. The focus is on the partitioning of processing tasks between the CPU and GPU, the independent processing of each actor, and the data communication between them.

4.3.1 CUDA Code Generation

As explained in the previous section, this work utilizes the CUDA application programming interface to run DPN actor networks on NVIDIA GPUs. However, the methodology and approach are general and can be extended to other GPU interfaces and platforms as long as they provide similar control and appropriate APIs. By utilizing CUDA, the researchers can take full advantage of the fine-grained control over the diverse range of NVIDIA hardware, including various generations and families of GPUs. The CUDA notation for multiple computing contexts is extensively used, including dynamic parallelism, and the concurrency between memory transfers and computation.

As previously presented, a typical CUDA application program consists of both host (CPU) and device (GPU) code in a single-source program, differentiated using pre-processor directives. The device code is written in the form of *kernels*, which are special functions that can be invoked from the host code and executed on the device. These kernels will be used to represent GPU actors in our model.

The approach taken in this work is to execute both the action selection and the actions themselves on the GPU for selected dataflow actors in the dataflow network that can benefit from GPU execution. This allows the necessary data to fire an action to be readily available in the device memory, resulting in a reduction of data transfers between the host and device memory.

By fully porting an actor's execution to the GPU, as opposed to the traditional approach found in the literature that uses GPUs as co-processing units, this approach eliminates the need for allocating a separate CPU core for each actor to schedule and launch kernels for GPU co-processing.

Actors of the dataflow network can be either fully executed on the host (CPU) or on the device (GPU). To implement this porting of dataflow network nodes, three different FIFO communication mechanisms are required: one for communication between actors mapped on the CPU, one for communication between actors mapped on the GPU, and one for communication between an actor mapped on the CPU and one mapped on the GPU. A different implementation is required for each type of communication mechanism.

These three FIFO communication implementations enable both independent and parallel computation between the host and the co-processing device and full parallel execution of

actors on the co-processing device. (More details are provided in Section 4.3.4).

As previously noted, the RVC-CAL programming language, models an actor as an atomic kernel of execution that consumes tokens from the input buffer and produces tokens in the output buffer. As such, actors are not inherently designed for internal parallelization. However, the performance of an actor's execution can be improved by parallelizing its computation.

Performance improvements can be achieved by launching multiple instances of the same action in parallel on different data, as long as the order of tokens within the communication channels between actors is preserved and the internal state dependencies of each actor are respected. This will be discussed in further detail in Section 4.4.1.

Before generating the execution code for an RVC-CAL application program, designers must provide a partitioning configuration. This involves assigning each actor or sub-network to either the CPU side, with a specific sub-partition (for multicore systems), or the GPU side. This assignment enables the compiler to generate the actor code intended for execution on the CPU using the CPP backend, and to generate the GPU platform code using the newly developed Exelixi CUDA backend.

The code generated for the *main* function is responsible for creating actor instances, creating partition using *pthread* to be executed on the CPU, instantiating GPU actors and launching their corresponding main kernels in separate streams (as detailed in Section 4.3.2), and instantiating and connecting the appropriate FIFO implementations between actors' ports.

In Listing 4.1, the code generated by the backend for the GPU-mapped *Producer* actor is presented. The *__Global__* directive declares *kernel* functions. The *action_selection* function is the principal kernel of the code, launched from the *main*. It is responsible for determining the next action to be executed, by checking for available data, available FIFO output space, and by evaluating constraints expressed by the guard, represented by the *isSchedulable_action* function. As depicted, the actions are also *kernel* functions, launched from the *action_selection*. This is made possible thanks to the dynamic programming feature provided by the CUDA API, which allows to select the number of SIMD threads available for the execution of each action, based on the executed code (refer to Section 4.4.1).

As an optimization, in the general case, if the number of SIMD threads used to execute the parent kernel (*action_selection*) matches the optimal number of threads for a specific action, the kernel call and the *cudaDeviceSynchronize()* call (line 11 and 12 in Listing 4.1) can be replaced by a simple function call. This eliminates the overhead introduced by the CUDA runtime that is not necessary in this case.

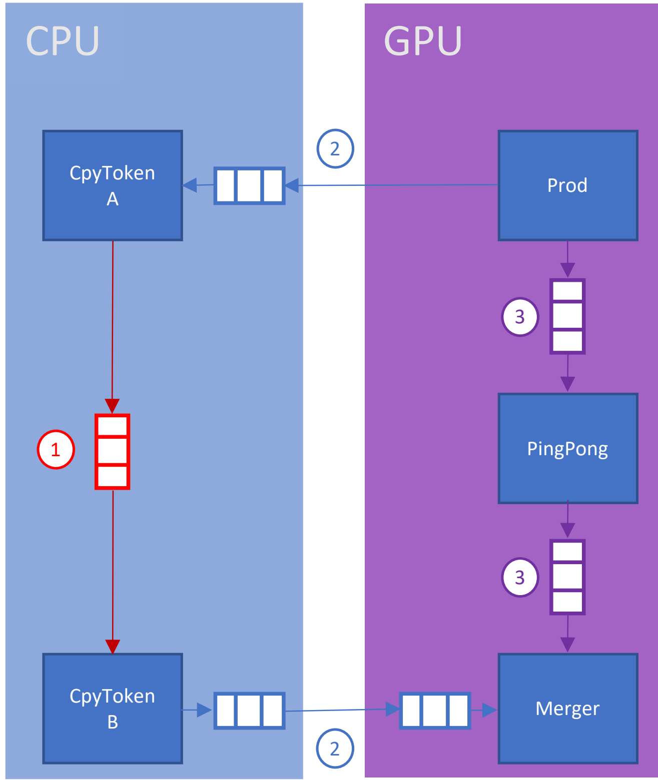


Figure 4.6: Program partitioning over CPU and GPU. Four types of FIFOs are used (1-CPU FIFO, 2-Legacy-HostFifo, 3-HostFifo, 4-Device FIFO.

Listing 4.1: Striped down example of the *CUDA* implementation of the action selection of the *Producer* actor

```
__device__ bool Prod::isSchedulable_p() {...}
__global__ void ProdNS::p(Prod* prod) {...}

__global__ void ProdNS::action_selection(Prod* prod) {
  prod->status_0_ = prod->port_0->rooms();
  bool res = true;
  while (res) {
    res = false;
    if(prod->isSchedulable_p()) {
      if(prod->status_0_ >= 1 ) {
        ProdNS::p<<<1, 1, 0>>>(prod);
        cudaDeviceSynchronize();
        res = true;
      }
    }
    __syncthreads();
  }
}
```

4.3.2 Independent GPU Actors

To optimize GPU resource utilization, multiple independent compute contexts are employed to run different actors concurrently with multiple CUDA streams. Figure 4.7 illustrates a possible execution of the dataflow program depicted in Figure 3.3, where each actor runs on its own isolated CUDA stream and If four tokens are generated by the producer actor. The number of parallel actions that can be executed depends on the availability of GPU device resources, such as CUDA cores, memory, and registers. As shown in green in Figure 4.7, multiple memory transfers can occur simultaneously by utilizing the copy engines of the GPU device. However, the number of copy engines may vary based on the hardware generation and model. Additionally, the number of available copy engines can impact the overlap between memory transfer and compute time, potentially affecting the performance of the dataflow program.

In Figure 4.7, the first firing of the action p by the *prod* actor can be seen in slot 1 (in red). The generated token of p is then made available to the *PingPong* actor in slot 2. As shown in Figure 4.6, the token must then be transferred to the CPU, where it is processed by the action c of the *CopyTokenA* actor in slot 3. In slot 3, it can also be observed that there is simultaneous CPU and GPU computation as well as a transfer of memory from GPU to CPU.

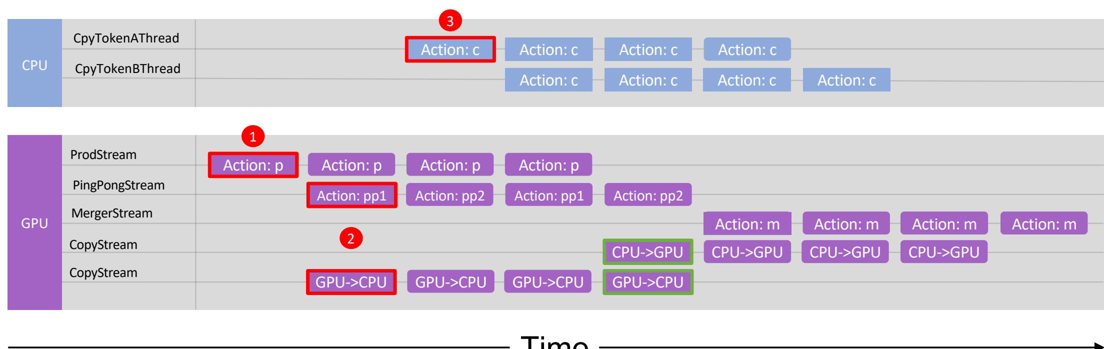


Figure 4.7: Example of the execution of the dataflow program depicted in Figure 3.3. In this example, four tokens are generated by the producer actor, and actors are mapped over both a CPU and a GPU, using multiple CUDA streams to showcase multiple independent actor executions.

4.3.3 GPU Partitions

In most of the literature, the GPU is typically viewed as a secondary processing platform that only executes tasks that are scheduled by the CPU, rather than as a standalone platform. This leads to an excessive use of CPU processing resources just for scheduling purposes.

The objective of this section is to demonstrate how each GPU-mapped actor can execute autonomously. Using the formalism presented in [9], each GPU actors behave as it is it's own

partition. Each actor is responsible for monitoring its own termination status as determined by the application software, and thus the GPU partitions can be considered fully independent for the duration of the application. To achieve this goal, after all the instantiation and initialization phases are completed, the main program thread waits for all CPU and GPU partitions to stop before terminating the application. This is achieved through each partition (whether CPU or GPU) keeping track of its actors' progress. If, after a set amount of time, no progress has been made by any actors (i.e., no actions have been fired), the partition terminates.

The implementation of the new GPU partition design is described in more detail below. The low-level scheduling generated by the RVC-CAL backend, referred to as *action_Selection*, has been implemented as a long-running kernel. Figure 4.2 provides a simplified illustration of this implementation. As can be seen from line 25, the *status* array is updated every time an action is triggered. This array is accessible to all other GPU actors and is used to inform them that one of the actors has completed further processing. The *checkStatus* function in line 30 checks the status of the other actors. If no actors have produced any results after a predetermined period of time (*waitPeriod*), the actors terminate. Finally, the software application terminates after all CPU-side and GPU-side actors have completed execution.

4.3.4 Data Communications

To support CPU/GPU co-processing execution, three types of FIFOs are required, named: *Fifo*, *CudaFifo*, and *HostFifo*. These are depicted in Figure 4.6. The first type, depicted in red and inherited from the original Exelixi C++ backend, is used for communication between CPU actors. The second type, depicted in purple, is used for communication between GPU actors and has an implementation similar to the CPU FIFO, with the exception of using device memory instead of CPU memory. The third type, depicted in blue, is used for communication between the CPU and GPU and requires at least one writer or reader to be on the CPU side and one on the GPU side.

A *HostFifo* is a cross-platform FIFO buffer that has been designed to take advantage of recent CUDA APIs and the accompanying hardware capabilities. To achieve this, *pinned* memory is allocated on the CPU RAM, which prevents the operating system from swapping this allocation to disk or moving it to a different physical address space. This type of allocation gives GPU actors direct access to the memory and enables them to access it efficiently.

The addresses of the pinned memory are registered in the GPU's virtual address space and the corresponding pointers are obtained. This results in different pointers being used for memory access from the CPU side and the GPU side. The advantage of this implementation is that it eliminates the need for any synchronization or software API calls, as memory accesses are performed automatically in hardware. This efficient implementation does not affect the dataflow model of computation or the use of FIFOs.

However, this third type of FIFO implementation is only compatible with hardware with a

computing capability of 6.x or higher, which supports fine-grained memory pinning. For hardware with compute capabilities below 6.x, a *Legacy-HostFifo* is provided. It consists of two identical *CudaFifo* buffers, one allocated on the CPU side and the other on the GPU side, along with additional functions (*hostFifoSyncWrite* and *hostFifoSyncRead*) that need to be called from the CPU to synchronize them. Since these FIFOs accept a single writer and multiple readers, it is necessary to keep track of which reader/writer is on which side to identify the most up-to-date data and determine the direction of the explicit memory transfers. However, the continuous use of CUDA software APIs to synchronize the two FIFOs introduces a significant runtime overhead compared to the hardware-optimized *HostFifo* implementation.

The subsequent sub-sections present the results of the evaluation of two application programs aimed at evaluating the performance gains achieved by replacing the inter-partition communication (CPU/GPU and GPU/GPU) implementations with the new *HostFifo* approach, compared to the previous *Legacy-HostFifo*.

The platforms used to conduct the experiments in the following sections have the following specifications:

- System 1: Intel Skylake I5-6600 CPU with 16 GB of DDR4 RAM, paired with a GeForce GTX 1660 SUPER NVIDIA GPU with 6 GB of memory.
- System 2: AMD Threadripper 3990X CPU with 256 GB of DDR4 RAM, paired with a GeForce RTX 3080 Ti NVIDIA GPU with 12 GB of memory.

Both systems run CUDA version 11.6.2 for the GPU software library.

4.3.4.1 RVC-CAL FIR Filter

Figure 4.8 shows the dataflow network of a simple FIR filter application program, consisting of 13 actors. The source code for this network is publicly available on the web repository, which can be accessed at the following URL [93].

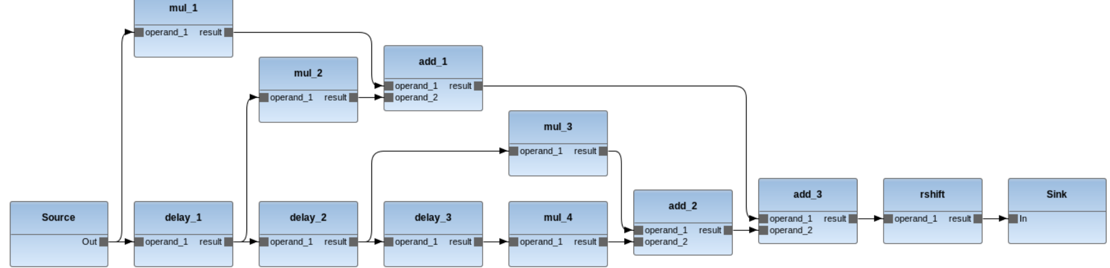
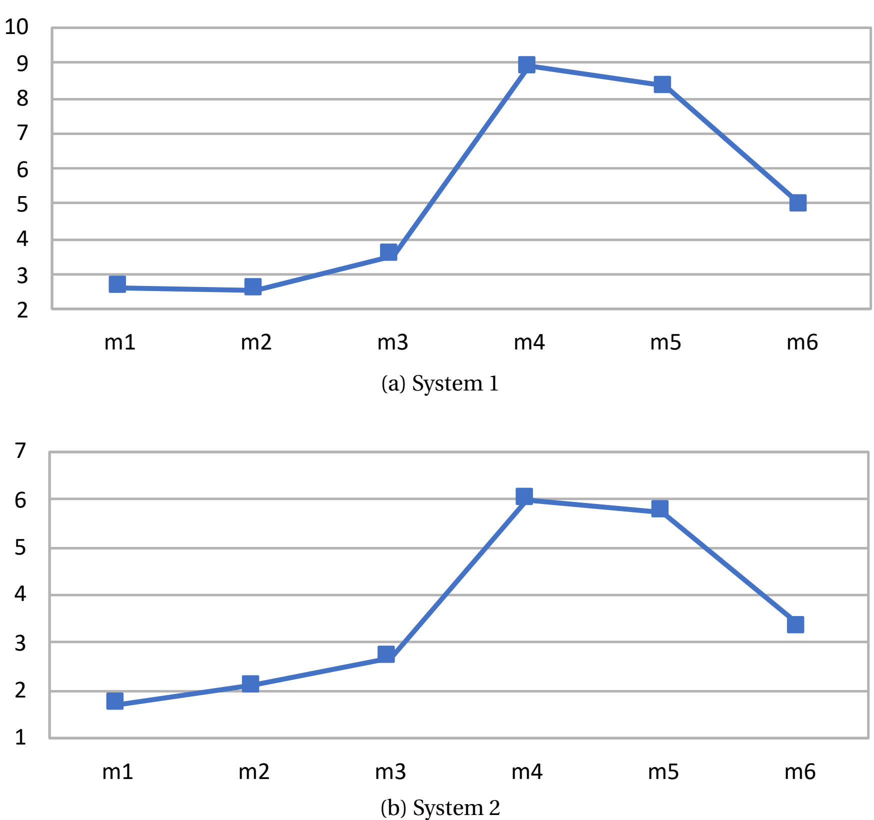


Figure 4.8: RVC-CAL FIR filter dataflow network: actors and communication FIFO buffers.

Figure 4.9 displays the performance speedup achieved by the new *HostFifo* implementation compared to the previous *Legacy-HostFifo* implementation. In order to assess the performance

of CPU/GPU data throughput, a complex dataflow network of actors, each performing simple internal processing, was chosen as a validation test. Thus, the performance of a FIR application was evaluated by comparing the results of six randomly selected mappings (refer to Table 4.3) of the actors to either the CPU or GPU platform on the two hardware systems. The results are compared to the performance of the previous inter-platform communication mechanism and the same mapping/partition. It can be observed that the improvement in processing speed ranges from 1.70 to 8.91 times.



(a) System 1

(b) System 2

Figure 4.9: Speedup results for the RVC-CAL FIR filter. On the x-axis the different mapping configurations and on the y-axis the speedup value with the application implemented using the new FIFO inter-partition methodology.

4.3.4.2 RVC-CAL JPEG Decoder

Figure 4.10 presents the top-level network of the JPEG decoder application program, consisting of 6 actors. This widely known application can also be found in the open-source orc-apps repository [93].

As with the results reported in previous sections, Figure 4.11 shows the speedup obtained by the new *HostFifo* implementation for six randomly selected configurations of the JPEG decoder. In this experiment, the performance of six arbitrary partitions (refer to Table 4.6) of

	New Fifo				Old Fifo				Speedup
	Min	Mean	Max	Var	Min	Mean	Max	Var	
m1	1.36	1.37	1.38	1.00E-4	3.56	3.60	3.63	1.23E-3	2.63
m2	1.31	1.34	1.35	4.33E-4	3.40	3.42	3.45	8.33E-4	2.56
m3	1.37	1.38	1.41	5.33E-4	4.90	4.91	4.92	1.33E-4	3.55
m4	5.73	5.76	5.80	1.43E-3	51.03	51.27	51.65	1.09E-1	8.91
m5	5.99	6.16	6.26	2.26E-2	51.23	51.30	51.37	4.90E-3	8.32
m6	5.79	5.80	5.82	3.00E-4	28.55	28.63	28.71	6.43E-3	4.94

Table 4.1: Statistics of the results depicted in Figure 4.9a using 10 executions.

	New Fifo				Old Fifo				Speedup
	Min	Mean	Max	Var	Min	Mean	Max	Var	
m1	1.25	1.26	1.27	1.33E-4	2.11	2.13	2.17	1.03E-3	1.70
m2	1.25	1.28	1.31	9.33E-4	2.65	2.67	2.68	3.00E-4	2.09
m3	1.30	1.31	1.32	1.00E-4	3.48	3.52	3.56	1.63E-3	2.68
m4	4.98	5.04	5.17	1.20E-2	30.10	30.24	30.42	2.72E-2	6.00
m5	5.23	5.26	5.31	1.73E-3	30.00	30.20	30.51	7.52E-2	5.74
m6	5.15	5.19	5.26	3.70E-3	17.16	17.31	17.53	3.72E-2	3.34

Table 4.2: Statistics of the results depicted in Figure 4.9b using 10 executions.

	CPU	GPU
m1	Source, Sink	delay_1, delay_2, delay3, mul_1, mul_2, mul_3, mul_4, add_1, add_2, add_3, rshift
m2	Source, Sink, delay_1, mul_1, mul_2, add_1, add_2, rshift	delay_2, delay3, mul_3, mul_4, add_2
m3	Source, Sink, mul_1, add_3, rshift	delay_1, delay_2, delay3, mul_2, mul_3, mul_4, add_1, add_2
m4	Source, Sink, delay_1, delay_3, mul_1, mul_3, add_1, add_3	delay_2, mul_2, mul_4, add_2, rshift
m5	Source, Sink, delay_1, delay_3, mul_1, mul_3, add_1, add_2	delay_2, mul_2, mul_4, add_3, rshift
m6	Source, Sink, delay_1, delay_2, delay_3, mul_1, mul_2, mul_3, mul_4	add_1, add_2, add_3, rshift

Table 4.3: The different mappings settings of the FIR implementation that are used in the results.

the actors to the GPU or CPU platform on the two different systems is presented and compared with the results of the previous GPU/CPU data communication buffer implementation. The speedup improvement of the overall application program execution ranges from 2.94 to 15.01 times, as observed in this test.

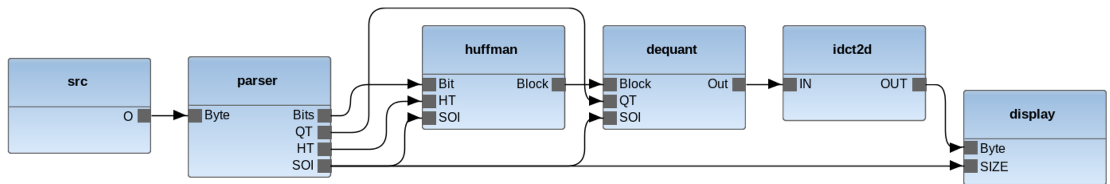
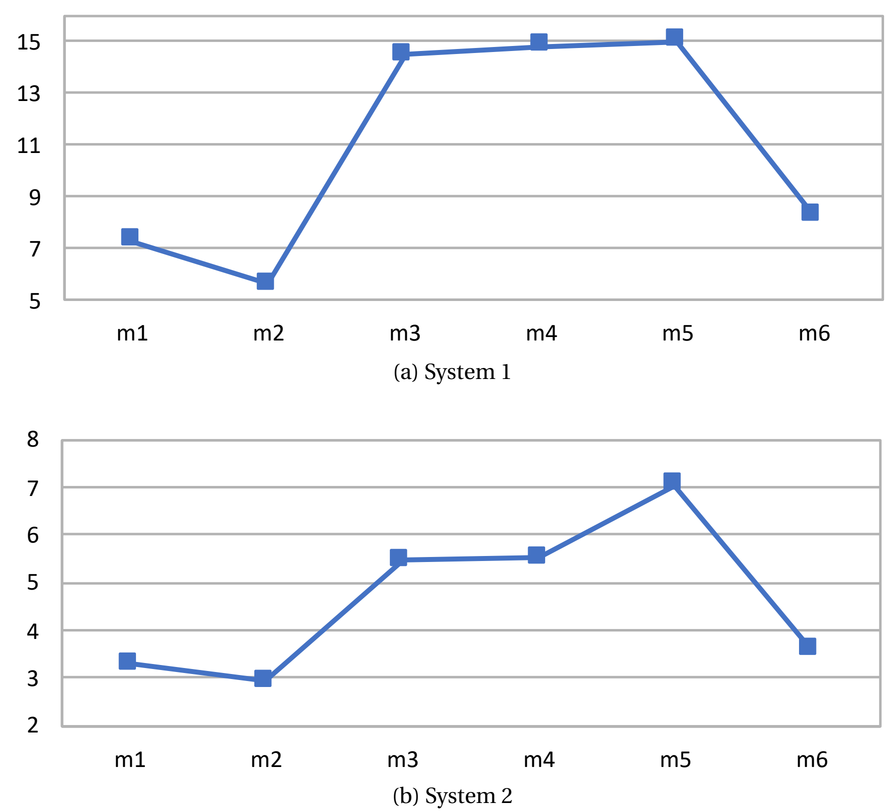


Figure 4.10: Network for the RVC-CAL implementation of a JPEG decoder.



(a) System 1

(b) System 2

Figure 4.11: Speedup results for the RVC-CAL JPEG decoder. On the x-axis the different mapping configurations and on the y-axis the speedup value with the application implemented using the new FIFO inter-partition methodology.

4.4 Generation Features

This section presents various optimization techniques that can be enabled independently within the Exelixi CUDA backend to improve performance or provide additional functionality in the automatically generated code. The choice of optimization technique to be applied depends on the specific context and requirements.

	New Fifo				Old Fifo				Speedup
	Min	Mean	Max	Var	Min	Mean	Max	Var	
m1	2.07	2.07	2.07	1.33E-04	14.98	15.08	15.15	7.11E-2	7.31
m2	1.88	1.89	1.89	5.33E-04	10.47	10.58	10.65	8.14E-2	5.64
m3	5.03	5.10	5.23	1.14E-01	73.04	73.70	74.51	5.00	14.26
m4	5.57	5.59	5.62	5.63E-03	82.61	82.72	82.81	9.31E-2	14.73
m5	9.57	9.60	9.62	6.70E-03	143.39	143.96	144.43	2.50	15.01
m6	7.70	7.72	7.75	7.23E-03	62.96	63.36	63.64	1.14	8.21

Table 4.4: Statistics of the results depicted in Figure 4.11a using 10 executions.

	New Fifo				Old Fifo				Speedup
	Min	Mean	Max	Var	Min	Mean	Max	Var	
m1	1.72	1.72	1.72	4.30E-5	5.44	5.67	5.90	4.84E-1	3.29
m2	1.31	1.31	1.32	2.52E-4	3.81	3.86	3.94	4.42E-2	2.94
m3	4.56	4.57	4.57	2.44E-4	24.93	24.95	24.99	8.83E-3	5.46
m4	5.12	5.13	5.14	7.46E-4	27.59	28.35	29.35	7.41	5.52
m5	8.44	8.45	8.46	5.37E-4	58.56	59.51	61.25	2.03E+1	7.04
m6	6.93	6.94	6.95	6.13E-4	23.62	24.87	25.52	1.06E+1	3.58

Table 4.5: Statistics of the results depicted in Figure 4.11b using 10 executions.

	CPU	GPU
m1	src, parser, huffman, dequant, idct2d, display	dequant
m2	src, parser, huffman, dequant, display	idct2d
m3	src, parser, dequant, idct2d, display	huffman
m4	src, parser, dequant, display	huffman, idct2d
m5	src, huffman, idct2d, display	parser, dequant
m6	src, display	parser, huffman, dequant, idct2d

Table 4.6: The different mappings settings of the JPEG decoder implementation that are used in the results.

4.4.1 SIMD Parallel Execution

This section details the utilization of Single Instruction Multiple Data (SIMD) parallelization techniques to speed up the runtime of dataflow actions, thereby enhancing the overall execution performance.

4.4.1.1 Design Methodology

Efficient generation of low-level SIMD-style code is essential for fully leveraging the capabilities of modern GPUs. Without this, only a single thread out of the 32 possible will be utilized on a NVIDIA GPU, as each CUDA core is a SIMD architecture with 32 threads, leading to

substantial waste of resources. To achieve this goal, it is necessary to understand the limitations of the computation model used for programming the platform. In the case of RVC-CAL, actors consume one or more tokens from input buffers, perform the computation defined by the action, modify their internal state, and then produce one or more tokens in the output buffers. This intrinsic behavior of actors makes the implementation not automatically suitable for SIMD-style parallelization. However, it is possible to improve the performance of actor execution by preserving the order of tokens within the actor communication channels and respecting the state dependencies of the model of execution in the actor.

In contrast to the methodology described in the previous section, where the *action_selection* and the actions was executed by a single CUDA thread, this work proposes a new approach. The dynamic parallelism necessary for executing the *action_selection* function is achieved through the use of a second, dedicated kernel for the actions. This approach enables SIMD parallel execution of multiple instances of the same action simultaneously.

As an illustration of the approach presented in this section, a summary of the implementation is provided in Listing 4.2. This improved implementation of the *action_selection* is shown and, as can be observed from line 23, in this particular example, 512 threads are used to execute this kernel, resulting in 512 instances of the action being executed in parallel.

Another important aspect to consider is that the handling of read and write addresses in FIFOs is performed within the *action_selection* function, rather than in the action implementation itself. This allows the *action_selection* to be called only once and in a sequential manner. In line 21, the read address is obtained by providing the number of tokens to be used at each call. For example, in the illustration, this number is 64 tokens multiplied by 512, the number of parallel instances. Proper sizing is critical to avoid data races and ensure that memory is aligned, allowing a specific amount of data to be accessed at consecutive addresses.

Within a network, it is possible to have both sequential and parallelized actors coexisting while maintaining the sequential semantics of the FIFO buffers. This can be achieved by implementing a mapping of token IDs and thread IDs within an action to properly index the read and/or write addresses.

Listing 4.2: Simplified implementation of a *CUDA* action selection function for an actor of the *idct* design illustrated in Figure 4.12

```
void action_selection(Idct* actor,
                      unsigned int* status,
                      unsigned int index,
                      unsigned int size) {
  unsigned int sTime = clock64();
  bool endExecution = false;
  do {
    status[index] = 0;
    bool r1 = true;
    while (r1) {
      r1 = false;
      actor->size_IN = actor->Prt_IN->count(0);
      actor->size_OUT = actor->Prt_OUT->rooms();
      bool r2 = true;
      while (r2) {
        r2 = false;
        if(actor->size_IN >= 512*64 &&
           actor->isSchedulableAction()) {
          if(actor->size_OUT >= 512*64) {
            Ports prts;
            prts.IN =actor->Prt_IN->read_address(0,512*64);
            prts.OUT =actor->Prt_OUT->write_address();
            idctNS::action<<< 1, 512 >>>(actor,prts);
            r1 = r2 = true;
            status[index] = 1;
          }
        }
      }
    }
    if (checkStatus(status, size) == NULL) {
      endExecution = (clock64() - sTime) > waitPeriod;
    } else {
      sTime = clock64();
    }
  } while(!endExecution);
}
```

Two application programs have been selected for evaluating the newly introduced methodology. When a new technology is used to generate executable code from a high-level dataflow program, two factors need to be taken into account. Firstly, it is important to verify that the generated executable code is semantically correct and equivalent to the source code in the dataflow representation. Secondly, it is essential to measure the performance improvement achieved in comparison to the original alternative.

4.4.1.2 IDCT Application

The results of the parallelization achieved by applying the new approach described in this section to a computationally intensive actor are presented here. To isolate the transformation and evaluate its potential, a test application consisting of a network of three actors was used. Two actors, the *Source* and *Sink*, provide the necessary input and output data flow, while the central actor *idct* implements a compliant Inverse Discrete Cosine Transform algorithm (refer to Fig.4.12 for a visualization of the dataflow network). To evaluate the performance of the proposed approach, two partitioning configurations were tested. The first configuration utilized only CPU processing, with one thread assigned to each actor and a total of three threads. The second configuration utilized GPU processing, with the CUDA kernel of the *idct* actor's action executed on a grid of 2 blocks, each with 512 threads. Both configurations used buffers of the same size. The results showed that the GPU configuration processed the data in less than half the time of the CPU-only configuration. These results are detailed in Table 4.7. These results demonstrate that the proposed methodology can provide a significant performance boost over implementations relying solely on multicore CPU processing, provided that sufficient data throughput can be achieved.

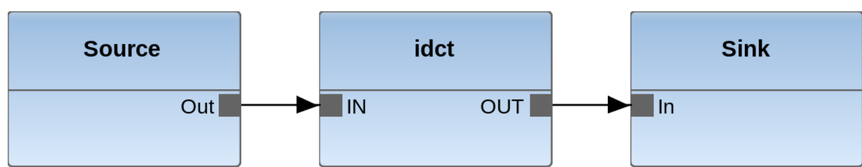


Figure 4.12: Illustration of the test RVC-CAL IDCT dataflow network.

	CPU				GPU				Speedup
	Min	Mean	Max	Var	Min	Mean	Max	Var	
System 1	9.23	9.31	9.42	2.90E-3	4.36	4.60	4.77	1.40E-2	2.02
System 2	8.50	12.57	17.97	1.39E+1	5.72	5.73	5.74	3.22E-5	2.18

Table 4.7: Speedup results with statistics for the the RVC-CAL IDCT application using 10 executions.

4.4.1.3 RVC-CAL JPEG Decoder

In this second experiment, the same JPEG decoder as in Section 4.3.4.2 is utilized. The parallelized *idct2d* actor, as described in the tests in the previous section, is integrated into the dataflow network implementing the full JPEG decoding algorithm. This experiment serves to validate the semantic correctness of the generated code for a generic dataflow program. The application is sufficiently complex and has not been explicitly designed and optimized for exploring efficient GPU parallelization. The experimental results in terms of performance are presented in Table 4.8, which compares the performance of the sequential GPU implementation of the *idct2d* actor, consistent with the methodology previously developed and reported,

with the improved implementation in which the *idct2d* actor is executed in parallel on the GPU platform. In both cases, all other actors are executed on the CPU partition. The results show that the parallel GPU implementation outperforms the sequential implementation by a speedup of 9.67 or 18.3, depending on the hardware system.

	Frame rate [image/sec]								Speedup
	Sequential GPU				Parallel GPU				
	Min	Mean	Max	Var	Min	Mean	Max	Var	
System 1	0.24	0.24	0.24	7.63E-8	2.28	2.28	2.28	4.06E-7	9.44
System 2	0.28	0.29	0.29	2.38E-7	5.16	5.19	5.24	8.17E-4	18.20

Table 4.8: Frame rate and speedup results for the RVC-CAL JPEG decoder.

4.4.2 Inter-Actions Parallel Execution

This section presents an extension of the high-level dataflow compiler backend Exelixi CUDA which utilizes a CUDA feature called Dynamic programming to offer inter-actions parallel execution, thereby improving performance.

4.4.2.1 Design Methodology

As a reminder, is this methodology when an actor is mapped to the GPU, both the action selection and the execution of the action take place on the GPU, which optimizes data accesses. This also frees up CPU partitions to allocate a CPU core solely for scheduling actions, allowing it to be available for executing other actors. A further advantage is that it enables actors with a fully dynamic dataflow model of computation to run on the GPU. This approach results in multiple actors running in parallel on both CPUs and GPUs.

If the dataflow model of computation is strictly adhered to, parallelization occurs solely at the actor level. However, the number of potential parallel executions offered by explicit actor parallelism may not always align with the hardware capabilities of modern GPUs, depending on the application and the coding style of the developers. To increase resource utilization and improve overall performance, inter-actor parallelization at the granularity of actions can be implemented. However, this approach must maintain the same guarantees offered by the dataflow Model of Computation.

Figure 4.13 illustrates the execution of the example described in Section 3.2 if all actors are mapped to the GPU, utilizing the SIMD Parallel Execution developed in the previous section. In this scenario, each actor has a dedicated *cudaStream* (CUDA's software execution queue), executing actions sequentially. If multiple instances of the same action need to be executed in sequence, they can be executed in parallel, as demonstrated by the action *c* in actors *CpyTokenA* and *CpyTokenB*, where two actions are executed at a time. On the other hand, this technique is not effective in parallelizing actions *pp1* and *pp2* in the *PingPong* actor, as they

are different actions,despite the absence of any dependencies between them.

Figure 4.13: Example of the execution of the dataflow program depicted in Figure 3.3 with all actors mapped to the GPU and utilizing the SIMD Parallel Execution feature.

Figure 4.14 depicts the execution of the same example, utilizing both SIMD optimization and the newly introduced inter-action parallelization. With this approach, actions *pp1* and *pp2* in the *PingPong* actor, as well as action *m* in the *Merger* actor, can be parallelized. This is due to the fact that consecutive actions can be launched in parallel as long as there are no dependencies on state variables, with dependencies on input and output data being handled by the FIFO buffer, as explained in Section 4.4.2.3. A small delay is observed between the launch of actions *pp1* and *pp2*, reflecting the fact that the action selection is executed in parallel, preparing for the next action without waiting for the previous one to complete. This is made possible by each action, including the action selection, running in separate *CudaStream*. This approach differs from the SIMD approach used for action *c*, which involves a single CUDA kernel launch but executed with two threads in parallel. As shown in the figure, higher GPU resource utilization is achieved due to increased parallel opportunities, resulting in a shorter overall execution time.

4.4.2.2 Implementation of the Inter-Actions Parallel Execution

As mentioned in the previous section, inter-actions parallel execution is implemented through the use of the *CudaStream* feature and the *dynamic programming* API. This is a feature provided by the CUDA API, which is supported by NVIDIA hardware and allows a GPU kernels to dynamically launch other GPU kernels on NVIDIA boards. In our approach, we utilize device-side compute streams (*CudaStreams*) to manage dependencies between kernel launches (actions). The combination of these two features is handled by the Actors' scheduler (*action_selection*).

The parallelization process is illustrated in Figure 4.15. The main program launches CPU partitions (groups of actors) on CPU threads, while GPU actors are executed on *CudaStreams*.

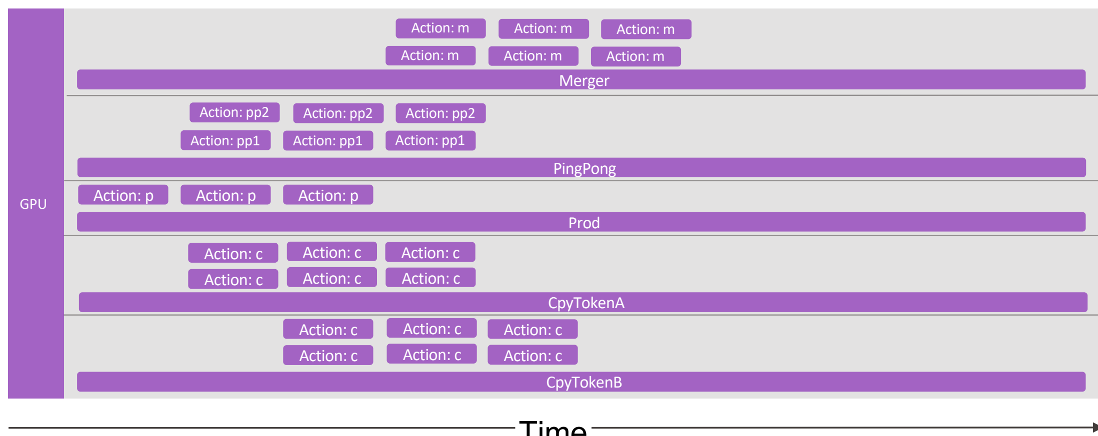


Figure 4.14: Example of the execution of the dataflow program depicted in Figure 3.3 with all actors mapped to the GPU and utilizing both the SIMD Parallel Execution feature and the inter-action parallelization optimization.

For GPU-based actors and actions that can be executed in parallel, separate *CudaStreams* are employed to run them.

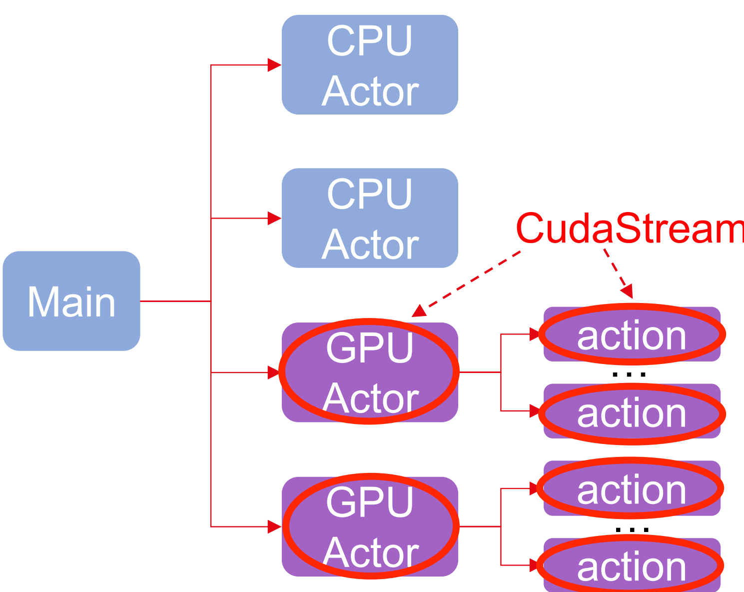


Figure 4.15: Parallel schema showcasing the parallel execution of two GPU actors and four actions by using separated cudaStream.

Listing 4.3 presents a simplified version of the scheduler for the *Merger* actor, where *Merger::m* is its only action. The *action_selection* is the main CUDA kernel of the actor, and the main loop is responsible for executing guards (*isSchedulable*), checking the inputs and outputs buffers for data and space, and if all conditions are met, launching a new action. The dynamic action

launch is performed in line 34 of the code, which is characterized by the "<<<" and ">>>" symbols.

In this code example, *cs* is an array of compute streams and *sid* is the identifier of the stream where this action execution will be queued. The call to *parallel_read_address* in lines 30 and 32 is used to obtain the pre-fetched reading pointers, as explained in Section 4.4.2.3. Finally, the *advance* function is launched on the same stream, which ensures that it will only be executed once the action is complete. The function is responsible for signaling the different parallel FIFOs that it has completed reading the requested data and that the memory can be freed.

4.4.2.3 Efficient Parallel FIFO

An important aspect in maintaining the consistency of the dataflow MoC is to ensure that actions execute as if they were executed sequentially. To achieve this, a modified version of the FIFO is used, which allows for parallel reads and parallel writes to the data while preserving the order-preserving property that is necessary for consistency. Figure 4.16 depicts a schematic representation of the parallel reading process in the aforementioned FIFO. The FIFO in the figure contains eleven tokens and is currently being read by four actions in parallel.

When an action needs to read, it specifies the number of tokens it requires for processing and receives a pre-fetched reading pointer (*pr1* through *pr4*) in return. This pre-fetched pointer is then incremented by the size of the requested read for the next read. The size is saved in the *size* array, and the *done* array reserves a spot. We can see that the second read (in green) has completed processing, and the corresponding *done* flag has been raised. *gr* is the global read pointer and represents the sequential reading status, which is advanced through the FIFO as consecutive reads are marked *done*, freeing up space in the FIFO. The writing process works in a similar way.

4.4.2.4 IDCT Application

In this section, we evaluate the performance improvement of the inter-actions parallel execution optimization using the IDCT RVC-CAL program described in Section 4.4.1.2. The platform-specific code was generated by the Exelixi CUDA backend and executed on the hardware platform referred to as *System 1* in the previous section.

Table 4.9 shows the performance results as the number of parallel actions increases. Both the total execution time and the speedup relative to the case with only one parallel action are reported. The case with a single stream serves as the baseline before optimization.

Listing 4.3: Stripped-down example of the *CUDA* implementation of the action selection of the *Producer* actor

```
_device_ bool Merger::isSchedulable_m() {}
_global_ void MergerNS::m(Merger* merger,Ports ports) {}
_global_ void MergerNS::m_advance(Ports ports) {}

_global_ void MergerNS::
                action_selection(Merger* merger,
                                EStatus* status,
                                size_t actorIdx,
                                size_t actorSize) {
  const size_t nb_stream(4);
  size_t sid(0);
  cudaStream_t cs[nb_stream];
  for (size_t i(0); i < nb_stream; ++i) {
      cudaStreamCreateWithFlags(&cs[i],
                                cudaStreamNonBlocking);
  }

  status_I1 = merger->I1->parallel_count(0);
  status_I2 = merger->I2->parallel_count(0);
  port_I1_slots = merger->I1->parallel_rd_slots(0);
  port_I2_slots = merger->I2->parallel_rd_slots(0);

  bool res = true;
  while (res) {
    res = false;
    if(status_I1 >= 1 && port_I1_slots >= 1 &&
        status_I2 >= 1 && port_I2_slots >= 1 &&
        merger->isSchedulable_m()) {
      Ports ports;
      merger->I1->parallel_read_address(
          ports.I1, ports.I1_done, 0, 1);
      merger->I2->parallel_read_address(
          ports.I2, ports.I2_done, 0, 1);
      MergerNS::m<<<1, 1, 0, cs[sid]>>>(merger, ports);
      MergerNS::m_advance<<<1, 1, 0, cs[sid]>>>(ports);
      --port_I1_slots;
      --port_I2_slots;
      status_I1 -= 1*1*1;
      status_I2 -= 1*1*1;
      res = true;
      stream_id = (stream_id+1) & (nb_stream-1);
    }
  }
}
```

The results show a graceful scaling of speedup with up to four parallel actions, offering significant performance improvement opportunities. The execution time with two and four parallel

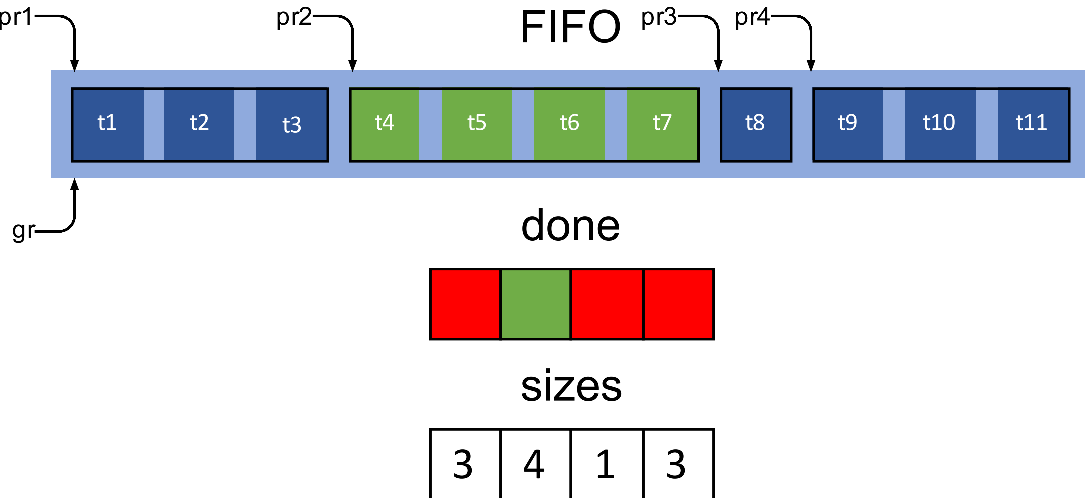


Figure 4.16: Illustration of the parallel reading process implementation in the FIFO buffer, with the writing process functioning similarly.

actions was 1.95 and 3.39 times faster than the baseline, respectively.

It should be noted that the lack of further improvement beyond four parallel actions is due to an undocumented software limitation in the current version of the CUDA API, which may be removed in future releases.

	1 stream	2 streams	4 streams	8 streams	16 streams
Execution time (sec)	30.99	15.93	9.15	9.15	9.15
Speedup	1	1.95	3.39	3.39	3.39

Table 4.9: Performance results of the IDCT application example, showcasing the impact of increasing the number of inter-action parallel executions from one to sixteen using multiple CUDA streams.

4.4.3 Dynamic Heterogeneous Actors

In this section, we introduce a methodology for generating dynamic RVC-CAL networks. Rather than undergoing the entire four-step compilation flow (depicted below) each time a new configuration needs to be evaluated, a single executable binary is created that allows for the specification of the partitioning and mapping at runtime during the application's startup process. This allows for testing of new configurations by simply changing the input XML file and re-running the program, eliminating the need for repeated recompilation.

1. Change the Exelixi CUDA backend parameters configurations files.

2. Generates code

3. Compile

4. Execute

The methodology utilizes the newly developed Exelixi CUDA backend option, which generates a shadow CPU version of each actor assigned to the parallel GPU partition. This shadow version is not connected by default in the network, and the proper version of the actor will be selected based on the input configuration file.

Figure 4.17 illustrates the methodology. In this example, the application program consists of four actors (A, B, C, D) connected in a back-to-back configuration by three FIFO buffers.

Figure 4.17a illustrates the former static methodology, in which the actors A and D are assigned to the CPU, and actors B and C are assigned to the GPU. In this scenario, the Exelixi CUDA backend generates the appropriate code to match the configuration requested by the developer.

Figure 4.17b displays the alternative dynamic methodology. In this scenario, actors B and C are assigned to the GPU, and the Exelixi CUDA backend automatically generates corresponding shadow actors B^* and C^*, which are targeted towards the CPU. During the runtime setup process, the appropriate versions of the actors' network are dynamically instantiated. The code handling the FIFO buffers (e.g. for reading, writing, and updates notification) has been modified such that the same FIFO instance can be utilized by actors running on either the CPU or GPU.

It should be noted that the current implementation restricts the dynamic setup to occur only once throughout the execution and cannot be altered during execution, particularly if the actor contains internal state variables. This limitation stems from the fact that the internal state of the actor is not mirrored in the current implementation. However, future releases of the tool may remove this limitation through new developments.

It should also be noted that in the two examples, the FIFOs connecting actors B and C are not identical (as indicated by the color difference). The FIFO in the static example is a specific GPU-to-GPU buffer that provides better performance. However, due to the dynamic nature of the second example, such specialization of the FIFO buffer is not possible and all buffers must be of type *HostFifo* to allow for dynamic switching between the actor and its shadow variant.

Since in the current development stage of the Exelixi CUDA backend, it is not possible to modify the mapping configuration during program execution. Therefore, as shown in Figure 4.17c, a third option is presented. Rather than connecting all actors and shadow actors using a universal *HostFifo*, only the necessary instances of the actor are created, and the appropriate specialized FIFO is utilized for communication.

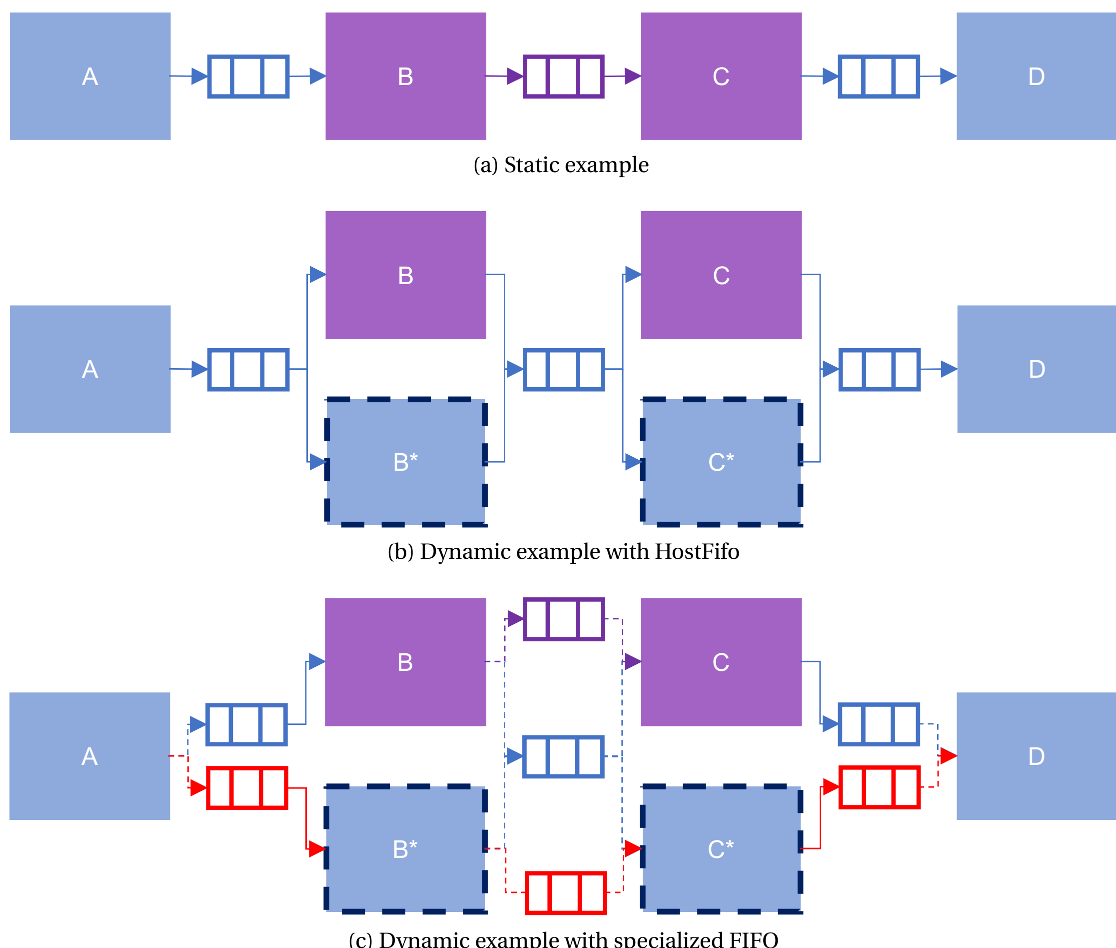


Figure 4.17: Illustration comparing the dynamic and static implementations of a dataflow network.

4.4.4 Dynamic SIMD Parallelization

This section describes a methodology developed to generate applications with dynamic SIMD parallelization, meaning that the number of parallel threads used to execute an action can be adjusted at runtime.

The dynamic SIMD parallelization optimization is based on generating multiple SIMD parallel executions of the same action, where the number of threads used dynamically changes over the runtime of the application, with the goal of maximizing performance and GPU resource utilization.

To achieve this goal, during the code generation phase, each action with a *parallel* flag is assigned a pair of integers $\{bl, th\}$, where bl represents the number of CUDA thread blocks and th represents the number of threads per block. Careful parametrization is required with this dynamic pair of variables, including the CUDA kernel launch parameters, the pre-

allocation of FIFO buffer token slots for the corresponding input or output ports, and the tests for available space for writing and available tokens to be read. Finally, the dynamic change must be synchronized with the end of the main inner loop to prevent size mismatches.

It should be noted that the maximum number of threads that can access a FIFO in parallel is related to the size of the FIFO's *threshold* and the number of tokens consumed or produced per firing of the action. In fact, for a FIFO buffer of size *"size"*, a memory of size "$size + threshold$"" is allocated. This is done because efficient execution performance requires SIMD threads to access consecutive memory locations. Thus, the threshold should be at least $Nb_{thread} * Nb_{tokens}$, where Nb_{thread} is the maximum number of SIMD threads that an action can be executed with and Nb_{tokens} is the number of tokens produced or consumed by the action to/from this FIFO buffer. In the static SIMD mode, this concern is not relevant because the Exelixi CUDA backend generates the proper size based on the flag set by the developer. However, in the Dynamic SIMD case, the threshold needs to be large enough to allow for effective parallelization opportunities.

An example of the internal representation of a FIFO buffer with a size of twelve tokens ($t1 - t12$) is shown in Figure 4.18. In this example, an action consumes three tokens per firing and is executed with four SIMD threads ($th1 - th4$) in parallel. As a result, the FIFO buffer requires a minimum $threshold$ size of $3 * 4 = 12$.

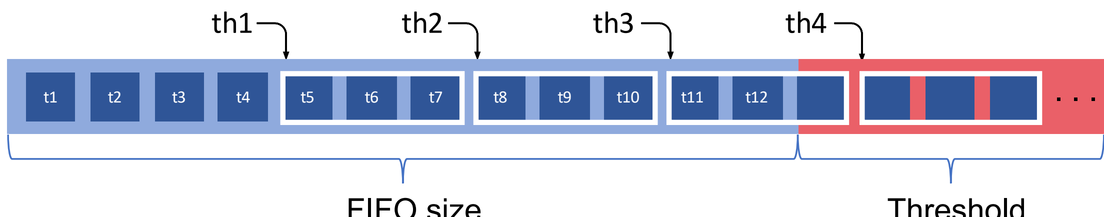


Figure 4.18: Representation of the SIMD parallel read/write of a FIFO buffer.

4.5 Conclusion

In this chapter, a complete end-to-end high-level synthesis of RVC-CAL dynamic dataflow programs on GPU architecture was presented, including both the designed model and its implementation. Three important components were presented.

The first part explains how the overall tool-flow design pipeline works, including the required inputs and several intermediate steps that lead to the final binary implementation. An intermediate step in this process is the generation of CUDA code, for which the programming model has been presented. This definition is crucial for the overall goal of this thesis because it determines the limitations and possibilities for designing a dataflow methodology mapping on GPU hardware.

In a second part, the actual model that has been developed for representing data flow in CPU/GPU co-processing hardware is discussed, with consideration given to the challenges outlined in Section 1.3. This includes the representation of actors, partitioning and mapping of actors, scheduling of actions, and data communication. The concepts, principles, and performance of these implementations have been successfully demonstrated using real-world standard applications.

In the final section, a list of generational features is presented that can be activated depending on the context to enhance the core execution model. The list begins with the SIMD parallel execution, which improves overall resource utilization by creating CUDA kernels that execute multiple consecutive action instances across multiple CUDA cores and on different input data read from the FIFO buffers. Next, inter-action parallel execution enables the actor scheduler to launch several distinct actions in parallel, provided there are no internal data dependencies. This approach is performed while still maintaining the consistency of the sequential semantics of the communication channels connecting actors. Lastly, dynamic heterogeneous actors and SIMD parallelization introduce a level of dynamism in the network configuration, which provides several benefits. These include ease of development and debugging, substantial support for design space exploration (as discussed in the subsequent chapter), and the potential for future work to enable automatic runtime reconfiguration to meet performance targets depending on input data sizes.

5 Automatic Design Space Exploration on GPU

5.1 Introduction

This chapter focuses on the work carried out in this thesis, which concerns the automatic design space exploration of RVC-CAL dataflow programs in the context of CPU/GPU co-processing platforms.

The first part of the chapter describes the tool flow and methodologies developed for clock-accurate profiling and performance metric generation. These metrics are utilized in the TURNUS post-processor to estimate the overall application execution time.

The second section details the utilization of metrics and performance evaluation strategies to devise a design space exploration methodology utilizing the Tabu search meta-algorithm.

Several of the results presented in the chapter have been published in various venues, including the clock-accurate profiling and performance estimation [10, 16, 14], and the Tabu search DSE [17] (currently under review).

5.2 Performance Estimation

This section explains the improvement over the Exelixi CUDA backend [94] for generating instrumented code for clock-accurate profiling and outputting the corresponding performance metrics. It also outlines how these measures can be utilized in the TURNUS post-processor to estimate the overall application execution time accurately.

5.2.1 Clock-Accurate Profiling

To achieve automated clock-accurate profiling, a new setting was introduced in the Exelixi CUDA backend. This improvement involved changes to the behavior of parallel GPU partitions and the inner scheduler of the GPU actors. To eliminate any interference with the

measurements, such as hardware resource access conflicts and memory transfers, the actors are executed sequentially, and a sequential partition is created to schedule at most one GPU actor and one CPU actor in parallel. The scheduling type is changed from fully parallel to non-preemptive, meaning that an actor executes actions until inputs, outputs, and predicates allow it, and then releases control back to the partition's scheduler.

The second improvement involved the implementation of a new profiling method. To obtain clock-accurate metrics, the ability to read platform counters that increment at a constant and known frequency is necessary. For actors running on the CPU, the *RDTSC* Intel register, as in [95], is used. For actors running on the GPU, a different approach is needed. The platform used for the results in this thesis is NVIDIA hardware, which provided equivalent functionality for profiling CUDA's streaming multiprocessors. Listing 5.1 shows the SASS assembly code used to access a performance register with a steady clock-cycle increase rate. These calls could be placed on opposite sides of the code section to be profiled.

In contrast to the software solution for homogeneous hardware presented in previous works, it cannot be assumed that the data transmission time for a typical action will be comparable regardless of the chosen partition (CPU or GPU) transmitting the data. In fact, cross-platform transmission (i.e., GPU to CPU) is far more resource-intensive. To account for these differences, the organization of the action software was revised to distinguish data transmission from action computation. Figure 5.1 shows that, in the revised approach, each input necessary for the action computation is read in a sequential manner before the action body, and the tokens produced as a result of the computation are written to the outputs in a similar manner. This allows for accurate differentiation and measurement of communication and execution weights.

The metrics from both the GPU and CPU sides are collected in the *profiling* class, which is responsible for computing the data statistics (mean, variance, min, max, and Gaussian filtering) and writing the results to three XML files: *weight.cxdf*, *weight.exdf*, and *weight.sxdf*. These files contain the measurements of the communication, action body computation, and scheduling metrics, respectively.

Listing 5.1: Simplified example of the utilization of NVIDIA's assembly language (i.e. *SASS*) was required for reading the GPU's stable autoincrementing register.

```
// -- PROFILE: START
asm volatile("mov.u64 %0, %%clock64;" : "=l"(__clock_1) :: "memory");

// section of code that needs profiling ....

// -- PROFILE: STOP
asm volatile("mov.u64 %0, %%clock64;" : "=l"(__clock_2) :: "memory");
actor_a->profiling->addFiring(ACTOR_ID::actor_a,
                              ACTION_ID::action_a,
                              (__clock_2 - __clock_1));
```

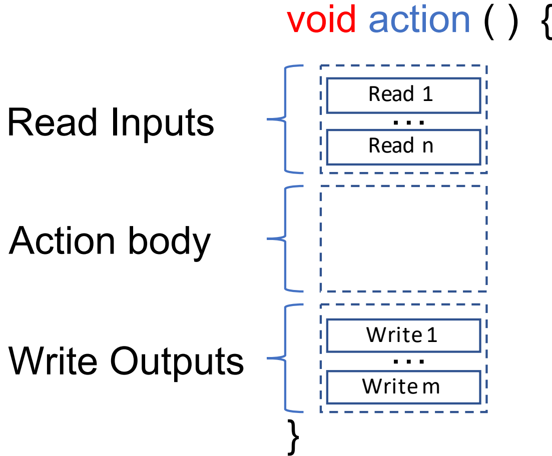


Figure 5.1: Illustration of the breakdown of an action's generated code: inputs are read sequentially, followed by the action body executing the processing, and concluding with the writing of outputs.

5.2.2 Static Heterogeneous Estimation

This section outlines the methodology for obtaining the profiling metrics generated from the method described in the previous section. The first challenge to address is how to handle the mismatch in frequency between the GPU and CPU clock rates. To overcome this, the clock rate of the hardware used for each measurement was added to the *xml* file. This information is used as an input to the TURNUS post-processor to normalize all measurements. To obtain the CPU clock cycle rate, the *RDTSC* register is sampled after a fixed interval and the resulting value is used to calculate the clock cycle rate. The GPU frequency is disclosed directly by *NVIDIA* via the CUDA API.

Another challenge to be addressed is the frequency variability. Despite the various terms used

such as *dynamic clocking*, *boost*, *step speed*, or *turbo boost*, various systems that cause clock rate volatility have been developed for each hardware. To ensure more accurate performance estimation, these technologies were disabled during the sampling process.

The static profiling method involves selecting and configuring the network for the application software during profiling. To ensure the generation of instrumented software, it is necessary to identify the design settings that needed to be provided to the Exelixi CUDA backend.

To fully explore the different combinations of FIFO buffers and actor platform assignments, four unique configurations are devised to estimate the runtime for any given configuration, regardless of the software program's complexity or the number of design options. These four design points are depicted in Figure 5.2.

Examples one and two involve actors assigned solely to either the CPU or GPU, respectively. These combinations allow us to measure the computation time of the action body and the scheduling time on each type of hardware platform. However, considering only these two design points would not be adequate as it wouldn't account for the data transmission time across the two platforms, such as the HostFifo (n°2 in blue) connecting the CPU and GPU. Instead, only the CPU-to-CPU transmission (Fifo n°1 in red) and GPU-to-GPU transmission (CudaFifo n°3 in purple) would be profiled.

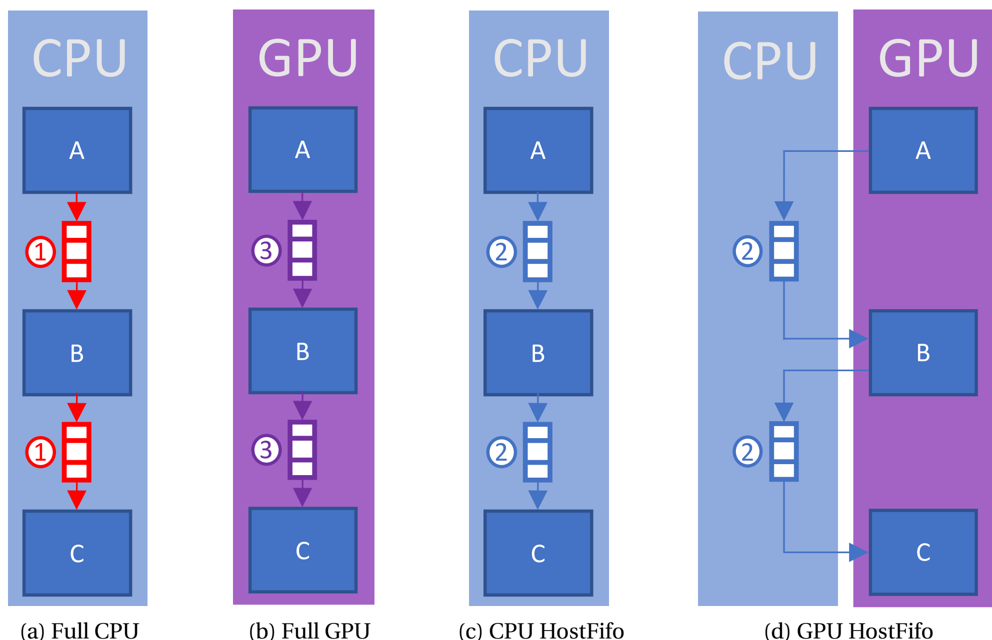
To address this issue, two additional profiling steps are taken by using a compiler option specifically developed to ensure the HostFifo (i.e., the cross-platform data transmission FIFO) is always produced, thus allowing us to obtain the data transmission time for all possible design points.

The performance of the developed methodology is evaluated using two application software programs. The aim is to determine the accuracy of the application runtime estimation, enabling efficient exploration of the design space. The runtime estimation is based on the abstract model of execution generated by the RVC-CAL construction within the TURNUS framework, which is executed using the software implementation and profiling results produced by the Exelixi CUDA backend. The backend is extended and improved as described in this section to ensure accurate results.

5.2.2.1 RVC-CAL JPEG Decoder

These experiments use the same JPEG decoder as described in Section 4.3.4.2. The focus of the first set of results is on the configuration with one partition, but with different input image definitions, quality factors, and FIFO buffer sizes. The reference design consists of the *Display* and *Src* actors in a single sequential partition assigned to the CPU, while the remaining actors are assigned to the highly parallel GPU.

For each configuration, an execution trace graph (ETG) is extracted using TURNUS. The performance weights are obtained by running the profiled binary, compiled from the code



(a) Full CPU (b) Full GPU (c) CPU HostFifo (d) GPU HostFifo

Figure 5.2: Illustration of the four static configurations required during profiling.

generated by the Exelixi CUDA backend, for each unique set of input stimuli. The TURNUS ETG post-processor is used to estimate the execution time of the application program. This estimated time is compared with the actual total time measured.

The results are shown in Figure 5.3 for height-distinct images and two different buffer sizes (512 and 1024 tokens). The different resolutions and quality factors of the input images used for the experiments are summarized in Table 5.1. The measures are normalized to a range of $[0,1]$, and it can be seen that the maximum deviation in the estimation is around 6.00%. This demonstrates that the performance estimation can be done with sufficient accuracy, regardless of the FIFO buffer sizes and inputs used.

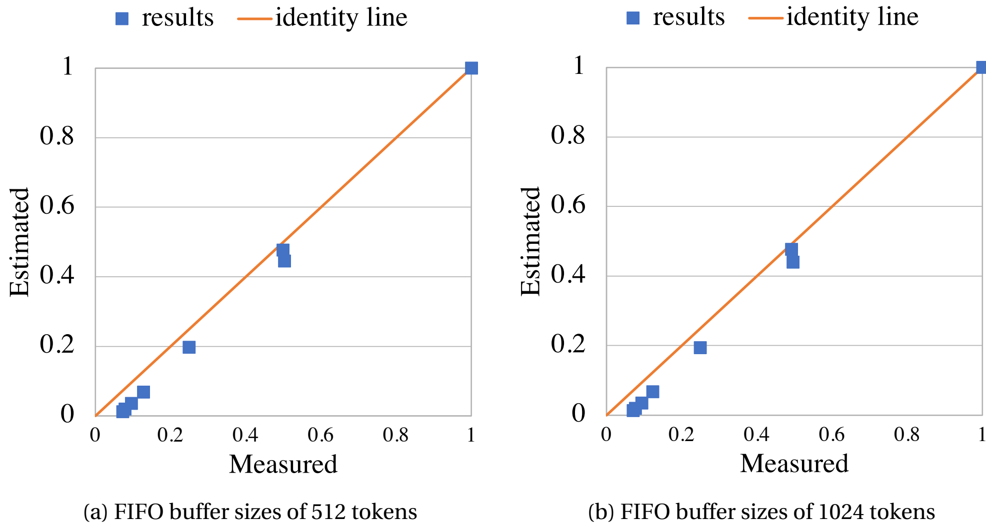
As for the second set of results, they use the same temporal stimulus and mapping configurations while comparing the estimated total time to the measured one. Unlike the first set of results, the same image is used as the input and thus, a single ETG is extracted. As previously explained, only four different configurations are sufficient for generating the weights, and each TURNUS estimation is executed with the appropriate combination of these weight files. Figure 5.4 shows the comprehensive results, with the 16 possible partitions. The partitions represent an arbitrary assignment of each actor to either the highly parallel GPU side or the CPU sequential side. The results are normalized to a range of $[0,1]$ and it can be seen that the maximum deviation of estimation is about 26.5From the graph, it can be observed that the mappings with the least precise estimated results are the partitions that result in the slowest runtime (top-right), which would not be the desired partitions for the design exploration process. This may be due to the fact that the number of FIFO buffers at the boundary between

the GPU and CPU is higher compared to other mapping configurations. The modeling for this type of FIFO buffer is optimistic and does not take into account factors such as memory bus access conflict and congestion, leading to less precise results.

Significantly, the results demonstrate that the performance trends, either improvement or deterioration, can be easily identified using the estimated time, without the need for measuring new weights or extracting new ETGs. This capability to detect performance trends is the minimum requirement for TURNUS to effectively explore the design space and identify optimal configurations. The purpose of the qualitative evaluation is to ensure that mapping configurations can be ranked and the best ones can be selected automatically, without the need for a time-consuming and resource-intensive evaluation on the hardware platform, which involves synthesizing, generating, compiling, and measuring each design point on the actual heterogeneous hardware by hand.

Resolution	4096x2240	2048x1536	4096x2240	2048x1536
QF	90	90	50	50
Resolution	1920×1080	1280×720	640×480	512x512
QF	55	65	80	75

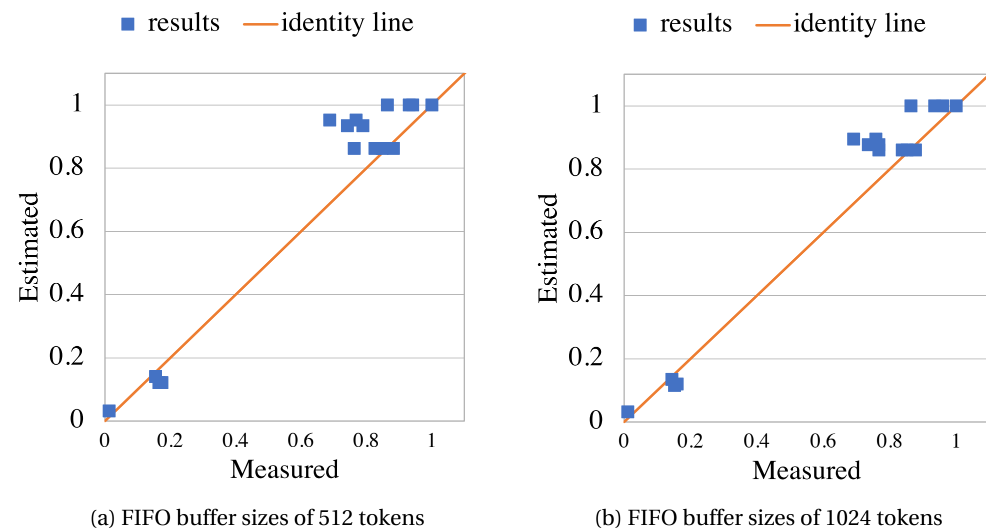
Table 5.1: The different resolutions and quality factors of the input images used for the JPEG Decoder results.



(a) FIFO buffer sizes of 512 tokens

(b) FIFO buffer sizes of 1024 tokens

Figure 5.3: Normalized comparison between the estimated and measured total runtime of the JPEG application with height inputs and two FIFO buffer configurations (512 and 1024 tokens). The identity line in orange corresponds to the 1:1 line for visual reference.



(a) FIFO buffer sizes of 512 tokens (b) FIFO buffer sizes of 1024 tokens

Figure 5.4: Normalized comparison between the estimated and measured total runtime of the JPEG application with all 16 possible partitions and two FIFO buffer configurations (512 and 1024 tokens). The identity line in orange corresponds to the 1:1 line for visual reference.

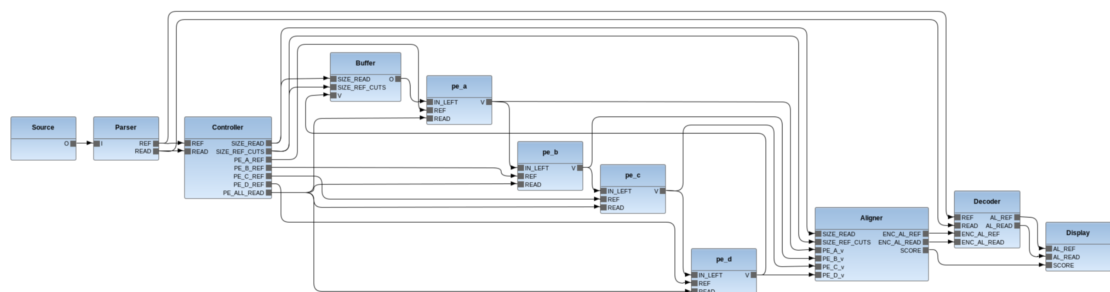


Figure 5.5: Graphical representation for the Smith-Waterman aligner application in RVC-CAL.

5.2.2.2 RVC-CAL Smith-Waterman Aligner

Figure 5.5 shows the graphical representation of the network of the Smith-Waterman (S-W) aligner software application presented in [96]. The S-W aligner performs a local alignment of two sequences, such as protein, RNA, or DNA sequences. The first sequence, $A = \{a_1, a_2, \ldots, a_n\}$, is typically referred to as the reference and the second sequence, $B = \{b_1, b_2, \ldots, b_m\}$, as the query or read. The S-W aligner consists of two processing steps: computation of a cost matrix and backtracking from the highest value in the matrix. The backtrack path compute the alignment (in terms of matches, mismatches, insertions, and deletions) between the query and the reference input sequences. The RVC-CAL implementation used in this article consists of eleven interconnected actors, including four PE actors that are responsible for computing the matrix scores and an *Aligner* actor that performs the backtracking path. Except for the *Source* actor, which is responsible for reading the input files and must run on the CPU, all other actors can be mapped to either the GPU or CPU partitions.

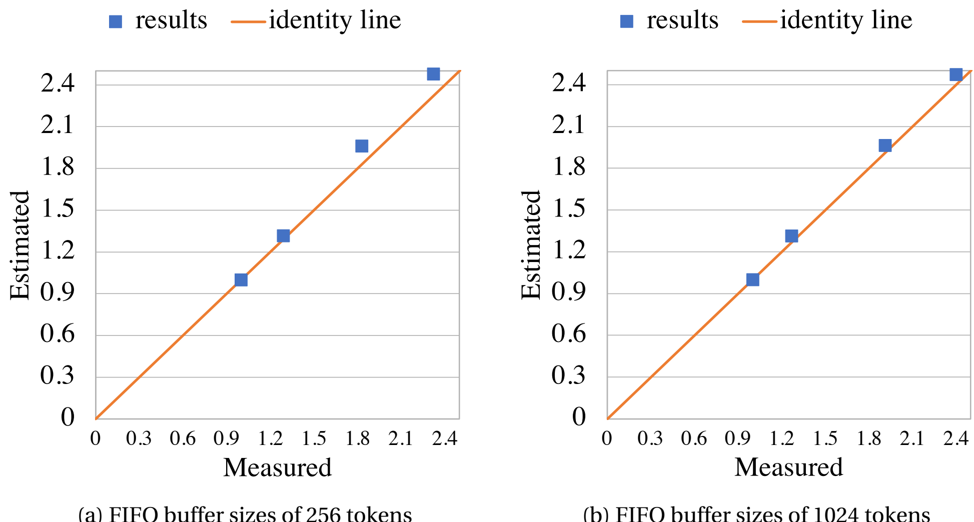
Like in the case of the JPEG decoder software program, two sets of experiments are conducted for the Smith-Waterman (S-W) aligner software. The first set of experiments examines a single partition configuration with different input sets and FIFO buffer sizes. The input sets are named $l_m_l_n$, where l_m is the length of the query sequence and l_n is the length of the reference sequence. All sequences are generated from human DNA data. In this experiment, the reference design consists of the *Source* actor mapped to the single sequential partition assigned to the CPU side, and the other actors assigned to the highly parallel GPU side. A similar approach is used for the results presented in Figure 5.6, where four different inputs (150_250, 150_200, 100_250, and 100_200) and two buffer sizes (1024 and 256 tokens) are tested. The measures are normalized to a range of $[0, 1]$ and it can be observed that the maximum deviation of estimation is about 15.6%. This result indicates that the performance can be estimated with sufficient accuracy, regardless of the FIFO buffer size or input used.

The second set of results in Figure 5.7 focuses on the exhaustive testing of all 1024 possible mapping configurations for different buffer sizes. This emphasizes the importance of evaluating the impact of different mappings on performance. The partitions represent the arbitrary selection of assigning each actor to either the highly parallel GPU side or the sequential CPU side. The measures are normalized to a range of $[0, 1]$, and it is observed that the maximum divergence of estimation is about 22.7%.

It can be noted that the mappings with the highest deviation in estimation correspond to the slowest runtimes and would not be desirable for the design exploration process. This is likely due to a higher number of FIFO buffers at the boundary between the GPU and CPU compared to other mapping configurations, as previously reported in the investigation of the JPEG decoder. However, the results show that the tendencies of performance deterioration or improvement can still be clearly identified by using the estimated runtime obtained through the TURNUS framework.

5.2.3 SIMD Parallel Estimation

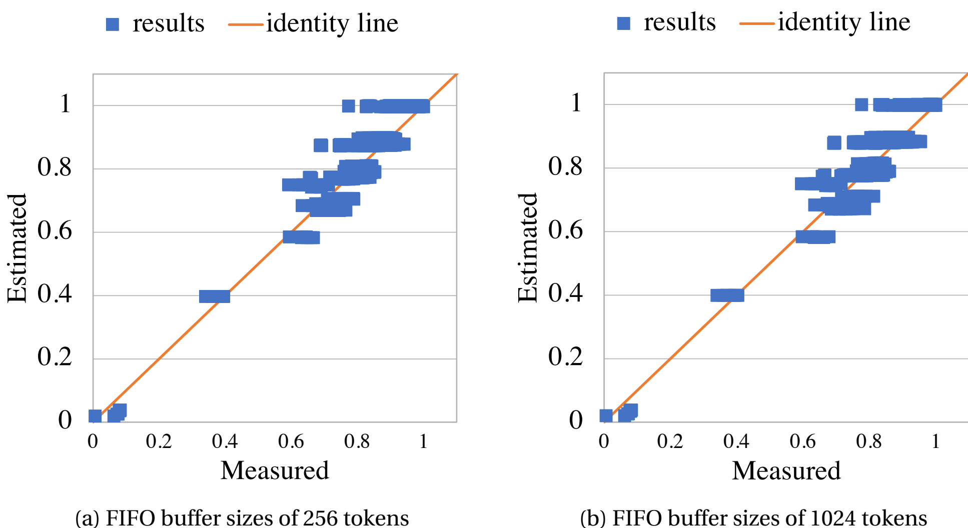
The GPU actor methodology has been extended with the use of SIMD parallelization for action execution, as described in Section 4.4.1. To generate the performance weight accurately in these cases, the following modifications were made. Listing 5.2 shows the SASS assembly code used to access the performance counter for clock measurement during SIMD parallel action execution. The first factor to consider is adding an if-statement based on the thread index (*thIdx*) to ensure that only one thread reports the elapsed time for the entire thread batch, avoiding race contention for the shared *profiling* weight collection. The second factor is to provide the number of threads in terms of th and bl, which correspond to the number of CUDA thread blocks and the number of threads per block, respectively.



(a) FIFO buffer sizes of 256 tokens

(b) FIFO buffer sizes of 1024 tokens

Figure 5.6: Normalized comparison between the estimated and measured total runtime of the Smith-Waterman aligner application with four inputs and two FIFO buffer configurations (256 and 1024 tokens). The identity line in orange corresponds to the 1:1 line for visual reference.



(a) FIFO buffer sizes of 256 tokens

(b) FIFO buffer sizes of 1024 tokens

Figure 5.7: Normalized comparison between the estimated and measured total runtime of the Smith-Waterman aligner application with all 1024 possible partitions and two FIFO buffer configurations (256 and 1024 tokens). The identity line in orange corresponds to the 1:1 line for visual reference.

Listing 5.2: Simplified example of the utilization of NVIDIA's assembly language (i.e. *SASS*) was required for reading the GPU's stable autoincrementing register in parallel mode.

```
// -- PROFILE: START
if (thIdx == 0) {
  asm volatile("mov.u64 %0, %%clock64;" : "=l"(__clock_1)::"memory");
}

// section of code that needs profiling ....

// -- PROFILE: STOP
if (thIdx == 0) {
  asm volatile("mov.u64 %0, %%clock64;" : "=l"(__clock_2)::"memory");
  actor_a->profiling
         ->addFiring(ACTOR_ID::actor_a,
                     ACTION_ID::action_a,
                     (__clock_2 - __clock_1),
                     actor_a->paction_a.th * actor_a->paction_a.bl);
}
```

5.2.3.1 RVC-CAL IDCT

In this section, the performance evaluation of the IDCT application program is presented. The program consists of three actors, all of which are mapped to the GPU. The focus of the evaluation is on the effect of varying the parallelization degree of the actions' execution, by using different combinations of thread blocks and threads per block, ranging from 1 block of 32 threads to 16 blocks of 256 threads each.

A single execution time graph (ETG) is generated using the TURNUS framework, and for each configuration, the profiled executable generated by the CUDA backend is executed with a different kernel launch configuration. The total estimated execution time is calculated using the TURNUS ETG post-processor and compared with the actual measured total time. Figure 5.8 displays the simulation results of the design with 8 different SIMD sizes. The values are normalized with respect to the first configuration, and it can be observed that the maximum estimation error is approximately 16%. More notably, the trend of performance improvement or deterioration can be clearly seen.

5.2.4 Dynamic Methodology Estimation

In this section, the dynamic network methodology is used to estimate the performance of a specific partition configuration. The performance weights are generated by injecting instrumented code into both the regular actors and the complementary shadow actors to measure the elapsed time for communication, execution, and scheduling. The main advantage of this

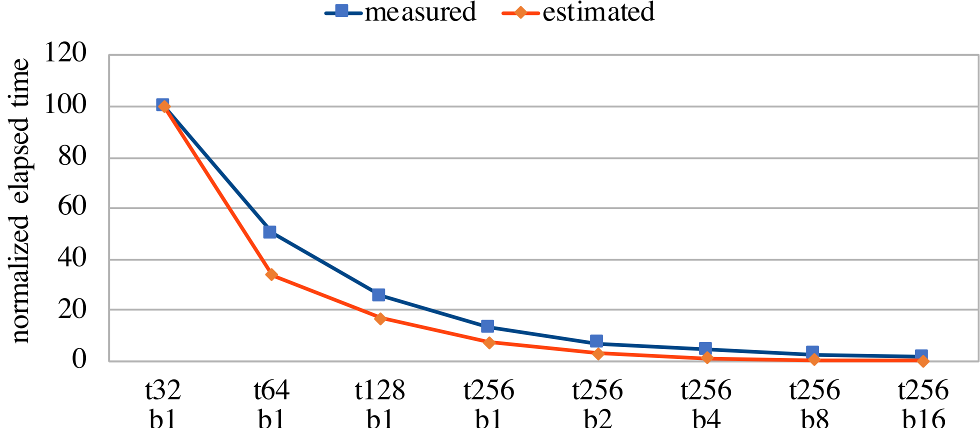


Figure 5.8: Comparison between the measured and estimated normalized total elapsed execution time with varying numbers of SIMD threads when executing the IDCT application.

method over the previous one is that the action selection of each actor can be executed in any configuration specified in the input *XML* files, allowing for the generation of performance weights in the exact configuration with all the same resource contention and utilization as in actual execution of the application program.

The parallel profiling process brings new implementation challenges. To avoid memory contention among actors reporting their measures, each actor creates its own *profiling* object. For GPU actors, where actions can be executed in parallel, *cuda::atomic* variables are used to prevent data race. Finally, at the end of the profiling, all reported data is merged for computation and generation of the report files.

This newly developed technique enhances the accuracy of the full design space exploration methodology. As shown in Figure 5.9, an updated representation of Figure 3.6, TURNUS design space exploration tool can now provide configuration files to the compiled binary to generate updated performance weights as shown in Figure 5.10. These updated weights can be used by the performance estimation tool to complete the optimization cycle. This tighter loop not only offers a faster exploration of the design space, as the time required for code regeneration, compilation, and hardware testing is automated, but it also ensures that the generated weights reflect the actual hardware usage.

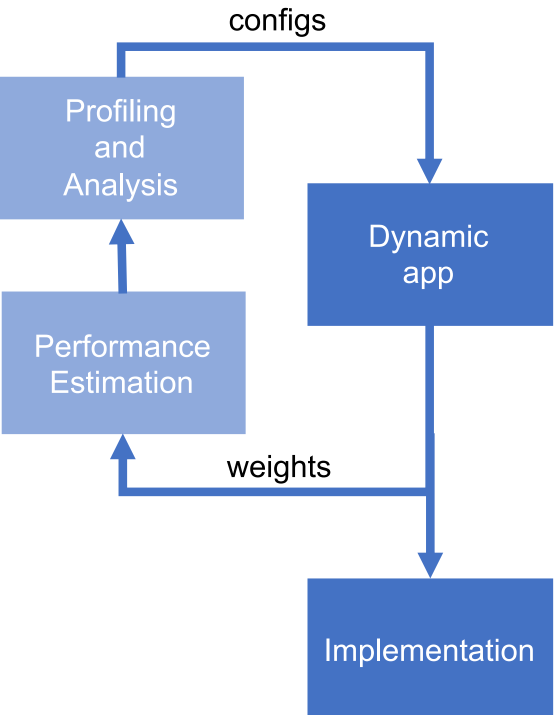


Figure 5.9: Design flow with dynamic networks. Deep blue elements represent the implementation path of the tool, while light blue the profiling and design space exploration path of the tool.

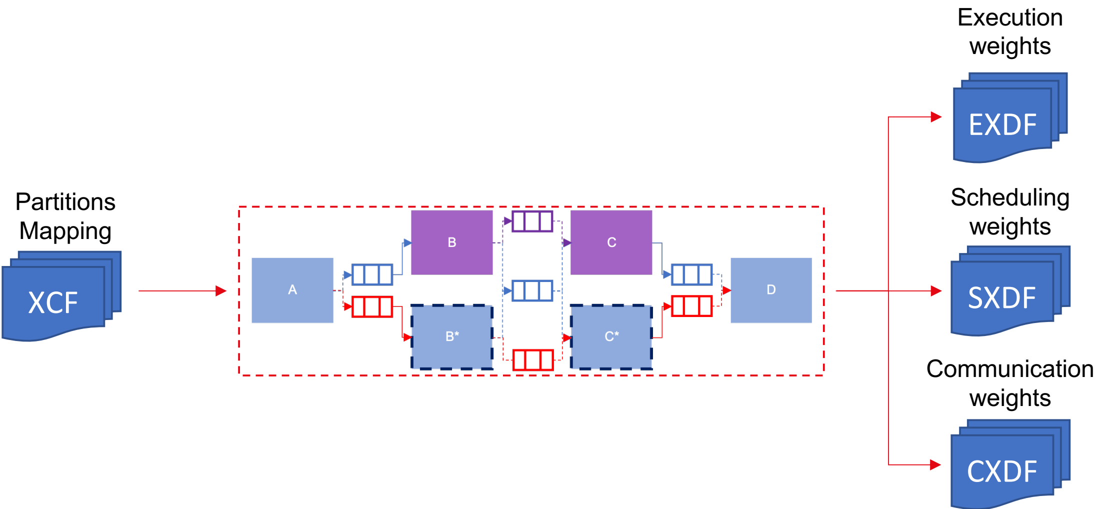


Figure 5.10: Dynamic heterogeneous actor methodology injected with instrumentation code so as to generate performance weights.

5.3 Design Space Exploration

Design space exploration tools are an important part of the design process. As previously mentioned, there are many design points that can be tuned in the process of creating a concrete implementation from a high-level dataflow representation, such as partitioning, mapping, and buffer dimensions. To quickly find an efficient set of these parameters that meet certain performance requirements, automation tools are necessary. This section, will describe the improvement and adaptation of the work done in [97] to the CPU/GPU model of this thesis. It will demonstrate how the performance estimation developed in the previous section makes Tabu search a good candidate as a meta-algorithm for generating efficient partitions and mappings using both the high-level dataflow representation and the actual implementation.

5.3.1 Tabu Search

The Tabu Search (TS), introduced by Glover [98], is a meta-heuristic optimization algorithm that can be applied to solve a wide range of optimization problems. TS involves iteratively exploring the solution space, making moves to solutions that are deemed "better" than the current one based on the given evaluation function. In the context of this study, the evaluation function refers to the minimal overall runtime of the application that needs to be optimized.

The core algorithm incorporates an explicit memory structure, known as the Tabu list, to prevent revisiting previously explored solutions and promote the exploration of the solution space, thereby avoiding getting stuck in local optima. This list, known as the Tabu list, can be adjusted in size and content to balance between exploration and exploitation in the search, and to ensure good overall performance in terms of quality of results, speed, robustness, simplicity and flexibility. It can be applied to problems with continuous or discrete variables, multiple objectives and is often used when the search space is large or the optimization problem is difficult to solve with other methods. In order to adapt TS to a specific problem, the representation of solutions, the neighborhood structure, the Tabu list structure, and the stopping criterion must be designed.

The TS meta-algorithm is applied in the context of partitioning actors across CPU and GPU partitions in order to find a solution that is defined as a map of actors and processing units (sequential CPU partitions or the parallel GPU partition). The number of actors is fixed by the dataflow application model, and the initial partition provides the upper bounds for the number of CPU and GPU partitions that can be used in a valid solution. It is not required that all partitions be used (meaning, no actor is be mapped to them), and allowing empty partitions as valid solutions allows for the discovery of optimal solutions that may not use all hardware resources if it results in better performance. It is important to note that not all actors are necessarily compatible with being run on the GPU, so the initial partition provided to the TS meta-algorithm should assign actors that can be mapped to either the CPU or GPU to the GPU partition, and actors that can only run on the CPU to a CPU partition.

To move from a valid solution to a neighboring one, the following types of moves are possible:

- REINSERT: move an actor from one partition to another.
- SWAP: exchanging the partitions of two actors that are currently assigned to different partitions. This can be done by moving each actor to the other actor's partition.

These basic moves can be combined in various ways to create different exploration strategies. Some examples of these strategies are discussed in Section 5.3.2.

TS can tuned using a couple of parameters a, b, t_{ab}, ϵ and T. When an actor, j, is transferred from one partition, ρ, to another, it is prohibited from returning to ρ for a certain number of iterations, t_{ab}. This number, t_{ab}, is a randomly chosen integer from the range $[a, b]$, and in previous experiments outlined in [97], the values of a and b were set to 5 and 15, respectively. Smaller values of t_{ab} do not allow for escape from local optima, while larger values do not allow for intensification of the search around promising solutions. There are two other adjustable parameters in the TS algorithm: ϵ (the proportion of neighboring solutions explored during each iteration) and T (the time limit). If the time limit T is reached, the search will be immediately terminated and the best solution found thus far will be returned. With a fixed T, a small value of ϵ results in more iterations being performed but fewer neighbors being examined in each iteration, resulting in more diversification. A large value of ϵ, on the other hand, plays an intensification role (more solutions around the current one are explored). Finally, a small (large) value of t_{ab} strengthens the intensification (diversification) ability of the search, respectively.

Figure 5.11 shows the generic design flow for using TS for design space exploration in the context of this work. The optimization process begins with a user-provided initial mapping and execution trace, and generates a tentative new mapping solution. The initial mapping and execution trace provided by the user can have a big impact on the efficacy of the solution. Regarding the execution trace the input stimulus provided to the application during the trace generation need to be a good representation of the general input space so that the conclusion drawn would be applicable to the performance of the application in production. Regarding the initial mapping, since the size of the design space is too big to feasibly be fully explore, starting from an initially good partition is very important to lead to best possible results using this methodology. This new suggested solution is then evaluated to determine its performance and analyzed to provide the meta-algorithm with sufficient information to continue exploring the design space.

5.3.2 Neighborhood Move Generator

Below is a list of neighborhood move generators used in this work during the Tabu search. They are mainly implemented using REINSERT moves. They can be used on their own during a design space exploration loop, or multiple of them can be combined.

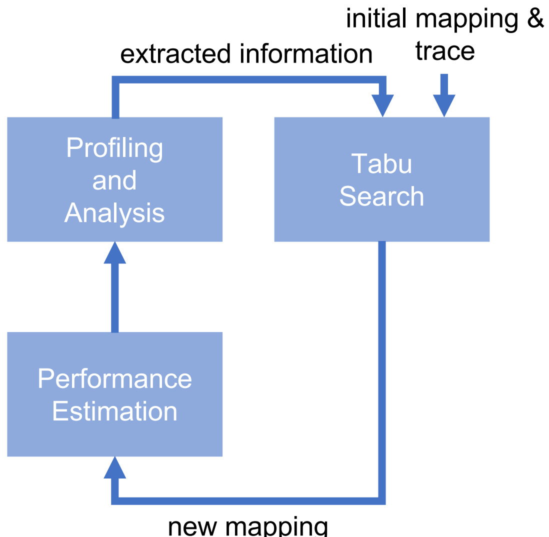


Figure 5.11: Generic design flow when doing design space exploration using Tabu search.

- Balancing ($N^{(B)}$): move randomly an actor from the partition with the least idle time and move it to the partition with the most idle time. The partition idle time is defined as the time frame during which no actor are executing due to constraint such as dependency or others.

- Idle ($N^{(I)}$): move consecutively each actor whose idle time is greater than its processing time to the most idle partition that is different from the current partition of that actor. The idle time of an actor is defined as the time frame when it could execute according to the satisfaction of its firing rules, but it has to wait to be scheduled because another actor in the same partition is currently executing.

- Communication frequency ($N^{(CF)}$): if an actor has a higher communication frequency (i.e., more token transfers) with actors in a given partition than with the ones in its current partition, move the actor to that partition.

- Random ($N^{(R)}$): choose randomly an actor and move it to a different partition also randomly chosen.

5.3.3 Design Point Evaluations

Three different DSE optimization loops have been developed and used for comparison in this work. Each of them uses a different design point evaluation methodology that matches the one presented in Section 5.2. These are the Static, Dynamic, and Measured methodologies and are presented in the following subsections.

5.3.3.1 Static Evaluation

In this version of the TS design space exploration, the static heterogeneous estimation described in Section 5.2.2 is used to evaluate the performance of the proposed new mapping configuration.

Figure 5.12 illustrates this methodology variant. First, the initial mapping is provided, along with the execution trace and an heterogeneous set of weights (as described in Section 5.2.2) that have been generated using that initial mapping on the actual hardware platform using the profiled application generated by the Exelixi CUDA backend. This information is then used to suggest a new mapping, which is fed to the 'Weights Updater'. The appropriate weights from the heterogeneous set of weights are selected based on the new mapping, meaning that the appropriate set of weights will be used depending on whether an actor is being mapped to a CPU or GPU partition. The same applies for the four sets of communication weights (CPU-CPU, CPU-GPU, GPU-CPU, GPU-GPU) based on the platform the actor it is communicating to is assigned. Finally, the updated weights are fed to the performance estimation engine, and the model analysis is conducted to extract information that helps with the neighboring move generation, after which the optimization loop can start again.

This methodology does not necessitate continuous access to the hardware platform as only the four static configurations need to be evaluated beforehand. During the design space exploration loop, only post-mortem access to the weights and execution trace is needed. This methodology places no restrictions on the move generators that can be used and any of the Neighborhood move generators defined in Section 5.3.2 or others can be used. However, it should be noted that this solution is not the most precise in terms of performance evaluation and may be relatively slow due to the use of the TURNUS performance estimation engine.

5.3.3.2 Dynamic Evaluation

In this version of the Tabu search design space exploration, the dynamic heterogeneous estimation described in Section 5.2.4 is used to evaluate the performance of the proposed new mapping configuration.

Figure 5.13 depicts this methodology variant. First, the initial mapping and the execution trace is provided by the user to the Tabu search meta-algorithm. In addition, a compiled version of the application using the profiled dynamic network methodology and generated by the Exelixi CUDA backend is made available to the optimization loop and is depicted in dark blue on the schema. During each iteration of the loop, the new mapping configuration is fed to the dynamic app, which is executed with this new mapping and generates new performance weights directly on the hardware platform. These new weights are then integrated into the execution trace and used by the TURNUS performance estimation engine to estimate the performance and provide insight for the analysis, allowing for the generation of neighboring moves.

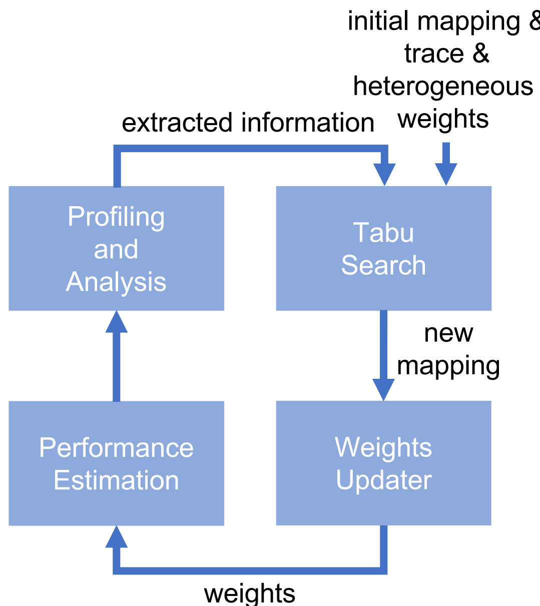


Figure 5.12: Design flow when doing design space exploration using TS and the static evaluation strategy.

This methodology requires continuous access to the hardware platform, as each new candidate mapping configuration needs to be evaluated by executing the profiled application. This methodology has no restrictions on the move generators that can be used; all the Neighborhood move generators defined in Section 5.3.2 and more can be used. This solution is relatively precise compared to the static methodology. However, it is the slowest due to the combination of full execution on the hardware platform with profiling weight generation and the TURNUS performance estimation engine.

5.3.3.3 Measured Evaluation

In this version of the Tabu search design space exploration, no estimation methodology is used, instead the performance are directly measure using the actual application on the hardware platform.

Figure 5.14 depicts this methodology variant. First, the initial mapping and the execution trace is provided by the user to the TS meta-algorithm. In addition, a compiled version of the application using the dynamic network methodology and generated by the Exelixi CUDA backend is made available to the optimization loop and is depicted in dark blue on the schema. During each iteration of the loop, the new mapping configuration is fed to the dynamic app, which is executed with this new mapping directly on the hardware platform and the total execution time is measured. This information is then used directly to generate neighboring moves.

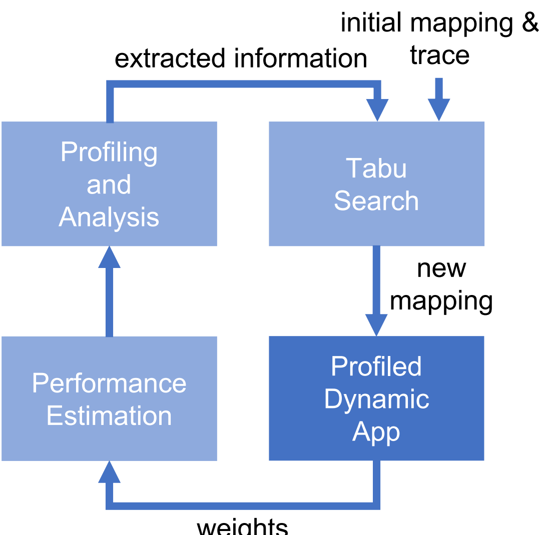


Figure 5.13: Design flow when doing design space exploration using TS and the dynamic evaluation strategy.

This methodology requires continuous access to the hardware platform, as each new candidate mapping configuration needs to be evaluated by executing the dynamic application. This solution is the most precise, as the evaluation takes place in a setting that is quasi-identical to the 'production execution' and is also the fastest, as no post-mortem analysis is required and the performance evaluation time is as fast as the final execution time. However, not all the Neighborhood move generators defined in Section 5.3.2 can be used. In particular, generators that use insight from the performance estimation engine to suggest a new mapping cannot be used. This is the case for the Balancing and Idle generators, which use the Idle metrics provided by the analysis.

5.3.4 Results

A simple program example has been selected to evaluate the methodologies introduced in this section. The goal is to determine how the different neighborhood move generators and design point evaluations affect the results obtained when exploring the design space using Tabu search. This example also illustrates how the method can be applied to optimize any particular application.

5.3.4.1 IDCT Example

The synthetic application used to demonstrate this technique is derived from the SIMD IDCT application used in Section 4.4.1.2. Figure 5.15 illustrates the top-level representation of the dataflow network that comprises a chain of five IDCT actors that compute the inverse discrete cosine transform on their input. This chain of actors is a representation of an intensive

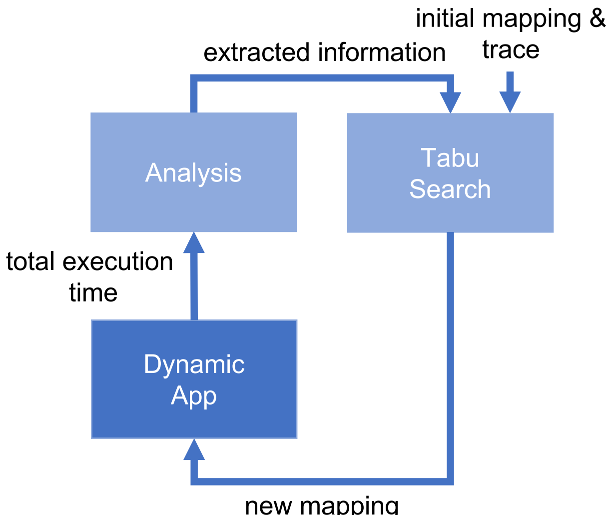


Figure 5.14: Design flow when doing design space exploration using TS and the measured evaluation strategy.

computation that could be found in a real-world application.

All results have been generated allowing the solution to use a single CPU core and the parallel GPU partition. The initial partition for all DSE runs is set to the sequential CPU partition, with all actors mapped to it. The value T has been set to 20 minutes as in the related work [97], meaning the algorithm halt after 20 minutes without discovering a better-performing configuration.

Table 5.2 presents the summary of experimental results obtained using three evaluation methods (Static, Dynamic, Measured) and four neighboring move generators (Balancing, Communication Frequency, Idle, Random). Additionally, two composite generators were used: Joint, which combines moves from the four move generators, and Prob, which selects moves from one of the four generators with a 25% probability initially, increasing over time if the generator yields better configurations. The third column in the table displays the number of configurations evaluated, while the fourth column shows the number of empty iterations, or the number of iterations where the move generator could not suggest a neighboring configuration from the current one. The "Time to best" column indicates the amount of time, in minutes, required to find the best-performing configuration among those tested, and the "Best Performance" column shows the overall time needed, in seconds, to run the program using the best configuration found by the TS algorithm. Table 5.3, shows the mapping of the bests/worst partitions between CPU and GPU and there corresponding performances.

The results show that the Measured evaluation methodology was able to find the best time for mapping all actors to the GPU. The Joint Generator was found to be the most efficient, as it found the best-performing configuration in under 3 minutes with the lowest number of total tries (205). In contrast, the Static and Dynamic methodologies were found to be too slow to

generate meaningful results within the given time limit, with one simulation taking around 18 minutes. This approach would likely require increasing significantly the T value. However, the insights offered by these two other methods are still valuable as shown in the related work cited in the previous section. This showed that to improve the results, it would be beneficial to investigate, in further research, combining the speed of the Measured evaluation methodology with the insights gained from the Static and Dynamic methods.

It should be noted that the limited information available for the Communication Frequency move generator is due to the fact that, starting from the initial configuration, it is unable to generate any neighboring configurations as the mapping is already balanced. However, this does not mean that this move generator is not useful.

It is also interesting to note that the Measured Joint method outperforms the Measured Random and Comm Freq methods. The Measured Joint method combines the benefits of different generators, which leads to a faster and more efficient search for good partitions. The Random generator is good for exploring the search space and avoiding getting stuck in a local minimum. On the other hand, other generators may converge faster to a good local partition. By combining these generators in the Measured Joint method, the algorithm can explore the search space efficiently and converge quickly to a good partition. Overall, the Measured Joint method seems to strike a good balance between exploration and exploitation of the search space, leading to improved performance compared to other methods.

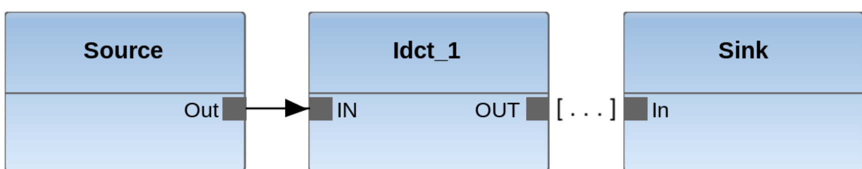


Figure 5.15: Illustration of the test RVC-CAL IDCT dataflow network.

Evaluation	Generator	# Iterations	# Empty Iterations	Time to Best [min]	Best Perf [sec]
Static	Balancing	2	0	x	1.35
	Comm Freq	1	x	x	1.35
	Idle	2	0	x	1.35
	Random	2	0	x	1.35
	Joint	2	0	x	1.35
	Prob	2	0	x	1.35
Dynamic	Balancing	2	0	20	5.6
	Comm Freq	1	x	x	1.35
	Idle	2	0	21	4.86
	Random	2	0	39	2.64
	Joint	2	0	22	4.66
	Prob	2	0	23	4.94
Measured	Comm Freq	1	x	x	1.35
	Random	212	77	5	1.17
	Joint	205	71	3	1.17
	Prob	3030426	3030376	5	1.17

Table 5.2: Summary table of results with three evaluation methods (Static, Dynamic, Measured), and four neighboring move generators.

	Partition		Perf [sec]
	CPU	GPU	
Best Measured	x	All actors	1.17
Best Dynamic/Static	All actors	x	1.35
Worst	All other actors	Idct_5	5.6

Table 5.3: Mapping of the bests/worst partitions between CPU and GPU and there corresponding performances.

5.4 Conclusion

The chapter presented a comprehensive end-to-end automated design space exploration of RVC-CAL dynamic dataflow programs on GPU architecture. It included both the strategy's design and its implementation. The chapter covered three crucial components.

The first section explains how an automated, clock-accurate profiling of the dataflow application is carried out on both the CPU and GPU platforms. This profiling generates performance weights that indicate the number of clock cycles that have elapsed during processing, scheduling, and communication, respectively.

In the second section, the actual automated heterogeneous performance estimation model is presented. This model enables the automatic evaluation of an application's overall runtime with a set of input stimuli and configuration parameters, such as partitioning and mapping in this case. Additionally, it allows for the automatic evaluation and comparison of two sets of configurations, which is an important step towards automated design space exploration. Two methodologies were presented in this section. The first methodology uses a static network, while the second employs dynamic, heterogeneous actors to instantiate the dataflow network. These methodologies were evaluated to demonstrate their accuracy by comparing the estimated time with the measured time on different standard applications with varying input sizes and buffer sizes.

In the final section, the actual design space exploration strategy is presented. It consists of a loop where, at each iteration, a new set of configuration is suggested by the algorithm. The performance of the application being optimized when using this new configuration is then evaluated and compared with the best configuration found so far. This process continues until a fixed optimization time set by the user is reached. This section presented six strategies to explore the design space: Balancing, Communication Frequency, Idle Random, Joint, Probabilistic, and three evaluation methodologies: static, dynamic, and measured. These different methods were illustrated using a simple application program to demonstrate their functionality.

6 Additional Experimental Results

6.1 Introduction

This chapter presents experimental results that serve to further illustrate the methodologies and tools developed in this thesis. By showcasing how these techniques can be applied across a range of applications and platforms, readers will gain a deeper understanding of the versatility and efficacy of the proposed methods. To this end, the experiments were conducted across three distinct hardware platforms, including a personal computer, a high-performance workstation, and an embedded platform. By using such a diverse range of hardware, the aim to demonstrate how the proposed methods can be effectively applied across a variety of use cases. The overarching goal of these experiments is to provide a comprehensive validation of the thesis work, bringing together all previous partial results and methodologies to form a cohesive and conclusive body of research.

6.2 Experimental Setup

The platforms used to conduct the experiments in this chapter have the following specifications:

- System 1: Intel Skylake I5-6600 CPU with 16 GB of DDR4 RAM, paired with a GeForce GTX 1660 SUPER NVIDIA GPU with 6 GB of memory.
- System 2: AMD Threadripper 3990X CPU with 256 GB of DDR4 RAM, paired with a GeForce RTX 3080 Ti NVIDIA GPU with 12 GB of memory.
- System 3: NVIDIA Jetson AGX Xavier embedded platform composed of a Carmel Armv8.2 8-core processor paired with an integrated 512 CUDA core Volta GPU and 32 GB of LPDDR4x memory.

All systems run CUDA version 12 for the GPU software library.

6.3 Experimental Applications

This Section presents the design used in the experiments. The application is implemented using the aforementioned RVC-CAL programing language.

6.3.1 HEVC Decoder

The latest video coding standard, known as HEVC (High Efficiency Video Coding) [99], is currently the leading technology for widespread application. This new standard boasts improved compression capabilities, surpassing the previous standard, AVC (Advanced Video Coding) [100, 101]. However, it comes with a tradeoff, as its complex design requires efficient HEVC codec implementations to accommodate video products approaching 4K (2160p) resolution.

The MPEG-RVC Framework [75] has standardized the specifications for a dataflow implementation of HEVC [102]. The implementation comprises 34 actors in its basic form, with its primary functional units corresponding to the algorithmic blocks of the HEVC standard decoder. These units include the bit-stream Parser, Motion Vector Prediction, Inter Prediction, Intra-Prediction, IDCT, Reconstruct Coding Unit, Deblocking Filter, Sample Adaptive Offset Filter and Decoding Picture Buffer. Figure 6.1 depicts the entire network with all its components.

6.4 Model Validation

The aim of this subsection is to demonstrate the validity of the model. For this purpose, the complex HEVC Decoder RVC-CAL application is being utilized. The decoding speed in images per second of the HEVC decoder, implemented in RVC-CAL and generated using the dynamic network feature of the Exelixi CUDA backend developed in this thesis, is shown in Figure 6.2. The results were obtained using the BQSquare benchmark, with QF22, a resolution of 416x240, and 60 FPS as input. The graph displays 33 different partitions, where in each partition, a single actor is mapped to the GPU and the remaining actors are assigned to a single CPU partition. The purpose of this test is to demonstrate that even with a highly complex standardized application, all actors can be mapped to the GPU, with the exception of the input (Source) and output (display) actors. Furthermore, regardless of the model of computation (MoC) that an actor uses, it can still be executed on the GPU. This test even shows that fully dynamic actors can run on the GPU and that the FIFOs are automatically generated accordingly.

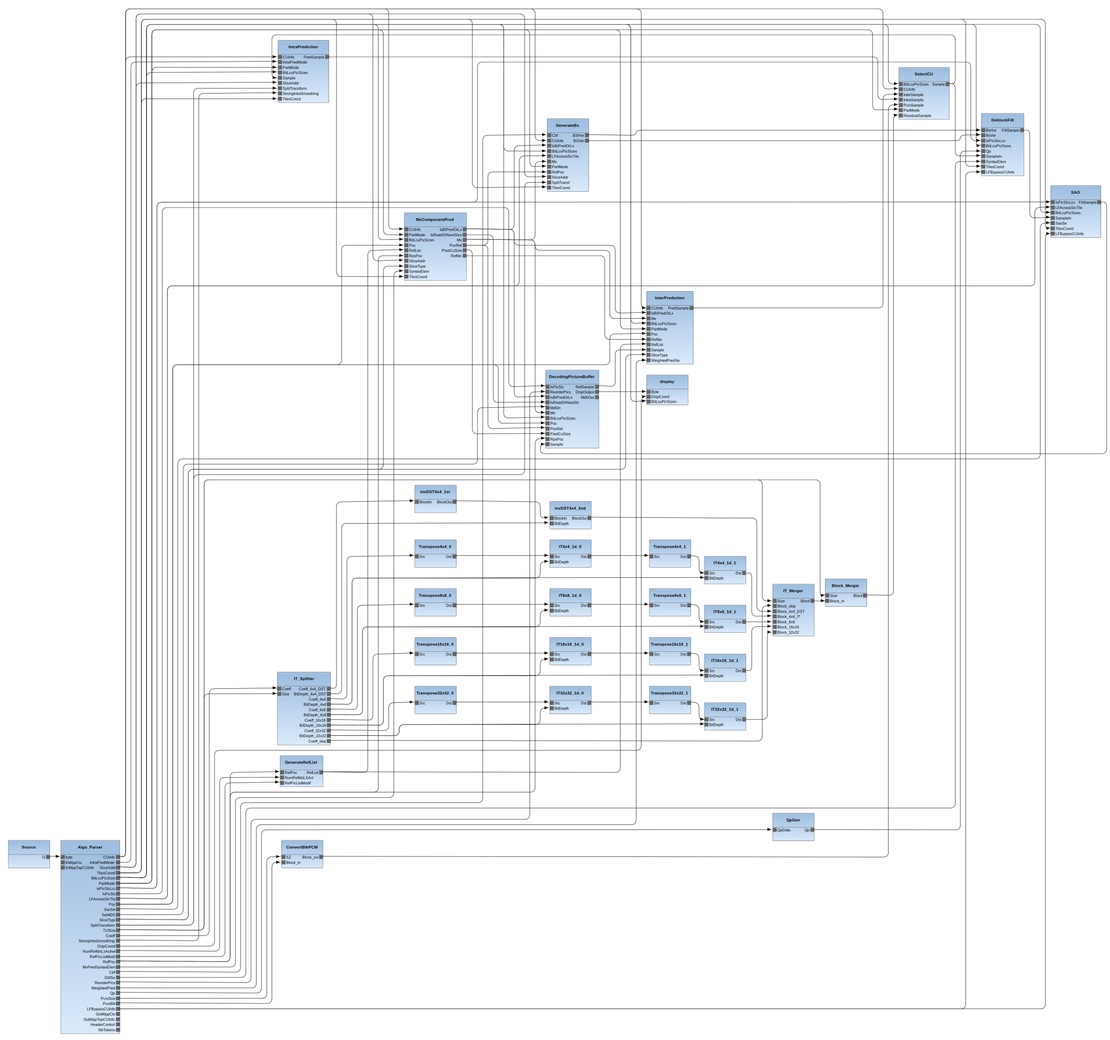


Figure 6.1: Illustration of the test RVC-CAL HEVC decoder dataflow network.

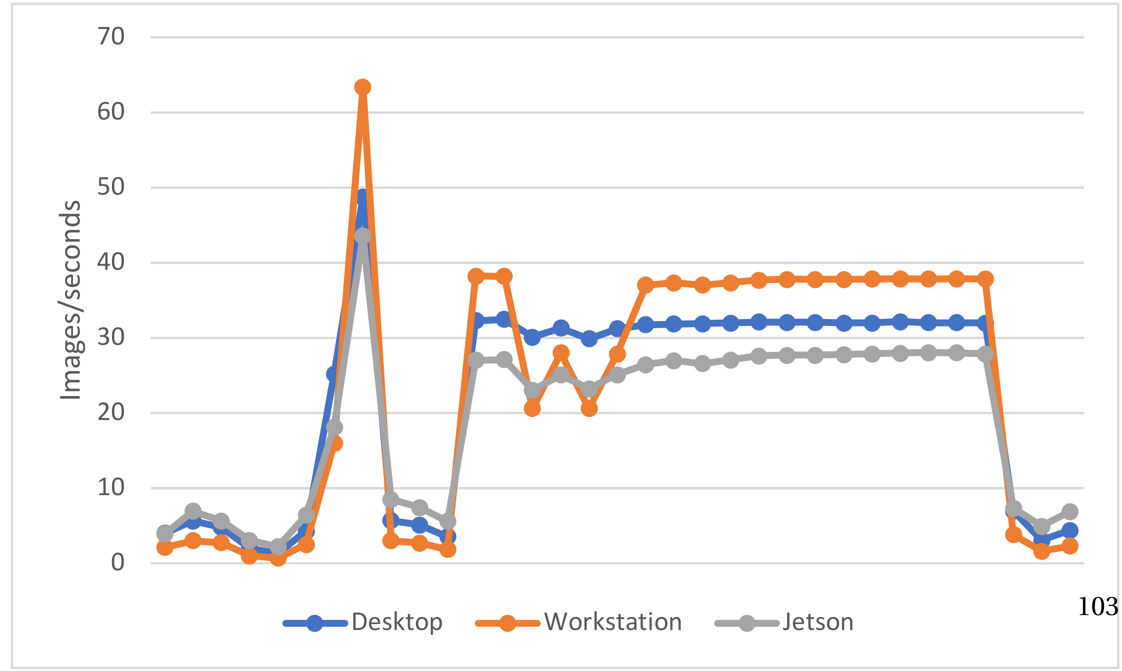


Figure 6.2: Performance comparison of the RVC-CAL HEVC decoder implementation across three hardware platforms for 33 different partitions, with each partition having a single different actor executing on the GPU and the remaining actors on the CPU.

6.5 Conclusion

In this chapter, the results aimed to demonstrate the integration of various optimization techniques through a single, complex application that represents a real-world, high-performance processing algorithm, thereby providing robust validation of the methodologies introduced. To achieve this, the HEVC decoder, one of the most complex standardized algorithms, was executed on three distinct platforms, encompassing all processing hardware families, including embedded systems, personal computing, and data centers. The outcomes displayed diverse partitioning configurations and effectively illustrated the key attributes of abstraction, concurrency, analyzability, modularity, and portability, as defined in the problem statement section.

7 Conclusions and Future Work

7.1 Conclusion

The field of heterogeneous computing is continually evolving, and it is crucial to optimize the performance of software and adapt the methodology to offer better utilization of the new hardware resources. The contributions presented in this thesis address this need by developing a high-level synthesis compiler infrastructure that targets CPU/GPU heterogeneous processing platforms and supports the full specification of the RVC-CAL dataflow programming language. The compiler infrastructure, based on the open-source ORCC compiler and the Exelixi CUDA backend, is an essential step towards optimizing software for heterogeneous computing platforms. It suggests a novel model for representing parallel dataflow computation on modern GPU co-processing platforms.

One of the significant contributions of this thesis is the implementation of a set of compiler optimizations within the newly designed Exelixi CUDA backend to enhance the performance and functionality of the generated low-level implementations. The first optimization utilizing Single Instruction Multiple Data (SIMD) parallelization techniques to speed up the runtime of dataflow actions, resulting in improved overall execution performance. This is accomplished through parallel execution of multiple instances of the same action using CUDA. The Second optimization utilizes the dynamic programming feature provided by the CUDA API to implement inter-action parallel execution techniques, enabling the parallel execution of different action inside the same actor if they do not have any internal state dependency. This techniques allows to speed up the runtime of dataflow actions, leading to improved overall execution performance. The third contribution is the introduction of a methodology for generating dynamic RVC-CAL networks. This technique allows for the specification of the partitioning and mapping at runtime during the application's startup process, resulting in better utilization of resources and improved performance. Finally, a dynamic SIMD parallelization optimization has been developed that involves generating multiple SIMD parallel executions of the same action. The number of threads used could change dynamically throughout the runtime of the

application, with the objective of maximizing performance and maximizing utilization of GPU resources.

Another significant contribution of this thesis is the improvement over the Exelixi CUDA backend for generating instrumented code for clock-accurate profiling and outputting the corresponding performance metrics, that can be used in the TURNUS post-processor to estimate the overall application execution time, accurately. The accuracy of the estimated execution time enables the development of automatic optimization tools.

Finally, the adaptation of the design space exploration tool TURNUS to the CPU/GPU model of this thesis is a significant contribution. The design space exploration tool is a powerful tool for exploring the vast design space of heterogeneous computing platforms. The tool enables the optimization of RVC-CAL dataflow software targeting CPU/GPU heterogeneous processing platform by automatically suggesting good performing partition and mapping configuration for better utilization of resources and improved performance.

These developed methodology have been demonstrated and evaluated through the presentation of well known RVC-CAL software implementations allowing to better reflect how a designer or software developer would benefit from the proposed methodologies.

The developed methodologies have been demonstrated and evaluated through the presentation of well-known RVC-CAL software implementations, enabling a better reflection of how a designer or software developer can benefit from the proposed methodologies. These implementations serve as concrete examples of how the developed methodologies can be applied and their efficacy in improving the performance and functionality of the RVC-CAL software when targeting CPU/GPU processing platform. The evaluation of the developed methodologies through these software implementations provides a comprehensive understanding of the impact of the proposed model and optimizations, such as the dynamic network generation, SIMD parallelization, inter-action parallelization techniques etc... Furthermore, they show what kind of benefit is to be gained from a model that can be easily ported on different platform that also provide automated way of tuning the low-level implementation to the available hardware resources. This assessment enables a thorough investigation of the benefits of the developed methodologies, ensuring that they are reliable and effective for practical use.

7.2 Future Work

The proposed design flow has demonstrated its efficiency for programming CPU/GPU heterogeneous systems and providing methodologies for profiling and optimizing performance.

However, there are still open research and engineering questions that need to be addressed in the future. This section will provide an overview of the suggested research directions, which can be decomposed into the two main axes of this thesis. The first axis focuses on the development of the model and code generation. This includes new developments on both the ORCC and Exelixi CUDA backend sides, such as providing new features, optimizations, analysis methods, and performance improvements. The second axis focuses on the evaluation and tuning of the configuration of the parameters that drive the implementation of the high-level representation of RVC-CAL applications. This leads to new suggestions for development for the TURNUS design space exploration framework. Overall, while the proposed design flow has shown its effectiveness, there is still room for improvement and further research in order to address open questions and enhance performance.

7.2.1 High-Level Synthesis of RVC-CAL Dataflow Programs

This section presents ideas and research directions for improving the high-level synthesis of RVC-CAL dataflow programs for heterogeneous CPU/GPU systems.

7.2.1.1 Dataflow Compiler Extensions

The first proposed enhancement is to implement automatic detection of actors that can take advantage of parallel execution of actions, using the compiler's intermediate representation. Currently, actors must be manually identified by the developer through the use of annotations. To achieve this improvement, a dependency analysis that examines state variables and scheduling is required.

The current model presented in this thesis represents the GPU hardware as a single, fully parallel partition, where each actor assigned to it is designed to run in full parallel mode. To achieve this, each actor is represented as its own long-running kernel. It would be beneficial to explore the possibility of representing the GPU hardware as a set of parallel sequential partitions, similar to multicore CPUs, where each partition runs in parallel but the actors inside each partition run in sequence. This approach could potentially allow for a higher number of smaller actors to be mapped to the GPU without them all competing for parallel resources at the same time. However, the major challenge in this approach is that, unlike CPUs, there is no direct control over the mapping between partitions and hardware resources (cores) in GPUs through GPU APIs. This creates the need for a methodology to evaluate the optimal number of parallel partitions for a specific GPU architecture. The DSE tools extended in the research could be used to solve this issue.

The experiments and models in this thesis focus on CPU/GPU heterogeneous systems. Previous studies have demonstrated that the methodology and tools developed are capable of

supporting other computing platforms, such as FPGAs or many-core processors. Further research should aim to combine these techniques to enable dataflow applications to execute on any combination of these hardware processing platforms. While this is technically feasible, in practice, the user interface of the framework must be updated to facilitate cross-backend actor partitioning and mapping. Additionally, data communication between the different hardware platforms must be optimized.

Only a single GPU and multicore CPU were used to evaluate the design model in this thesis, despite the absence of any technological restrictions against supporting multiple GPUs in the same system. Future developments should assess the performance of cross-GPU data communication by utilizing the current FIFO implementation, and evaluate the need for adding a dedicated GPU-to-GPU implementation. The user interface of the framework and configuration files should be updated to allow for assigning actors to different GPUs. Finally, the Exelixi CUDA backend should implement proper hardware detection and assignment of actors to the correct GPU in the runtime library generated by the Exelixi CUDA backend by utilizing available CUDA APIs.

7.2.1.2 Features for Performance Improvement

In its current state, the software developer annotates actions that can benefit from parallelization on a GPU and provides a single set of parameters, including the number of CUDA blocks and threads per block, for each eligible action. To improve the software, the option for multiple sets of these parameters could be added, allowing for a configuration that better fits the hardware platform or a particular point of the execution to be used.

One of the major drawbacks of heterogeneous systems is the large amount of data that must be transferred between platforms constantly. To optimize performance, data movement optimization is a crucial design consideration. One way that further research could enhance performance is by providing data prefetching between platforms.

Several novel techniques have been developed in this work to enhance parallelism, including SIMD and inter-action parallelization. Another promising approach to leveraging the massive number of threads available on GPUs is the use of "repeat" or "for comprehension" constructs. These constructs allow for the efficient expression of repetitive loops that can be parallelized, provided that the inner code does not contain any dependencies. To achieve this, the compiler's intermediate representation must be analyzed, and the necessary transformations must be implemented in the Exelixi CUDA backend.

7.2.1.3 Auto-Tuning Programs

In CPU/GPU heterogeneous systems, performance can vary greatly even between systems from the same generation and vendor. Thanks to advancements in GPU APIs, an interesting research direction would be to develop self-optimizing compiled implementations from the high-level RVC-CAL representation. This would involve performing live analysis and profiling during application execution to dynamically modify parameters, such as increasing or decreasing SIMD parallelization, changing the level of intra-action parallelization, or even moving actors between partitions or hardware platforms. Several components required to achieve this goal have been developed. This including the dynamic heterogeneous actor methodology, which allows binaries to contain both CPU and GPU versions of an actor and decides which one to execute at runtime. The dynamic SIMD parallelization capability that allows for the dynamic configuration of the number of parallel SIMD threads used during action execution while maintaining the computational model intact. Additionally, dynamic heterogeneous actors can be generated with instrumented code, allowing for profiling metrics to be extracted. To fully realize the goal presented, a runtime controller needs to be created that will analyze current performance metrics and make configuration changes using the dynamic capabilities to try to improve the overall performance of the application on the specific platform it is running on.

7.2.1.4 Profiled Application Generation

In the proposed profiling methodology outlined in this thesis, the issue of the volatile frequency of the processing platforms was addressed by manually disabling it. However, a potential avenue for further research would be to enhance the profiling process while maintaining the dynamic frequency changes, as this more accurately represents the platform's usage in a production environment. This could be achieved by periodically sampling the frequency while the application is executing and using these samples for normalization, or by comparing the results of two measurements, one with the dynamic frequency enabled and one with it disabled, to estimate the degree of inaccuracies.

7.2.2 Design space exploration

There is much scope for further research in the context of design space exploration for CPU/GPU heterogeneous systems. This thesis has only addressed the partitioning and mapping aspect, and there is potential for research on scheduling, critical path detection, actor fusion, and more. Two suggestions related to the proposed work are as follows: Firstly, an interface could be created for the TURNUS Tabu-Search methodology, allowing for the manipulation of the number of SIMD parallelizations. This would help in finding the most performing parameters in terms of CUDA blocks and CUDA threads per block. Secondly, it is worth exploring the idea of mixing evaluation methodologies (static, dynamic, and measured) into a single tool flow. By leveraging their differences, a similar approach to the PROB and

JOINT methodology that mixes the neighborhood move generator could be utilized.

Bibliography

[1] “NVIDIA Jetson Nano is a tiny AI computer for $99 and up,” https://liliputing.com/nvidia-jetson-nano-is-a-tiny-ai-computer-for-99-and-up, online, accessed February. 2023.

[2] “The Qualcomm QCA4020 SoC, a multi-mode system-on-chip,” https://www.qualcomm.com/products/technology/wi-fi/qca4020, online, accessed February. 2023.

[3] “The Apple M1 Max SoC architecture,” https://seekingalpha.com/article/4460844-apple-m1-promax-silicon-steals-the-show, online, accessed February. 2023.

[4] A. Caulfield, E. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, “A cloud-scale acceleration architecture,” in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, October 2016. [Online]. Available: https://www.microsoft.com/en-us/research/publication/configurable-cloud-acceleration/

[5] “NVIDIA Grace Hopper Superchip Architecture,” https://nvdam.widen.net/s/qjzrmfdn2j/nvidia-grace-hopper-superchip-architecture-whitepaper-v1.0, online, accessed February. 2023.

[6] “Exploring the GPU Architecture,” https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#from-graphics-processing-to-general-purpose-parallel-compu online, accessed February. 2023.

[7] “Exploring the NVidia GPU Architecture,” https://doi.org/10.1371/journal.pone.0061892.g001, online, accessed February. 2023.

[8] “NVIDIA CUDA Compute Unified Device Architecture,” https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, online, accessed February. 2023.

[9] A. Bloch, E. Bezati, and M. Mattavelli, “Programming Heterogeneous CPU-GPU Systems by High-Level Dataflow Synthesis,” in *2020 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE, 2020, pp. 1–6.

[10] A. Bloch, S. Casale-Brunet, and M. Mattavelli, “Methodologies for synthesizing and analyzing dynamic dataflow programs in heterogeneous systems for edge computing,” *IEEE Open Journal of Circuits and Systems*, vol. 2, pp. 769–781, 2021.

[11] A. Bloch, E. Bezati, and M. Mattavelli, “Composite data types in dynamic dataflow languages as copyless memory sharing mechanism,” in *International Conference on Computational Science*. Springer, 2019, pp. 717–724.

[12] A. Bloch, S. C. Brunet, and M. Mattavelli, “Simd parallel execution on gpu from high-level dataflow synthesis,” in *2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 2021, pp. 62–68.

[13] A. Bloch, S. Casale-Brunet, and M. Mattavelli, “Inter-actions parallel execution on gpu from high-level dataflow synthesis,” in *2021 55th Asilomar Conference on Signals, Systems, and Computers*. IEEE, 2021, pp. 1151–1155.

[14] ——, “Performance estimation of high-level dataflow program on heterogeneous platforms by dynamic network execution,” *Journal of Low Power Electronics and Applications*, vol. 12, no. 3, p. 36, 2022.

[15] ——, “Dynamic simd parallel execution on gpu from high-level dataflow synthesis,” *Journal of Low Power Electronics and Applications*, vol. 12, no. 3, p. 40, 2022.

[16] A. Bloch, S. C. Brunet, and M. Mattavelli, “Performance estimation of high-level dataflow program on heterogeneous platforms,” in *2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 2021, pp. 69–76.

[17] A. Bloch, S. Casale-Brunet, and M. Mattavelli, “Design space exploration for partitioning dataflow program on cpu-gpu heterogeneous system,” *Journal of Signal Processing Systems*, (Manuscript submitted for publication.

[18] “Microsoft ARM,” https://www.microsoft.com/en-us/surface/business/surface-pro-x/processor, online, accessed February. 2023.

[19] “NVIDIA Grace,” https://nvidianews.nvidia.com/news/nvidia-introduces-grace-cpu-superchip, online, accessed February. 2023.

[20] “Apple M1,” https://www.apple.com/newsroom/2020/11/apple-unleashes-m1, online, accessed February. 2023.

[21] J. E. Stone, D. Gohara, and G. Shi, “Opencl: A parallel programming standard for heterogeneous computing systems,” *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.

[22] R. Reyes and V. Lomüller, “Sycl: Single-source c++ accelerator programming,” in *Parallel Computing: On the Road to Exascale*. IOS Press, 2016, pp. 673–682.

[23] B. Ashbaugh, A. Bader, J. Brodman, J. Hammond, M. Kinsner, J. Pennycook, R. Schulz, and J. Sewall, "Data parallel c++ enhancing sycl through extensions for productivity and performance," in *Proceedings of the International Workshop on OpenCL*, 2020, pp. 1–2.

[24] R. Farber, *Parallel programming with OpenACC.* Newnes, 2016.

[25] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP.* Morgan kaufmann, 2001.

[26] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[27] "Hip: C++ heterogeneous-compute interface for portability," https://github.com/ROCm-Developer-Tools/HIP, 2017, online, accessed February. 2023.

[28] W. Lund, S. Kanur, J. Ersfolk, L. Tsiopoulos, J. Lilius, J. Haldin, and U. Falk, "Execution of dataflow process networks on opencl platforms," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing.* IEEE, 2015, pp. 618–625.

[29] J. Boutellier and T. Nylanden, "Programming graphics processing units in the rvc-cal dataflow language," in *2015 IEEE Workshop on Signal Processing Systems (SiPS).* IEEE, 2015, pp. 1–6.

[30] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele, "Scenario-based design flow for mapping streaming applications onto on-chip many-core systems," in *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, 2012, pp. 71–80.

[31] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[32] O. Rafique, F. Krebs, and K. Schneider, "Generating Efficient Parallel Code from the RVC-CAL Dataflow Language," in *2019 22nd Euromicro Conference on Digital System Design (DSD).* IEEE, 2019, pp. 182–189.

[33] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing.* IEEE, 2013, pp. 1299–1308.

[34] L. Schor, A. Tretter, T. Scherer, and L. Thiele, "Exploiting the parallelism of heterogeneous systems using dataflow graphs on top of opencl," in *The 11th IEEE Symposium on Embedded Systems for Real-time Multimedia*, 2013, pp. 41–50.

[35] S. Lin, Y. Liu, W. Plishker, and S. S. Bhattacharyya, "A Design Framework for Mapping Vectorized Synchronous Dataflow Graphs onto CPU-GPU Platforms," in *Proceedings of*

the 19th International Workshop on Software and Compilers for Embedded Systems, ser. SCOPES '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 20–29. [Online]. Available: https://doi.org/10.1145/2906363.2906374

[36] A. Sbîrlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar, "Mapping a data-flow programming model onto heterogeneous platforms," *ACM SIGPLAN Notices*, vol. 47, no. 5, pp. 61–70, 2012.

[37] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[38] "CAL design suite," http://sourceforge.net/projects/caldesignsuite/, online, accessed February. 2023.

[39] C. Lucarz, "Dataflow Programming for Systems Design Space Exploration for Multicore Platforms," Ph.D. dissertation, EPFL - STI - EDIC, Lausanne, 2011.

[40] H. Yviquel, J. Boutellier, M. Raulet, and E. Casseau, "Automated design of networks of transport-triggered architecture processors using dynamic dataflow programs," *Signal Processing: Image Communication*, vol. 28, no. 10, pp. 1295 – 1302, 2013.

[41] "Daedalus: System-Level Design For Multi-Processor System-on-Chip," http://daedalus.liacs.nl, online, accessed February. 2023.

[42] M. Thompson, H. Nikolov, T. Stefanov, A. Pimentel, C. Erbas, S. Polstra, and E. Deprettere, "A Framework for Rapid System-level Exploration, Synthesis, and Programming of Multimedia MP-SoCs," in *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '07. New York, NY, USA: ACM, 2007, pp. 9–14.

[43] H. Nikolov, T. Stefanov, and E. Deprettere, "Systematic and Automated Multiprocessor System Design, Programming, and Implementation," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 3, pp. 542–555, Mar. 2008.

[44] J. Ceng, J. Castrillon, W. Sheng, H. Scharwachter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda, "MAPS: an integrated framework for MPSoC application parallelization," in *Proceedings of the 45th annual Design Automation Conference*. ACM, 2008, pp. 754–759.

[45] J. Castrillon, R. Velasquez, A. Stulova, W. Sheng, J. Ceng, R. Leupers, G. Ascheid, and H. Meyr, "Trace-based KPN composability analysis for mapping simultaneous applications to MPSoC platforms," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, Mar. 2010, pp. 753–758.

[46] R. Leupers and J. Castrillon, "MPSoC programming using the MAPS compiler," in *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, Jan. 2010, pp. 897–902.

[47] J. Castrillon, R. Leupers, and G. Ascheid, "Maps: Mapping concurrent dataflow applications to heterogeneous MPSoC," *Industrial Informatics, IEEE Transactions on*, vol. 9, no. 1, pp. 527–545, 2013.

[48] A. Mihal, C. Kulkarni, M. Moskewicz, M. Tsai, N. Shah, S. Weber, Y. Jin, K. Keutzer, C. Sauer, K. Vissers, and S. Malik, "Developing Architectural Platforms: A Disciplined Approach," *IEEE Des. Test*, vol. 19, no. 6, pp. 6–16, Nov. 2002.

[49] M. Gries and K. Keutzer, *Building ASIPs: The Mescal Methodology*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[50] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: an integrated electronic system design environment," *Computer*, vol. 36, no. 4, pp. 45–52, Apr. 2003.

[51] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y. Joo, "PeaCE: A Hardware-software Codesign Environment for Multimedia Embedded Systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 3, pp. 1–25, May 2008.

[52] "PREESM: the parallel and real-time embedded executives scheduling method," http://sourceforge.net/projects/preesm/, online, accessed February. 2023.

[53] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J. Nezan, and S. Aridhi, "Preesm: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming," in *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*. IEEE, 2014, pp. 36–40.

[54] "Ptolemy project: heterogeneous modeling and design," http://ptolemy.eecs.berkeley.edu, online, accessed February. 2023.

[55] E. Lee, "Overview of The Ptolemy Project," Electronics Research Laboratory, University of California at Berkeley, Technical Memo UCB/ERL M98/71, Nov. 1998.

[56] S. Stuijk, M. Geilen, and T. Basten, "SDF3: SDF for free," in *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, Jun. 2006, pp. 276–278.

[57] A. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *Computers, IEEE Transactions on*, vol. 55, no. 2, pp. 99–112, Feb. 2006.

[58] "Space Codesign Systems," http://www.spacecodesign.com, online, accessed February. 2023.

[59] J. Chevalier, M. de Nanclas, L. Filion, O. Benny, M. Rondonneau, G. Bois, and E. Aboulhamid, “A SystemC refinement methodology for embedded software,” *Design Test of Computers, IEEE*, vol. 23, no. 2, pp. 148–158, Mar. 2006.

[60] B. Gedik, H. Andrade, K. Wu, P. Yu, and M. Doo, “SPADE: The System S Declarative Stream Processing Engine,” in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’08. New York, NY, USA: ACM, 2008, pp. 1123–1134. [Online]. Available: http://doi.acm.org/10.1145/1376616.1376729

[61] W. De Pauw, M. Leţia, B. Gedik, H. Andrade, A. Frenkiel, M. Pfeifer, and D. Sow, “Visual Debugging for Stream Processing Applications,” in *Runtime Verification*, ser. Lecture Notes in Computer Science, H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, Eds. Springer Berlin Heidelberg, 2010, vol. 6418, pp. 18–35.

[62] “SynDEx,” http://www.syndex.org, online, accessed February. 2023.

[63] “SystemCoDesigner,” http://www.mycodesign.com/research/scd, online, accessed February. 2023.

[64] C. Haubelt, M. Meredith, T. Schlichter, and J. Keinert, “SystemCoDesigner: Automatic Design Space Exploration and Rapid Prototyping from Behavioral Models,” in *Proceedings of the 45th Design Automation Conference (DAC0́8)*, Anaheim, CA, USA., Jun. 2008, pp. 580–585.

[65] J. Keinert, M. Streubuhr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith, “SystemCoDesigner: an Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 1, pp. 1–23, Jan. 2009.

[66] C. Bolchini, S. Cherubin, G. C. Durelli, S. Libutti, A. Miele, and M. D. Santambrogio, “A runtime controller for opencl applications on heterogeneous system architectures,” *ACM SIGBED Review*, vol. 15, no. 1, pp. 29–35, 2018.

[67] G. Kahn, “The semantics of a simple language for parallel programming,” in *Information processing*, J. L. Rosenfeld, Ed. Stockholm, Sweden: North Holland, Amsterdam, Aug. 1974, pp. 471–475.

[68] E. Lee and T. Parks, “Dataflow Process Networks,” in *Proceedings of the IEEE*, 1995, pp. 773–799.

[69] A. Grabowski, “Scott-continuous functions,” *Journal of Formalized Mathematics*, vol. 10, 1998.

[70] D. McAllester, P. Panangaden, and V. Shanbhogue, “Nonexpressibility of Fairness and Signaling,” *J. Comput. Syst. Sci.*, vol. 47, no. 2, pp. 287–321, Oct. 1993. [Online]. Available: http://dx.doi.org/10.1016/0022-0000(93)90034-T

[71] J. Dennis, "First Version of a Data Flow Procedure Language," in *Programming Symposium, Proceedings Colloque Sur La Programmation*. London, UK: Springer-Verlag, 1974, pp. 362–376. [Online]. Available: http://dl.acm.org/citation.cfm?id=647323.721501

[72] E. Lee and E. Matsikoudis, "A Denotational Semantics for Dataflow with Firing," in *Memorandum UCB/ERL M97/3, Electronics Research*, 1997.

[73] J. Janneck, "Actor Machines: A machine model for dataflow actors and its applications," Lund University, Computer Science Department, Technical Memo LTH Report 96, 2011 (corrections 2013-03-01), Mar. 2013.

[74] J. Eker and J. Janneck, "Cal language report," Tech. Rep. ERL Technical Memo UCB/ERL, Tech. Rep., 2003.

[75] I. 23001-4:2011, "Information technology - MPEG systems technologies - Part 4: Codec configuration representation," 2011.

[76] K. Jerbi, D. Renzi, D. de Saint-Jorre, H. Yviquel, M. Raulet, C. Alberti, and M. Mattavelli, "Development and optimization of high level dataflow programs: the HEVC decoder design case," in *48th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, USA, Nov. 2014.

[77] A. A.-H. Ab Rahman, S. Casale Brunet, C. Alberti, and M. Mattavelli, "Dataflow program analysis and refactoring techniques for design space exploration: Mpeg-4 avc/h.264 decoder implementation case study," in *2013 Conference on Design and Architectures for Signal and Image Processing*, 2013, pp. 63–70.

[78] D. de Saint Jorre, C. Alberti, M. Mattavelli, and S. Casale-Brunet, "Exploring mpeg hevc decoder parallelism for the efficient porting onto many-core platforms," in *2014 IEEE International Conference on Image Processing (ICIP)*, 2014, pp. 2115–2119.

[79] "Caltoopia," https://github.com/Caltoopia, online, accessed February. 2023.

[80] G. Cedersjö and J. W. Janneck, "Tÿcho: a framework for compiling stream programs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 6, pp. 1–25, 2019.

[81] E. Gebrewahid, "Tools to compile dataflow programs for manycores," Ph.D. dissertation, Halmstad University Press, 2017.

[82] S. Savas, Z. Ul-Abdin, and T. Nordström, "A framework to generate domain-specific manycore architectures from dataflow programs," *Microprocessors and microsystems*, vol. 72, p. 102908, 2020.

[83] J. Boutellier and A. Ghazi, "Multicore execution of dynamic dataflow programs on the distributed application layer," in *2015 IEEE global conference on signal and information processing (GlobalSIP)*. IEEE, 2015, pp. 893–897.

[84] E. Bezati, M. Emami, J. Janneck, and J. Larus, “Streamblocks: A compiler for heterogeneous dataflow computing (technical report),” 2021. [Online]. Available: https://arxiv.org/abs/2107.09333

[85] “Orcc source code repositoy,” http://github.com/orcc/orcc, online, accessed February. 2023.

[86] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet, “Orcc: Multimedia Development Made Easy,” in *Proceedings of the 21st ACM International Conference on Multimedia*, ser. MM ’13. ACM, 2013, pp. 863–866.

[87] J. desRivieres and J. Wiegand, “Eclipse: A platform for integrating development tools,” *IBM Systems Journal*, vol. 43, no. 2, pp. 371–383, 2004.

[88] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend.* Packt Publishing Ltd, 2016.

[89] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.

[90] “TURNUS source code repositoy,” http://github.com/turnus, online, accessed February. 2023.

[91] S. Casale-Brunet, “Analysis and optimization of dynamic dataflow programs,” Ph.D. dissertation, EPFL STI, Lausanne, 2015.

[92] M. Michalska, S. Casale-Brunet, E. Bezati, and M. Mattavelli, “High-precision performance estimation for the design space exploration of dynamic dataflow programs,” *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, no. 2, pp. 127–140, 2017.

[93] “Orcc-Apps source code repositoy,” https://github.com/orcc/orc-apps, online, accessed February. 2023.

[94] “CAL Exelixi Backends source code repositoy,” https://bitbucket.org/exelixi/exelixi-backends, online, accessed February. 2023.

[95] M. Michalska, S. Casale-Brunet, E. Bezati, and M. Mattavelli, “High-precision performance estimation of dynamic dataflow programs,” in *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC).* Ieee, 2016, pp. 101–108.

[96] S. Casale-Brunet, E. Bezati, and M. Mattavelli, “High level synthesis of Smith-Waterman dataflow implementations,” in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* Ieee, 2017, pp. 1173–1177.

[97] M. Michalska, N. Zufferey, and M. Mattavelli, “Tabu search for partitioning dynamic dataflow programs,” *Procedia Computer Science*, vol. 80, pp. 1577–1588, 2016.

[98] F. Glover, "Tabu search—part i," *ORSA Journal on computing*, vol. 1, no. 3, pp. 190–206, 1989.

[99] "Plastics – Determination of fracture toughness – Linear elastic fracture mechanics (LEFM) approach," International Organization for Standardization, Geneva, CH, Standard, 2013.

[100] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the h. 264/avc video coding standard," *IEEE Transactions on circuits and systems for video technology*, vol. 13, no. 7, pp. 560–576, 2003.

[101] F. Bossen, B. Bross, K. Suhring, and D. Flynn, "Hevc complexity and implementation analysis," *IEEE Transactions on circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1685–1696, 2012.

[102] D. J. De Saint Jorre, D. Renzi, S. Casale Brunet, M. Michalska, E. Bezati, and M. Mattavelli, "Mpeg high efficient video coding stream programming and many-cores scalability," Tech. Rep., 2014.

Aurelien Bloch

bloch.aurelien@gmail.com · linkedin.com/in/aurelienbloch

- Heterogeneous systems co-design & Parallel computing
- Compiler & Programming languages

WORK EXPERIENCE

September 2018 – March 2023

Doctoral researcher, EPFL

- Research:
  Compilation and Design Space Exploration of Dataflow Programs for Heterogeneous CPU-GPU Platforms. Development of the Exelixi CUDA compiler backend generating C++/CUDA platform-specific code from high-level dataflow synthesis to target CPU/GPU co-processing platform.
- Teaching:
  - Project oriented programming C++ (2020 - 2022)
  - Computer Architecture (2019 – 2020)

September 2017 – August 2018

Technical Student, CERN

- Knowledge acquisition of LuaJIT internal working system by studying and documenting this innovative and scarcely reported just-in-time compiler.
- Industrial quality documentation, testing, and optimization for the MAD-NG (Methodical Accelerator Design Next Generation) application used to design and simulate particle accelerators at CERN. Development of the plotting framework in Lua.

Summer 2016

Software Engineer Intern, KALRAY

Implement software for instrumentation of applications on Kalray's MPPA processor platform to automatically generate the application's call graph annotated with performance information using advance Makefile and C/C++ code injection.

EDUCATION

EPFL, École Polytechnique Federale de Lausanne

- 2018-2023, PhD. Electrical Engineering
- 2015-2018, Master. Computer Science
  - Specialty in Computer Engineering
- 2012-2015, Bachelor. Computer Science

TECHNICAL SKILLS

- **Programming**: C, C++, CUDA, VHDL, Java, Lua, Scala, Ruby, Python, LaTeX, SQL, Bash
- **Environment:** Git, JetBrains, Eclipse, Android Studio, Quartus, Modelsim, R, MATLAB
- **System**: OSX, Linux, Windows

OPENSOURCE PROJECTS

- **2018 – 2023:** Open RVC-CAL Compiler (https://github.com/orcc)
  The Open RVC-CAL Compiler (ORCC) is a dataflow compiler capable of generating optimized low-level code for various heterogeneous platforms starting from a high-level, platform agnostic, program representation. This tool was also used in MPEG to standardize the AVC and HEVC video standards.
- **2018 – 2023:** Exelixi (https://github.com/exelixi)
  A CAL Dataflow Compiler for High-Level Synthesis for heterogeneous platform such as the Xilinx platforms or Nvidia GPUs platforms. It is an extension of ORCC that allows the user to implement low-level code for state-of-the-art FPGA and GPU platforms. It has been mainly used to implement and accelerate bioinformatics algorithms.
- **2018 - 2023**: TURNUS (https://github.com/turnus)
  A design space exploration and optimization framework for dynamic dataflow programs. This framework allows the analysis of dynamic dataflow programs (the most complex to be analyzed) through a representation of their execution in the form of a graph. This graph (which can contain billion nodes) is post-processed using a set of heuristics that allow to identify the critical points of an application, which are the close to optimal mappings in the target heterogeneous platforms, and how to reduce the use of resources (e.g., energy, memory) preserving the performance of a dataflow program. This tool was also used in MPEG to standardize the AVC and HEVC video standards.

ACADEMIC PROJECTS

- **Profiling of dynamic dataflow programs on embedded multi-core ARM architectures**
  Implementation of a software tool, fully integrated within an existing dataflow compiler, that automatically generates low-level instrumented source code in C/C++/Assembly for the target hardware platform.
- **Multithreaded Scale-out Processors: A Performance and Area Analysis for Cloud Computing**
  Extend the methodology of Scale-Out processors to account for multithreading to derive pod designs with multithreaded cores that maximize the chip‘s performance density. Involving related work studies and simulations using, simulators (Simics/Flexus/Simflex) and CloudSuite benchmark suite.

LANGUAGES

- **English**: Advanced (C1)
- **French**: Native language

PUBLICATIONS

- Bloch, Aurelien, Simone Casale-Brunet, and Marco Mattavelli. "Design Space Exploration for Partitioning Dataflow Program on CPU-GPU heterogeneous system" Journal of Signal Processing Systems. (2023) Manuscript submitted for publication.

- Bloch, Aurelien, Simone Casale-Brunet, and Marco Mattavelli. "Dynamic SIMD Parallel Execution on GPU from High-Level Dataflow Synthesis." Journal of Low Power Electronics and Applications 12.3 (2022): 40.
- Bloch, Aurelien, Simone Casale-Brunet, and Marco Mattavelli. "Performance Estimation of High-Level Dataflow Program on Heterogeneous Platforms by Dynamic Network Execution." Journal of Low Power Electronics and Applications 12.3 (2022): 36.
- Bloch, Aurelien, Simone Casale Brunet, and Marco Mattavelli. "SIMD Parallel Execution on GPU from High-Level Dataflow Synthesis." 2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC). IEEE, 2021.
- Bloch, Aurelien, Simone Casale Brunet, and Marco Mattavelli. "Performance Estimation of High-Level Dataflow Program on Heterogeneous Platforms." 2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC). IEEE, 2021.
- Bloch, Aurelien, Simone Casale-Brunet, and Marco Mattavelli. "Inter-actions parallel execution on GPU from high-level dataflow synthesis." 2021 55th Asilomar Conference on Signals, Systems, and Computers. IEEE, 2021.
- Bloch, Aurelien, Simone Casale-Brunet, and Marco Mattavelli. "Methodologies for Synthesizing and Analyzing Dynamic Dataflow Programs in Heterogeneous Systems for Edge Computing." IEEE Open Journal of Circuits and Systems 2 (2021): 769-781.
- Bloch, Aurelien, Endri Bezati, and Marco Mattavelli. "Programming heterogeneous cpu-gpu systems by high-level dataflow synthesis." 2020 IEEE Workshop on Signal Processing Systems (SiPS). IEEE, 2020.
- Bloch, Aurelien, Endri Bezati, and Marco Mattavelli. "Composite data types in dynamic dataflow languages as copyless memory sharing mechanism." International Conference on Computational Science. Springer, Cham, 2019.
- S. C. Brunet, E. Bezati, A. Bloch and M. Mattavelli, "Profiling of dynamic dataflow programs on MPSoC multi-core architectures," 2017 51st Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, CA, USA, 2017, pp. 504-508.