Received 29 June 2021; revised 23 September 2021; accepted 26 September 2021. Date of current version 9 December 2021.

Digital Object Identifier 10.1109/OJCAS.2021.3116342

Methodologies for Synthesizing and Analyzing Dynamic Dataflow Programs in Heterogeneous Systems for Edge Computing

AURELIEN BLOCH, SIMONE CASALE-BRUNET (Member, IEEE), AND MARCO MATTAVELLI (Member, IEEE)

EPFL SCI-STI-MM, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland

This article was recommended by Associate Editor A. Bermak.

CORRESPONDING AUTHOR: A. BLOCH (e-mail: aurelien.bloch@epfl.ch)

ABSTRACT The possibility of using the increasing computing power available in cloud infrastructures requires the development of new approaches for application software development and optimization. Emerging edge computing paradigms offer the possibility of reducing bandwidth needs and of optimizing latency, features particularly relevant for Big Data applications, by bringing computation closer to the user and to the data generation processes. However, edge computing approaches pose several challenges in terms of how to be able to efficiently take advantage of a distributed network of heterogeneous processing nodes. This paper deals with this problem by extending a dynamic dataflow software development framework and related design flow tools to support heterogeneous platforms. The paper describes the methodology steps for the synthesis of application software executing on heterogeneous CPU/GPU co-processing nodes. The steps do include the optimization of the communication between heterogeneous processing elements, a technique for the efficient mapping and parallelization of computation on independent GPU partitions, and the introduction of dynamic programming approach for leveraging the SIMD nature of GPU computing. To complete the methodology of seamless porting of dataflow software and partition on CPU or GPU computing nodes, an automated methodology for exploring the configuration space and to identify high performance working points is developed.

INDEX TERMS Edge computing, dynamic dataflow programs, RVC-CAL, parallel computing, SIMD, source-to-source compiler, GPU programming, heterogeneous systems profiling, performance estimation.

I. INTRODUCTION

THE SUCCESS of the Internet of Things (IoT), and the growing trend of implementing decentralized processing applications close to the data generation devices, has created the needs for efficient edge computing approaches, in which data processing occurs partially (or completely) already at the network edge, rather than (completely) in the cloud [1]–[4]. The proliferation of diverse IoT devices and applications, and the growing diffusion of 4G/5G communication technologies, are gradually changing users' habits of accessing and processing all types of data. These trends challenge the linear growth capacity of cloud computing infrastructures. Edge computing can be considered as a new computing paradigm in which the data is processed at the edge of the network, where the usage of cloud resources can be reduced (or even completely eliminated) [1]. The problems that such new paradigm intends to solve covers several aspects: data access latency, limited battery life of mobile devices, limited amount and cost of bandwidth, security and privacy of interchanged or stored data are just some of the examples [2], [5], [6]. The solutions to these problems or resource limitations make it clear why there is a rapid growth in demand and a wide interest in this area, and why edge computing systems and tools are flourishing.

The purpose of this paper is first to briefly analyze from a high-level technical perspective what are the challenges and opportunities in this area and show how some of these can be addressed and solved using dataflow application development approaches. Then the paper shows how, by using high-level dataflow computation modeling, it is possible to efficiently exploit the computing power available at network nodes and enable automatic and dynamic reconfiguration of both the processing network geometry and the software application itself. To complete the approach with a design exploration and optimization stage, a methodology that allows to find close-to-optimal working points of the software application according to desired specifications and constraints is also described as well as the complete framework of tools supporting the integrated design flow [7], [8]. To the best of the authors' knowledge, this work is the first to propose a methodology entirely based on a dataflow paradigm, from design to implementation and final optimization, that fully exploits heterogeneous systems composed of GPUs and multicore CPUs.

The paper is structured as follows: Section II defines what is meant by edge computing, summarizes the state of the art, and analyzes which challenges and which opportunities this paper intends to address. Section III illustrates the principles of dataflows high-level computing models and shows how dataflow application software can be systematically analyzed, optimized and automatically synthesized for edge computing platforms. Section IV describes how the executable code of an application specified with an high-level dataflow computation model can be automatically generated by a low-level code generation framework implemented by the authors of this paper. Section V illustrates, with some examples, how edge computing applications could benefit from the methodology not only for the phase of development and synthesis of executable application code, but also for the phase of optimization and configuration of edge processing by exploring the design space with automatic profiling and estimation of the performance of the dataflow program execution. Section VI concludes the article and summarizes some open problems and future direction of research.

II. EDGE COMPUTING

Edge computing refers to technologies that allow computation to be performed at the edge of the network so that computation occurs close to data sources while reducing bandwidth usage between geo-locally distant devices [1], [3]–[5]. Edge computing can be applied on both downstream data on behalf of cloud services and upstream data on behalf of IoT services. An edge device is any computing or network resource that resides near the sources that generate the data and/or need the computing results. For example, an edge device could be a smartphone located between body sensors and the cloud [6].

A. CHALLENGES

Deployment strategies, i.e., where to map workloads, how to interconnect devices placed at the edge of the network while ensuring latency and power consumption specifications and how to deal with node heterogeneity are all issues that need to be considered for deploying applications at the edge. There are many challenges posed by this new computing and architectural paradigm. The following section provides a summary of the main challenges structured into five items as suggested in [5], [6].

1) GENERAL PURPOSE COMPUTING ON EDGE NODES

In theory, edge computing can be facilitated on various nodes that lie between the edge device and the cloud, including for example access points, base stations, gateways, traffic aggregation points, routers, and switches [1], [5]. Edge node can incorporate, for instance, digital signal processors (DSPs) that are sized and chosen based on the workloads they handle. In practice, base stations may not be suitable for handling analytical workloads simply because the DSPs are not designed for general computation or the required task. Furthermore, it is not immediately known whether these nodes can perform computations on top of their existing workloads. Therefore, it is necessary to be able to dynamically leverage all the computational and data storage power present in the vicinity of a node by allowing full reconfigurability and dynamic mapping of workloads. This is possible only if functions and tasks are defined with a high level language that is not specific to a given specific architecture [6].

2) PARTITIONING AND OFFLOADING TASKS

In edge computing, as highlighted by several research studies (e.g., see [5], [9]), programmers have to manually partition their application functions between various devices on the edge and the cloud. These early manual efforts are not scalable nor extensible. Thus, computation models and analysis methodologies (supported by easy-to-use programming frameworks and tools) are needed to enable automation of this phase where cost functions on compute and bandwidth resource utilization, and constraints on latencies and channel widths, can be automatically considered without necessarily requiring to explicitly define the capabilities or location of edge nodes [10].

3) DISCOVERING EDGE NODES

Discovering the resources and the services available in a distributed computing environment is a well studied and explored topic. Techniques such as benchmarking form the basis of decision making for mapping tasks to the most appropriate resources to improve performance [11]. However, exploiting the network edge requires discovery mechanisms to find appropriate nodes that can be leveraged in a decentralized cloud configuration and also requires dynamic reconfigurability of applications and devices [6], [9]. Thus, mechanisms are required that cannot simply be manual due to the huge volume of devices that must be available at this level. Furthermore, these mechanisms must allow for the integration (and removal) of nodes into the computational workflow at different hierarchical levels without increasing latencies or compromising the user experience, in a completely dynamic and secure manner [2].

4) QUALITY-OF-SERVICE AND EXPERIENCE

The quality provided by edge nodes can be defined by both the QoS and the QoE provided to each user [12]. An increasingly complex principle to follow is to not overload nodes with computationally intensive workloads. Thus, the challenge is to ensure that nodes achieve high throughput and are reliable. Regardless of whether an edge node is leveraged, the user of an edge device or data center expects a minimum level of service [9]. Thus, a thorough understanding of the geometry and utilization of nodes in the network is required, but at the same time raises issues related to monitoring, dynamic reprogramming at the infrastructure and application levels.

5) PRIVACY, SECURITY AND DATA INTEGRITY

If every edge device can potentially be used to store data and be accessible to the public, a number of security and reliability challenges need to be addressed. For example, a router that handles Internet traffic cannot be compromised when it is used as an edge computing node. A minimum level of service will need to be defined and guaranteed to each user of the edge node for decentralized computing to become possible and affordable [13], [14]. To this end, workloads, computation, location, and energy consumption will need to be considered to develop appropriate architectural and pricing models to make edge nodes affordable.

B. OPPORTUNITIES

In cloud computing, it's increasingly popular to use high-level languages to define and deploy workloads. For example, in the scientific domain (e.g., see bioinformatics and astronomy domains) it is increasingly common to define various work pipelines with dataflow languages [15], [16]. These pipelines require access to external databases, inputs and outputs are typically of substantial sizes, and the computational load is substantial. The dataflow approach has proven to be very useful in handling this amount of data being analyzed and produced by heterogeneous sets of tools [17], [18]. However, this approach typically involves running on well-specified hardware platforms, where issues such as optimizing computational loads, bandwidth reduction, dynamic application reconfigurability, and network geometry are points not considered. By developing methodologies, frameworks, and toolkits, it is then possible to accelerate and systematize how these data and computation streams can support general purpose computing with the addition of dynamically reconfigurable, variable geometry edge nodes [19].

The work presented in this paper aims to address the problem of how to schedule, configure and use the resources available in an edge computing network in an efficient and productive way starting from the application modeling up to its dynamically configurable implementation. The problems to be solved are to find a computation model that is expressive, but at the same time analyzable and portable, not specific to a particular hardware platform [17]. In addition, the programming language must be usable without having any information about the architecture in which the program will be implemented to ensure that the application is portable. Alongside the programming language, it is necessary to provide a methodology for analysis and automatic generation of low-level code that is able to support dynamic reconfigurations of the network geometry. The source code implemented on a particular platform must be automatically generated by the framework following directives obtained from a performance monitoring and analysis system capable of optimizing the final implementation and meeting design requirements such as computation speed, bandwidth and energy consumption [6]. With this in mind, this work illustrates how it is possible to efficiently generate low-level code optimized, in the specific example, for both multi-core and GPU systems and heterogeneous CPU/GPU systems.

1) COMPUTATIONAL MODEL

In order to support dynamic application reconfigurability on a variable heterogeneous architecture, it is essential that the application layer be defined with some level of abstraction that allows for analysis and monitoring. One model that allows this, is the dataflow computation model. With it, it is possible to define applications using a high level of abstractions that are architecture independent. Typically, a dataflow application consists of computational blocks interconnected with buffers where data exchange takes place. There are several different models of computation described in the literature [10], [17], [20]. For the aim of this work, a dynamic dataflow model, supported by a standard language [21]–[23], will be briefly introduced. The dataflow approach has shown that it support the development of highly complex real-world applications [24]–[26], the portability on any network of heterogeneous processing nodes including CPUs and reconfigurable HW and the exploration of close-to-optimal configurations without the need of manual rewriting of the application.

2) SPACE EXPLORATION AND OPTIMIZATION

A network of edge computing nodes has many design points in terms of selecting and configuring software components and hardware architectures for implementation. These point choices create a large space of possible design solutions referred to as design space. The design process requires exploration through this design space to find design solutions before actual implementations [7], [27], [28]. The purpose of design space exploration (DSE) is to find design solutions that satisfy functional and performance constraints and enable optimization of all or certain portions of the system. These requirements have led to the notion of system-level design, in which key roles are played by aspects such as high-level modeling and simulation.

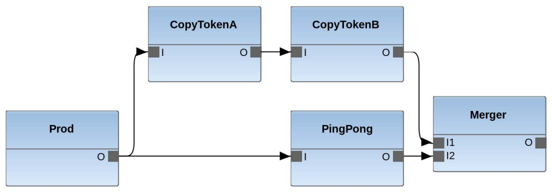
III. DATAFLOW MODEL

One of the most important properties of dataflow programs is their high-level modular structure, which increases the portability of programs across heterogeneous platforms. Dataflow programs are considered to be highly analyzable and platform independent [29]–[31]. In fact, dataflow computation models explicitly expose the potential parallelism of an application and provide the basis for developing extensive and systematic implementation analysis methodologies [30], [31]. For these interesting features, they are often considered as an alternative to the classical sequential programming methods particularly when targeting many-multicore processing systems. The features of dataflow computation models have been recently an objective of several research works [7], [30]–[35].

A. RVC-CAL PROGRAMMING LANGUAGE

In principle many different programming languages can be used to model the semantic or model of computations expressed by dataflow programs [35]–[37]. Imperative languages (e.g., C/C++, Java, Python) have been extended with specific parallel constructs and/or data communication functions, or specific dataflow languages (e.g., Ptolemy, Esterel) have been developed and formalized with native dataflow based operators. In this variegated ecosystem, the RVC-CAL [22], [38] dataflow is currently the sole dataflow programming language standardized by ISO that fully captures, natively in its operators the behavioral features of a dynamic (dataflow) process network (DPN) model of computation (MoC). An RVC-CAL program is composed by a network of actors (which contain the algorithmic part of the program) interconnected by means of order preserving lossless communication channels (that usually are implemented by FIFO buffers). Each actor contains a set of atomic firing functions, called actions, and a set of internal state variables which cannot be shared among other actors. During execution, actors can fire only one action at a time: for each action a set of firing rules defines when the action can be executed and each firing rule can be defined as a function of the actor's internal variables and on the availability and values of the input tokens.

As an example, Figure 1 shows an RVC-CAL dataflow program. Figure 1(a) is the graphical representation of the Top-level network of the program. It is composed of five actors instances. Figure 1(b) depicts the RVC-CAL implementation of the Producer actor. It consists of a single action that produces one token per firing that increment each time. A guard prevents the action to fire more than four times. In the source code specifying the PingPong actor processing in Figure 1(d) a schedule statement acts as a Finite-state machine (FSM) where the transition from one state to the other is done by the firing of an action. In this work the Open RVC-CAL Compiler (Orcc) [39]–[41] has been used as development tool and compiler framework for the RVC-CAL programs illustrated in the following sections, but also other open source compilers can be used for the same purposes [28], [37], [42].



(a) An example with five actors (i.e. *Prod*, *CopyTokenA*, *CopyTokenB*, *PingPong* and *Merger*).

```
actor Producer () ==> int O:
  uint counter := 0;

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

(b) Producer.cal

```
actor CopyTokens (String name) int I ==> int O:
  @CUDA(block="1", thread="2")
  c: action  I:[val] ==> O:[val] end
end
```

(c) CopyTokens.cal

```
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
```

(d) PingPong.cal

```
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
```

(e) Merger.cal

FIGURE 1. RVC-CAL program example: dataflow network topology and actors source code.

B. DESIGN SPACE EXPLORATION

The quality of an application implementation is undoubtedly subject to finding an appropriate combination of different permissible configurations in the implementation space. These configuration points can generally involve: the assignment of data flow modules to processing units (partitioning), the ordering within each processing unit (scheduling), and the sizing of the communication channels and bandwidth used by the modules (buffer sizing). However, due to the large number of allowable configuration points, the problem of finding a configuration that satisfies the design constraints is generally considered to be NP-complete [8]. As objective, only close-to-optimal solutions are desired. These can only be derived using an appropriate design space exploration heuristic supported by an easily analysable execution model. An effective approach that has been shown to be able to efficiently explore the design space is based on the analysis of the execution trace graph (ETG) of a dataflow program [43]. The ETG is modeled as a directed acyclic graph (DAG) in which each node represents a single action firing and each directed arc represents either a data or a logical dependency between two different action firings [43]. Based on this methodology, a design space exploration framework called TURNUS has been developed [44], [45]. This framework can be used to explore and optimize the design space of an RVC-CAL application: its main components and its design flow are extensively discussed in [7]. The inputs to this graph are the RVC-CAL program, the target architecture in which the project will be implemented, and the constraints imposed by the architecture (e.g., available resources) and/or the designer (e.g., performance requirement). The output of the design flow is the implementation of the dataflow program, which may consist of a circuit synthesised in hardware and/or a binary executable via software. By executing or interpreting the code, it is possible to achieve a profiled, high-level, platform-independent execution of the program in which the ETG is evaluated.

C. PERFORMANCE ESTIMATION

The ETG is then analysed by adding measurements obtained from application profiling. Using this information, it is possible to calculate and identify the longest computational path, called the critical path (CP), which occurs when a set of input vectors is processed. Based on the CP information, it is possible to effectively guide the heuristics required to evaluate a mapping configuration that minimises an objective function (e.g., buffer size) based on certain constraints (e.g., throughput) provided by the designer. TURNUS also provides a fast performance estimation engine based on ETG analysis. This is used to quickly and efficiently explore different design alternatives (i.e., mapping configurations) of the design while minimising the number of low-level implementations [7].

IV. EXECUTING A SW APPLICATION ON HETEROGENEOUS EDGES NODES

As outlined in [6], one of the challenges in the implementation of edge computing systems is to develop solutions that are portable to different and possibly heterogeneous computing nodes or networks of nodes. It is clear that exploiting the opportunity offered by the edge computing paradigm necessitates encompassing a variety of different heterogeneous hardware architectures and configurations in a distributed manner. Thus, the need for a systematic and automated way of generating efficient configurations of computational tasks running on the heterogeneous hardware resources available at the edge, is a necessary feature to achieve the necessary portability, flexibility and ease of development needed to deploy IoT/edge applications. This section presents a methodology, capable of automatically generate C++/CUDA code for the execution of DPN dataflow SW on heterogeneous CPU/GPU co-processing platforms. Here the emphasis is on the CPU/GPU partitioning of processing tasks, whereas the partitioning on homogeneous many/multi core processing platforms is extensively discussed in [7], [8], [43]. Examples and experimental results are also provided and discussed.

A. CODE GENERATION

In this work, the CUDA (Compute Unified Device Architecture) [46] application programming interface is used to execute networks of DPN actors on NVIDIA GPUs, but the methodology and approach is general and can in principle be extended to other GPU interfaces and platforms. However, by using CUDA, the fine-grained control over the NVIDIA hardware range of different GPU generations and families can be fully leveraged. The CUDA notation for multiple computing context is extensively used, including dynamic parallelism, and the concurrency between memory transfers and computation.

A typical CUDA application program consists of an host (CPU) and a device (GPU) code in a single-source program using pre-processor directives to differentiate where the computation occurs. The device code is written in a collection of *kernels* that are special functions that can be called from the host code and are executed on the device. So as to take advantage of the massive GPU parallelism, the launch of kernels has to be parameterized by a number of SIMD threads that can exploit the full potential of the given GPU platform. Such threads are decomposed in groups of 32, called *warps*, in such a way that they all execute the same instructions. Sets of *warps* can be organized in thread blocks where all threads of the same block can have access to fast shared-memory and synchronization primitives.

The approach taken in this work is, for selected dataflow actor in a dataflow network that can benefit from GPU execution, to execute both the action selection and the actions themselves on the GPU. By doing so, the necessary data to fire an action is already available on the device memory. Thus, a minimization of the data transfers between the host memory and the device memory is fully achieved. Such approach of fully porting an actor execution on a GPU, instead of the classical SIMD approach using GPUs as co-processing units, also avoids the need of allocating a CPU core for each actor to schedule and launch kernels for GPU co-processing.

Actors can thus be either fully mapped to execute on the host (CPU) or on the device (GPU). The implementation of this porting of the dataflow network nodes, requires three different FIFO communication mechanisms: one to implement the communication between actors mapped on CPUs, one for communications between actors mapped on the GPU and one for communication between an actor mapped on a CPU and one on a GPU. For each type of communication mechanism, a different implementation is required. These three FIFO communication implementations fully enable the independent and parallel computation between the host and the co-processing device, but also the full parallel execution of actors on the co-processing device (more details are reported in Section IV-C).

So as to maximize the GPU resource utilization, multiple independent compute context are used to concurrently run different dataflow actors on separate CUDA streams. Multiple actions can be fired in parallel depending on the available GPU device resources in terms of CUDA cores, memory, and registers. Multiple CPU/GPU memory transfer might conjointly occur by using the copy engines of the GPU platform. The specific number of copy engines may depend on the hardware generation and model. This number may impact the concurrency between the memory transfer and compute time of task executions, and as a consequence the overall dataflow program performance may be affected.

As mentioned in Section III-A, by construction 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 buffers. Thus, actors are not directly ready for internal parallelization. Nonetheless, the performance of an actor execution can indeed be improved by parallelizing its computation. Performance improvements can be achieved by launching several instances of the same action in parallel on different data, as long as the order of the tokens within the communication channels between actors is preserved and the internal state dependencies of each actors respected.

B. PARTITION AND MAPPING

A partitioning configuration of the dataflow network nodes (actors) and the mapping on CPU or GPU for execution, has to be provided for the generation of the execution code for both CPU and GPU. This stage consists of assigning to each actor or sub-network sets whether they are mapped on the CPU side and in which sub-partition (for multi-core systems) or they are mapped on the GPU side. Such assignment allows the synthesizer and compiler to distinguish and correctly assemble by using the appropriate communication instantiation (CPU-CPU, GPU-GPU or CPU-GPU) the generation of the code intended to be executed on the CPU and the code for the GPU platforms. For doing so a new backend for the generation of CUDA code has been developed by the authors and added to the open source synthesis and compiling infrastructure called Exelixi [47]. The tool flow is presented in Figure 2. The high-level representation of the application program written in RVC-CAL, together with configuration files providing partitioning and buffer sizes information, are fed to the ORCC compiler, which uses the Exelixi CUDA backend [48], [49] to generate the C++/CUDA code that is then compiled with the *Nvidia* CUDA Compiler (NVCC) to obtain an executable of the heterogeneous program. Using a platform-specific compiler as the last layer of the tool-chain allows the methodology to be compatible with all *Nvidia* supported platforms (i.e., X86(_64), ARM, POWER9, and all *Nvidia* GPUs). To fully exploit actions parallelization, user defined compiler directives are used to identify suitable actions and associated number of blocks and threads necessary to fully specify the SIMD execution of parallel actions.

In other words, the generated code is capable of instantiating actors, creating partitions for the CPU to be executed using POSIX threads, instantiating CUDA actors and launching in separated streams their corresponding principal *kernels* (*action_selection*), and instantiating and connecting the appropriate FIFOs implementations connecting the inputs and outputs actors ports (see Section IV-C).

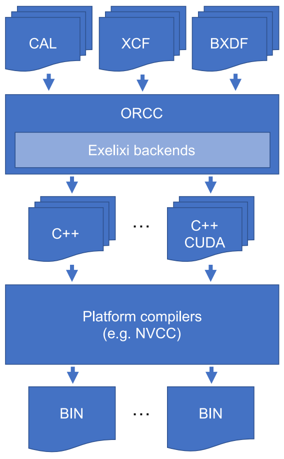


FIGURE 2. Tool flow of the heterogeneous code generation.

```
__global__ void CopyTokenNS::action_selection(
  CopyToken* copyToken, EStatus* status,
  size_t actorIdx, size_t actorSize) {
 uint64_t _start = clock64();
 bool stop = false;
 do {
  status[actorIdx] = None;
  bool res1 = true;
  while (res1) {
   res1 = false;
   copyToken->status_I = copyToken->port_I->count(0);
   copyToken->status_O = copyToken->port_O->rooms();
   bool res2 = true;
   while (res2) {
    res2 = false;
    if(copyToken->status_I >= 2 &&
     copyToken->isSchedulable_c()) {
     if(copyToken->status_O >= 2) {
      Ports ports;
      ports.I=copyToken->port_I->read_address(0,2);
      ports.O=copyToken->port_O->write_address();
      CopyTokenNS::c<<<1,2>>>(copyToken, ports);
      res1 = true;
      res2 = true;
      status[actorIdx] = hasExecuted;
     }
    }
   }
  }
  if (checkStatus(status, actorSize) == None) {
   stop = (clock64() - _start) > wait_period;
  } else { _start = clock64(); }
 } while(!stop);
}
```

FIGURE 3. Stripped-down example of the *CUDA* implementation of the action selection of the *CopyTokens* actor.

In the listing reported in Figure 3 an example of the source code generated by the backend for the *CopyTokens* actor meant to run on the GPU is reported. The *__Global__* directive designates *kernel* function. The *action_selection* is the principal *kernel* that is launched from the *main* function, it is responsible of testing which action has to be executed next by looking for available data, available FIFO output space and for constraints expressed by guard and represented by the *isSchedulable_c* function in line 19. As can be observed in line 24, the actions *c* is also a *kernel*, launched from the *action_selection*, and this is possible thanks to the added dynamic programming functionality provided by the *CUDA* API. The main benefit of such functionality is that it allows to select the number of SIMD threads available for the execution of each action depending on the executed code. In this example a "$<<< 1, 2 >>>$" configuration is used, which means that one block of two threads is used. It is important to remark that the management of the read and write addresses in the FIFOs must be handled in the *action_selection* instead of in the action itself, since they must be called only once and in the correct sequence. In addition, it can also be noticed, at line 22, that the read address is obtained by providing to the FIFOs the number of tokens that will be accessed (here one token multiplied by the number of thread, two in this case). This information is particularly relevant to ensure memory alignment so that a specific quantity of data can be accessed in consecutive addresses.

Within an action, both the read and the write addresses are indexed with the *token id* and the *thread id*, so that each thread works on its own portion of memory. Maintaining sequential semantics in FIFOs allows to easily connect parallelized actors with sequential actors in a transparent way. Each actor runs autonomously on the GPU without the need to be launched periodically by the CPU. In terms of the data flow model, the result is to have each CUDA actor mapped to a separate partition. Each actor is responsible for terminating itself by detecting when the application program reaches the end. In other words, the CUDA partition can be considered completely autonomous as long as the application program is executing.

With this aim, the *action_selection* is made as a long-running kernel with a loop that run until the application reach the end. In the listing reported in Figure 3, at line 27 it is defined how when an action is triggered the *status* array is put at *hasExecuted*. This array is shared by all other CUDA actors and is used to ping other actors and assert that a new computation has just been completed. The *checkStatus* function in line 32 is responsible for checking other actors' status. If no actor has produced any work after some fixed amount of time (*wait_period*) then the actors return. The application programs can terminate once all CUDA actors and all CPU actors are running.

To visualize how a typical program generated by this methodology is executed, Figure 4 depicts a possible execution of the dataflow program example reported in Figure 1 when executed with the partition configuration reported in Figure 5, where each GPU actor runs on its own separated CUDA stream. In Figure 4 it can be observed that in slot 1 (in red) the first firing of the action *p* of the *prod* actor is performed. The produced token of *p* is successively available in slot 2 to the *PingPong* actor. As depicted in Figure 5, the same token needs to be transmitted to *CopyTokenA* that

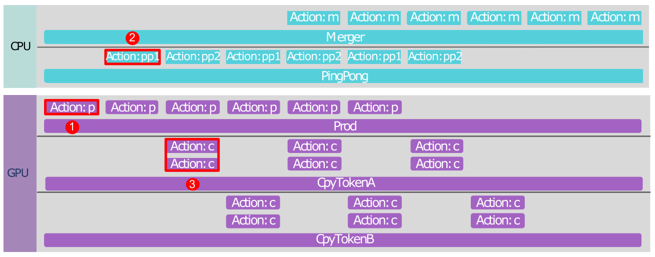


FIGURE 4. Cuda example of execution.

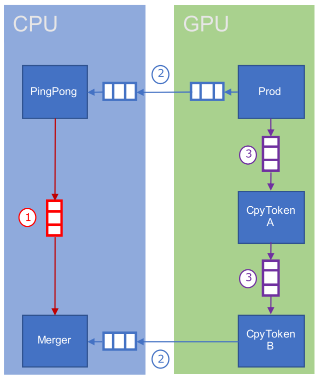


FIGURE 5. Program partitioning over CPU and GPU. Three types of FIFOs are used (1-CPU FIFO, 2-Legacy-HostFifo/HostFifo, 3-Device FIFO).

process it in slot 3 at the same time as the second token through the firing of the action *c*. As it can be observed in Figure 1(c), the action *c* has been flagged with the *@CUDA* annotation to make it an SIMD action with 1 block of 2 CUDA threads, meaning that the action will always execute twice in parallel. Finally, it can also be observed that simultaneous CPU and GPU computation as well as GPU to CPU memory transfer are fully achieved.

C. COMMUNICATION

Three types of FIFOs, respectively called *Fifo*, *CudaFifo*, and *HostFifo*, are needed in order to provide the correct support of a CPU/GPU co-processing execution. All of them are illustrated in Figure 5. The first type, inherited from the original Exelixi C++ backend, and represented in red is used for CPU to CPU communication. The second one, represented in purple, is used for GPU to GPU communication. Its implementation is very similar to the CPU FIFO except from the fact that it uses device memory instead of CPU memory. The last one, represented in blue in the figure, is used for any FIFO for which at least one of the writer or the readers is on the CPU side and one on the GPU side.

A *HostFifo* is a cross-platform FIFO that has been developed by exploiting the features of recent CUDA APIs and associated hardware capabilities. With this aim, on the CPU side the pinned memory is allocated. This means that the operating system does not need to swap this allocation nor to move it in a different physical address space. These addresses are successively registered in the virtual address space of the GPU and then the translated pointers are obtained. In this way, different pointers are used if memory is accessed from the CPU side or the GPU side. The advantage of such implementation is that there is no need for any synchronisation or software API calls: memory accesses are done automatically in hardware. This efficient implementation does not change the rest of the dataflow computation model or how the FIFOs are used. This third type of FIFO implementation is only compatible with hardware equipped with computing capability 6.x or higher (available hardware features), which are able to perform on-demand fine-grained memory pinning. Less efficient implementations that fully support the dataflow computation model are obviously also possible for platforms not supporting recent CUDA APIs. For hardware with compute capabilities under 6.x a *Legacy-HostFifo* is provided. It is composed of two identical *CudaFifo* that are allocated one on the CPU side, one on the GPU side and of additional functions *hostFifoSyncWrite hostFifoSyncRead* that have to be called from the CPU to synchronize them. Since these FIFOs accept one writer, but multiple readers, it is necessary to keep track of which readers/writer is on which sides so that the most up-to-date data can be identified and the direction to which explicit memory transfers should be issued can be inferred. However, The fact of continuously using CUDA software API to synchronize the two FIFO, introduces a significant runtime overhead compared to the hardware optimized *HostFifo* implementation.

D. EXPERIMENTAL EXAMPLE

In this section, two programs have been used to evaluate the methodology describe in this paper. The objective is to demonstrate the correctness and completeness of the code generation and to assess the achieved performance improvements.

For the experimental evaluation, the hardware used is a GeForce GTX 1660 SUPER NVIDIA GPU with 6GB of memory and an Intel Skylake i5-6600 CPU with 16 GB of DDR4 RAM. As for the software, the CUDA library version 11.3.1 has been used.

1) RVC-CAL IDCT SYNTHETIC APPLICATION

This section presents the results obtained from the parallelization of a computationally intensive actor. Here, this application is made with an actor implementing the *idct* algorithm connected to a *Source* and *Sink* actor responsible to feed the it with data (See Figure 6). For this example, two different configurations have been used. The first one is CPU only with three CPU threads and one actor per thread. The second one is GPU only and the CUDA kernel of the action of the *idct* actor is running on a grid of 2 blocks 512 threads per block. Both configurations were implemented with identical buffer sizes. The result is that the GPU configuration is more than two times faster than the CPU implementation. This demonstrates that if enough data is provided to the GPU-generated actors, this methodology provides significant performance improvements over the CPU-only multicore implementations.

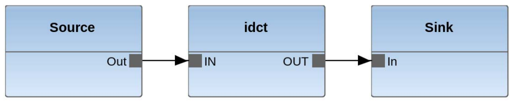


FIGURE 6. Network for the RVC-CAL IDCT application.

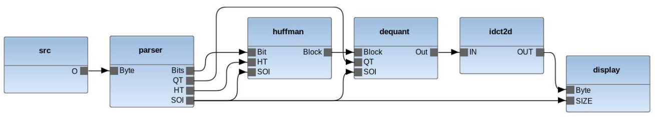


FIGURE 7. Network for the RVC-CAL implementation of a JPEG decoder.

TABLE 1. Results for the JPEG decoder.

	Seq GPU	Parallel GPU
Frame rate [image/sec]	0.24	2.32
Speedup	1	9.67

2) RVC-CAL JPEG DECODER

Figure 7 shows the top-level network of a RVC-CAL JPEG decoder composed of 6 actors. This is well-known application and its source code is available in the orc-apps repository [50].

In this application, only the *idct2d* previously analysed can be parallelized, as it does not contain any internal data dependency. This real-case application is used to validate the correctness of the code generated for a generic application that has not been written specifically to explore GPU parallelisation, showing the generality and functionality of the methodology described in this work. Regarding performance, Table 1 summarizes two different sets of results. The first one is when the *idct2d* actor runs on the GPU sequentially (this corresponds to the methodology presented in [48]), all other actors are running on the CPU. The second one corresponds to the improved methodology where the *idct2d* actor runs in parallel on the GPU. It can be noticed that the parallel GPU implementation is almost 10 times faster than the previous sequential implementation.

V. PARTITIONING AND OFFLOADING TASK

Making use of edge nodes for offloading computations poses the challenge of not only partitioning computational tasks efficiently, but doing so in an automated and dynamic way, without requiring to *a priori* and statically define the configuration, network geometry, and capabilities of each node [6]. This is because the Edge computing paradigm requires a reliable way to automatically discover good configuration points in terms of mapping and partitioning against available nodes.

Figure 8 illustrates how the dataflow methodology is used for the distribution of computational tasks both locally to each heterogeneous system and globally to the entire edge computing system. Indeed, communication between processing elements using FIFO buffer allows the model

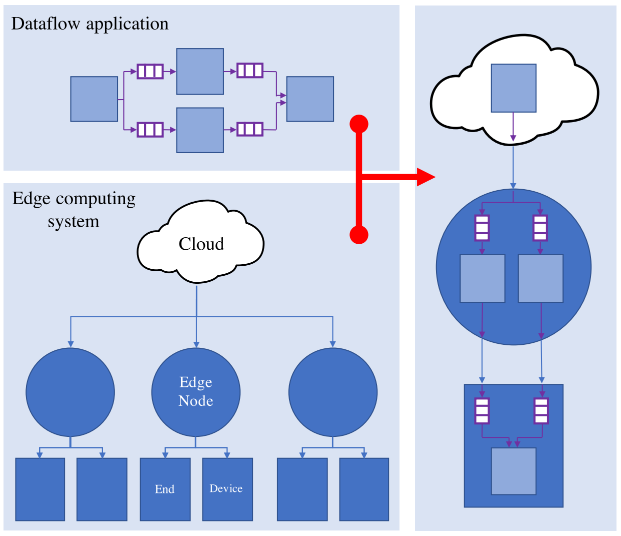


FIGURE 8. Illustration of how a dataflow application can program edge computing systems by mapping actors across the cloud, edge device and end device seamlessly when all communications are abstracted by FIFO buffers.

to encapsulate both the local communications via memory transfer and the global communications over the network in a way that is transparent to the processing element. In addition, performance evaluation and design space exploration are also unaffected, as the only difference would be in the higher communication cost that is measured in the profiling weights. In this example, we can see a four-actors dataflow application program description that is mapped on an Edge computing system where the chosen configuration allows for one actor to run in the cloud, two actors on an intermediate edge node, and one on the end-user device. We can also notice that all communications (local or remote) are performed through FIFO buffer (in purple).

This section presents how the Exelixi CUDA backend [47] has been extended to synthesize instrumented code to perform accurate clock profiling and generate execution weights. It shows how this can be used in the TURNUS post-processor to quickly and accurately estimate the overall performance of the application and then automatically propose performance configurations to the backend which can then generate code adapted for the available computational nodes.

A. CLOCK-ACCURATE PROFILING

For the purpose of this work, an extension of the compiler in the Exelixi CUDA backend was developed to automatically achieve accurate profiling capabilities at the clock of an application. It is necessary to specify that this way of code generation is only used during the profiling phase of the application and not for the final execution phase. In order to increase the accuracy of the profiling, it was necessary to modify the behavior of the CUDA parallel partition and the behavior of the GPU actor scheduler. In fact, to avoid interferences (e.g., memory transfer, hardware resource contention) all actors are executed only in sequential mode and

```
// -- PROFILING: TICK
asm volatile("mov.u64␣%0,␣%%clock64;" : "=l"(__cycles_0) :: "memory");
// action body code to be profiled ....
// -- PROFILING: TOCK
asm volatile("mov.u64␣%0,␣%%clock64;" : "=l"(__cycles_1) :: "memory");
actor_a->profilingData->addFiringDevice(ACTOR_ID::actor_a,
                        ACTION_ID::compute,
                        (__cycles_1 - __cycles_0));
```

FIGURE 9. Striped down example of the *SASS* assembly code used for accessing the GPU clock counter.

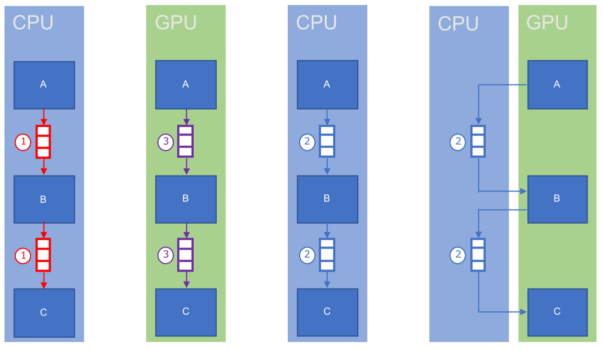
a sequential partition is created that schedules at most one CPU and one GPU actor at a time. The type of actor scheduling is then changed from fully parallel to non-preemptive: in other words, an activated actor is executed until it has completed its execution or entered a waiting state.

In addition to the execution mode, for an accurate measurement of the clock-cycle it is necessary to have access to a hardware counter with increment at a stable frequency. For CPU-mapped actors, the same Intel *RDTSC* counter proposed in [51] is used. For GPU-mapped actors, however, something dedicated and equally precise is needed. The NVIDIA platform used for this work offers something similar in their streaming multiprocessors, Figure 9 shows the SASS assembly code used to access a stable and precise performance counter to the clock. Access points can be placed around the section that is to be profiled. For SIMD actions where multiple CUDA cores are used to run multiple instances of the action simultaneously, the measured time is divided by the number of threads used to execute the action. This results in a virtual average time for the execution of a single action, since this is the number needed to execute the TURNUS ETG postprocessor.

Compared to the software implementation for homogeneous platform methodology previously developed, it is not possible to assume that the communication cost of a specific action do not vary regardless of the mapping of the actor it is communicating with. Indeed off-board communication (i.e., CPU to GPU) is much more costly, and the communication and execution phases during the firing of an action must be considered as independent.

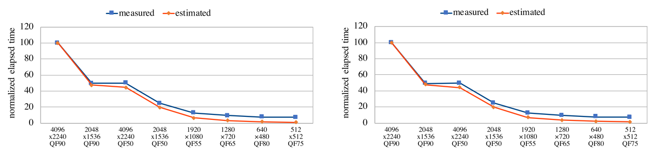
B. HETEROGENEOUS ESTIMATION

In this section, it is shown how the profiling weights generated by the methodology presented in the previous section are used. An important aspect to consider is how to estimate and treat the difference in CPU and GPU clock frequency. This is important information that must be provided to TURNUS in order for performance estimates to be made consistently. To obtain the CPU frequency, the *RDTSC* counter is measured for a known time: the result will provide the frequency value used for the *RDTSC*. For the GPU clock, NVIDIA exposes this value through the CUDA API. The last thing to think about is clock volatility. Indeed, whether it is called *boost*, *turbo boost*, *step speed* or *dynamic clocking* different technology for clock frequency variation exist on both sides. To optimize the performance estimation accuracy all these options are temporarily disabled.



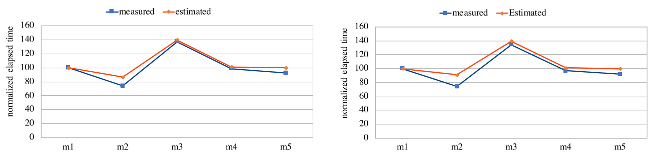
(a) Full CPU (b) Full GPU (c) CPU HostFifo (d) GPU HostFifo

FIGURE 10. **Four FIFOs configurations needed during profiling.**



(a) Buffer 512 (b) Buffer 1024

FIGURE 11. **JPEG: Normalized values with different inputs.**



(a) Buffer 512 (b) Buffer 1024

FIGURE 12. **JPEG: Normalized values with different partitions.**

In order to provide the TURNUS design space exploration framework with a detailed view of the architecture, it is important to profile the application in various scenarios. Those that manage to provide detailed and reliable information are the four scenarios described in Figure 10. The first and the second scenario are those in which all actors are mapped on the CPU partition for the first and GPU for the second. They both allow to have the scheduling and action execution body cost on each platform. Unfortunately, they are not sufficient as they do not allow to profile the communication cost across platform (HostFifo ② in blue) but only the CPU to CPU communication (Fifo ① in red) or GPU to GPU communication (CudaFifo ③ in purple). To solve this problem two extra profiling with a special Backend option (HostFifo only) activated which artificially generates HostFifo (FIFO for cross-platform communication) between every actor allowing to get the cost for any possible configurations are performed.

C. EXPERIMENTAL EXAMPLE

In this section, two programs have been used to validate if the theoretical model used in TURNUS and the implementation code and profiling result generated by the CUDA backend presented in this work are matching. These application programs are well-known and the source code is available in the orc-apps repository [50]. For the experimental evaluation, the hardware used is the same as in Section IV-D and as explained in Section V-B CPU frequency has been stabilized to 2.9GHz and the GPU frequency to 1.8GHz.

1) RVC-CAL JPEG DECODER

In this section the JPEG decoder application program is also used and its Top-level network is shown in Figure 7. It is composed of 6 actors. Besides the *Src* and *Display* actors that are handling input/output, reading/writing, all other actors can either be assigned to the CPU partition or the GPU partition.

The first set of results focuses on a single partition with varying buffer sizes, input image resolutions, and quality factors. The reference configuration is composed of the *Src* and *Display* actors in a single sequential partition mapped on a CPU core and the other actors mapped to the fully parallel GPU partition. For every configuration, a corresponding ETG has been generated using TURNUS. The profiled executable generated by the CUDA backend has been executed as many times as the set of different input sequence stimuli to generate the weights. The TURNUS ETG post-processor has been used to generate the total estimated execution time of the application program and compared it with the actual measured total time. Figure 11 shows the results, where the design has been simulated with 8 different images and two different buffer sizes (512 and 1024 tokens). The values are normalized to the first image and it can be seen that the maximal estimation error is around 6%. This shows that regardless of the inputs and buffer sizes the performance could be estimated just the same.

The second set of results focuses on the same comparison between the total time measured and estimated but this time with the same input stimuli and varying mapping configurations. Contrary to the first set of results and since the same input image is used a single ETG has been generated. Regarding the weights and as explained in Section V-B only four sets of weights files are needed and each simulation is run with the appropriate combination of these weights. Figure 12 show the results, where five random mapping partition as been used. Each mapping (m1-m5) corresponds to a random assignment of each actor to either the CPU sequential partition or to the fully parallel GPU partition. The values are normalized to the first partition and it can be seen that the maximal estimation error is around 16.9%. More importantly, it can be seen that the performance improvement or deterioration trend has been respected and this without the need to generate new weights or ETG for each partition.

2) RVC-CAL SMITH-WATERMAN ALIGNER

Figure 13 shows the Top-level network of the Smith-Waterman (S-W) aligner application program presented in [52]. The S-W aligner performs a local alignment of two sequences of RNA, DNA, or protein. The first sequence

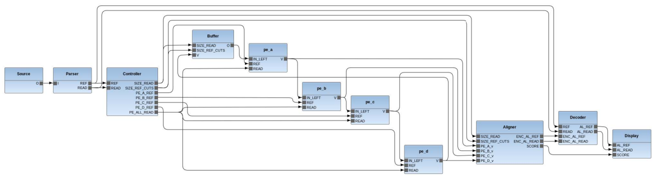


FIGURE 13. RVC-CAL implementation of a S-W aligner.

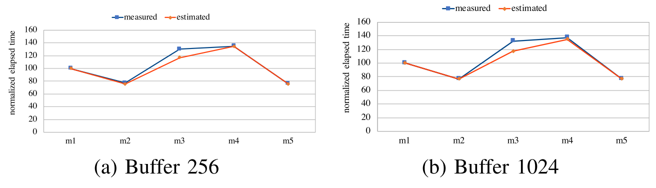


FIGURE 15. S-W: Normalized values with different partitions.

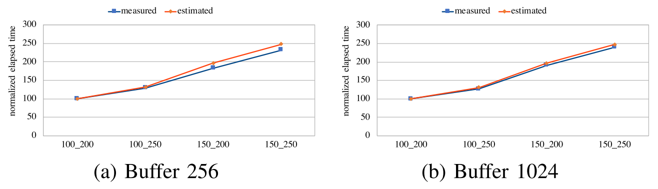


FIGURE 14. S-W: normalized values with different inputs.

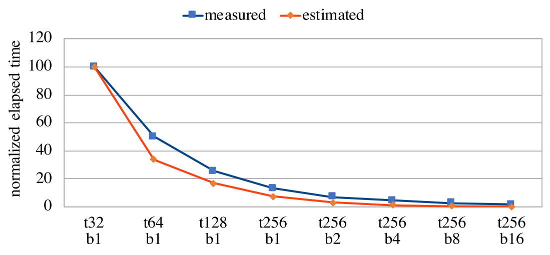


FIGURE 16. IDCT: Normalized values with different SIMD threads numbers.

$A = \{a_1, a_2, \ldots, a_n\}$ is generally referred to as the reference, and the second one $B = \{b_1, b_2, \ldots, b_m\}$ as the read (or query). The S-W is composed by two computational stages: a first step where a cost matrix is evaluated and a second step where this matrix is backtracked starting from the highest matrix value. The backtrack path defines the alignment (in terms of matches, mismatches, insertion, and deletions) between the reference and the query input sequences. The dataflow program used in the context of this work is composed of 11 actors and in Figure 13. The main components are the four PE elements that are the core elements in charge of evaluating the matrix scores and the Aligner module that is in charge of evaluating the backtracking path on the matrix. Besides the *Source* actor that is handling inputs readings, all other actors can either be assigned to the CPU partition or the GPU partition.

Similar to the JPEG Decoder application program two sets of result are presented. The first set of results focuses on a single partition with varying buffer sizes, and inputs sets. Each input set is named $l_m\_l_n$ with l_m being the length of the reads (query) and l_n the length of the reference sequences. The reference configuration is composed of the *Source* actors in a single sequential partition mapped on a CPU core and the other actors mapped to the fully parallel GPU partition. The same methodology is then applied to generate the result presented in Figure 14, where are provided 4 different inputs (100_200, 100_250, 150_200, and 150_250) and 2 different buffer sizes configurations (256 and 1024 token places). The values are normalized to the first input and it can be seen that the maximal estimation error is around 15.6%. This shows that regardless of the inputs and buffer sizes the performance could be estimated just the same.

The second set of results also focuses on varying the mapping configurations and the same methodology is applied. Figure 15 shows the results, where five random mapping partitions as been used. Each mapping (m1-m5) corresponds to a random assignment of each actor to either the CPU sequential partition or to the fully parallel GPU partition. The values are normalized to the first partition and it can be seen that the maximal estimation error is around 14.9%. More importantly, it can be seen once again that the performance improvement or deterioration trend has been respected.

3) RVC-CAL IDCT

In this section, the IDCT application program from Section IV-D1 is used and its Top-level network is shown in Figure 6. It is composed of 3 actors, that are all mapped to the GPU side. The results focus on a single partition with varying numbers of parallelization for the actions' execution. Each point is using a different number of thread blocks and thread per block. It goes from 1 block of 32 threads to 16 blocks of 256 threads each. A single ETG has been generated using TURNUS. For every configuration, the profiled executable generated by the CUDA backend has been ran with each time a different kernel launch configuration. The TURNUS ETG post-processor has been used to generate the total estimated execution time of the application program and compared it with the actual measured total time. Figure 16 shows the results, where the design has been simulated with 8 different SIMD sizes. The values are normalized to the first configuration and it can be seen that the maximal estimation error is around 16%. More importantly, it can be seen that the performance improvement or deterioration trend has been respected.

VI. CONCLUSION AND FUTURE WORK

A new methodology based on dataflow programming for the porting on heterogeneous CPU/GPU processing nodes is described in this paper. It intends to address two challenges of edge computing, namely: the software generation for heterogeneous and distributed nodes, and the problem of partitioning and distributing tasks to networks of processing nodes and dynamically modify their geometry and partitioning. The results have been achieved by extending the formalism already developed by the authors, based on the modelling, analysis and automatic generation of application source code from a high-level representation based on a dynamic dataflow computation model to include porting and partitioning on CPU/GPU nodes. In particular, the methodologies and results are obtained for parallel software implementations of CPU/GPU co-processing for CUDA platforms using RVC-CAL as high-level dataflow language. The experimental results showed a twofold performance improvement of the new GPU code generation over the CPU implementation using the IDCT application and a tenfold faster implementation when comparing the previous GPU code generation with the one containing the optimization presented in this paper when using the JPEG decoder application. To complete the methodology including a design exploration stage for the systematic search of close-to-optimal configurations in terms of execution performance, a method to estimate the executions of programs on heterogeneous CPU/GPU co-processing platforms is also presented. The methodology is applied independently of the chosen configuration (i.e., defined by partitioning, mapping, buffer sizing, and SIMD flag) without the need to run the application in any possible configuration on the hardware platform, thus allowing efficient exploration and optimization heuristics to explore very large design spaces in short times without reducing the accuracy of the results found. The experimental evaluation compared the measured and estimated performance in three different result sets. The first is where fixing the RVC-CAL network mapping configuration but varying the inputs and buffer size, the maximum estimated error is 6% to 15.6% for the JPEG decoder or Smith-Waterman alignment program respectively. The second set is where, fixing the input file but varying the mapping configuration, the maximum estimated error is 16.9% to 14.9% respectively for the same programs. The last experimental result shows that when varying the number of parallel CUDA threads for action parallelization in the IDCT program, the maximum estimated error is 16% More importantly, in all experiments, the trend of performance improvement or deterioration between the estimated and measured results is respected, allowing for proper exploration of the design space. Future extensions of the work presented in this paper, include the development of a runtime system that triggers source code generation, compilation and automatic code distribution depending on the nodes in the network. Such implementation should also provide monitoring metrics that can be used by analysis and partitioning frameworks to be able to change at runtime the partitioning and configuration of the network itself.

REFERENCES

[1] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An overview on edge computing research," *IEEE Access*, vol. 8, pp. 85714–85728, 2020.

[2] B. Varghese and R. Buyya, "Next generation cloud computing: New trends and research directions," *Future Gener. Comput. Syst.*, vol. 79, pp. 849–861, Feb. 2018.

[3] M. Satyanarayanan *et al.*, "Edge analytics in the Internet of Things," *IEEE Pervasive Comput.*, vol. 14, no. 2, pp. 24–31, Apr.–Jun. 2015.

[4] P. G. Lopez *et al.*, "Edge-centric computing: Vision and challenges," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, pp. 37–42, Sep. 2015.

[5] F. Liu, G. Tang, Y. Li, Z. Cai, X. Zhang, and T. Zhou, "A survey on edge computing systems and tools," *Proc. IEEE*, vol. 107, no. 8, pp. 1537–1562, Aug. 2019.

[6] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, "Challenges and opportunities in edge computing," in *Proc. IEEE Int. Conf. Smart Cloud*, New York, NY, USA, 2016, pp. 20–26.

[7] S. Casale-Brunet, "Analysis and optimization of dynamic dataflow programs," Ph.D. dissertation, Dept. Sci. Techn. Eng., EPFL STI, Lausanne, Switzerland, 2015.

[8] M. M. Michalska, "Systematic design space exploration of dynamic dataflow programs for multi-core platforms," Ph.D. dissertation, Dept. Sci. Techn. Eng., EPFL STI, Lausanne, Switzerland, 2017.

[9] L. Jiao, R. Friedman, X. Fu, S. Secci, Z. Smoreda, and H. Tschofenig, "Challenges and opportunities for cloud-based computation offloading for mobile devices," in *Proc. IEEE Future Netw. Mobile Summit*, Lisbon, Portugal, Jul. 2013, pp. 1–11.

[10] J. Ren, H. Guo, C. Xu, and Y. Zhang, "Serving at the edge: A scalable IoT architecture based on transparent computing," *IEEE Netw.*, vol. 31, no. 5, pp. 96–105, Aug. 2017.

[11] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 3, pp. 1628–1656, 3rd Quart., 2017.

[12] A. Nadembega, A. S. Hafid, and R. Brisebois, "Mobility prediction model-based service migration procedure for follow me cloud to support QoS and QoE," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Kuala Lumpur, Malaysia, 2016, pp. 1–6.

[13] J. Zhang, B. Chen, Y. Zhao, X. Cheng, and F. Hu, "Data security and privacy-preserving in edge computing paradigm: Survey and open issues," *IEEE Access*, vol. 6, pp. 18209–18237, 2018.

[14] T. Wang, G. Zhang, A. Liu, M. Z. A. Bhuiyan, and Q. Jin, "A secure IoT service architecture with an efficient balance dynamics based on cloud and edge computing," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4831–4843, Jun. 2019.

[15] J. Goecks, A. Nekrutenko, and J. Taylor, "Galaxy: A comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome Biol.*, vol. 11, p. R86, Jan. 2010.

[16] J. E. Ruiz, J. Garrido, J. D. Santander-Vela, S. Sánchez-Expósito, and L. Verdes-Montenegro, "AstroTaverna—Building workflows with virtual observatory services," *Astron. Comput.*, vols. 7–8, pp. 3–11, Nov./Dec. 2014.

[17] G. Paller, E. Bezati, N. Tauan, G. Farkas, and G. Él, "Dataflow-based heterogeneous code generator for IoT applications," in *Proc. 7th Int. Conf. Model Driven Eng. Softw. Develop.*, 2019, pp. 426–432.

[18] J. Castrillon *et al.*, "Trace-based KPN composability analysis for mapping simultaneous applications to MPSoC platforms," in *Proc. Conf. Design Autom. Test Eur. (DATE)*, Dresden, Germany, Mar. 2010, pp. 753–758.

[19] A. Barker, B. Varghese, J. S. Ward, and I. Sommerville, "Academic cloud computing research: Five pitfalls and five opportunities," in *Proc. 6th USENIX Conf. Hot Topics Cloud Comput.*, 2014, p. 2.

[20] S. Sen and J. W. Janneck, "Aegis : Reliable application execution over the mobile cloud," *Procedia Comput. Sci.*, vol. 109, pp. 482–489, May 2017.

[21] E. S. Jang, M. Mattavelli, M. Preda, M. Raulet, and H. Sun, "Reconfigurable media coding: An overview," *Signal Process. Image Commun.*, vol. 28, no. 10, pp. 1215–1223, 2013.

[22] *Information Technology—MPEG Systems Technologies—Part 4: Codec Configuration Representation*, Standard ISO/IEC 23001-4:2011, 2011.

[23] M. Mattavelli, I. Amer, and M. Raulet, "The reconfigurable video coding standard [standards in a nutshell]," *IEEE Signal Process. Mag.*, vol. 27, no. 3, pp. 159–167, May 2010.

[24] K. Jerbi *et al.*, "Development and optimization of high level dataflow programs: The HEVC decoder design case," in *Proc. 48th Asilomar Conf. Signals Syst. Comput.*, Pacific Grove, GA, USA, Nov. 2014, pp. 2155–2159.

[25] C. Lucarz *et al.*, "Dataflow/actor-oriented language for the design of complex signal processing systems," in *Proc. Conf. Design Archit. Signal Image Process. (DASIP)*, Bruxelles, Belgium, Nov. 2008, pp. 1–8.
[26] M. Abid, K. Jerbi, M. Raulet, O. Déforges, and M. Abid, "System level synthesis of dataflow programs: HEVC decoder case study," in *Proc. Electron. Syst. Level Synth. Conf. (ESLsyn)*, Austin, TX, USA, 2013, pp. 1–6.
[27] A. Nandi and R. Marculescu, "System-level power/performance analysis for embedded systems design," in *Proc. 38th Annu. Design Autom. Conf.*, Las Vegas, NV, USA, 2001, pp. 599–604.
[28] J. Ceng *et al.*, "Maps: An integrated framework for MPSoC application parallelization," in *Proc. 45th Design Autom. Conf. (DAC)*, Anaheim, CA, USA, Jun. 2008, pp. 754–759.
[29] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.
[30] M. Mattavelli, J. W. Janneck, and M. Raulet, *MPEG Reconfigurable Video Coding*. Cham, Switzerland: Springer Int., 2019, pp. 213–249.
[31] J. C. Mazo and R. Leupers, *Programming Heterogeneous MPSoCs*. Cham, Switzerland: Springer, 2013.
[32] S. S. Battacharyya, E. A. Lee, and P. K. Murthy, *Software Synthesis from Dataflow Graphs*. Boston, MA, USA: Kluwer Acad. Publ., 1996.
[33] S. S. Bhattacharyya *et al.*, "OpenDF—A dataflow toolset for reconfigurable hardware and multicore systems," in *Proc. 1st Swedish Workshop Multi-Core Comput. (MCC)*, Ronneby, Sweden, Nov. 2008, pp. 1–8.
[34] S. Casale-Brunet, E. Bezati, and M. Mattavelli, "Programming models and methods for heterogeneous parallel embedded systems," in *Proc. 10th Int. Symp. Embedded Multicore Many-core Systems-on-Chip (MCSOC)*, Lyon, France, 2016, pp. 289–296.
[35] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Comput. Surveys*, vol. 36, no. 1, pp. 1–34, 2004.
[36] J. Eker and J. W. Janneck, "Dataflow programming in CAL—-Balancing expressiveness, analyzability, and implementability," in *Proc. Conf. Rec. 46th Asilomar Conf. Signals Syst. Comput. (ASILOMAR)*, Pacific Grove, CA, USA, 2012, pp. 1120–1124.
[37] G. Cedersjö and J. W. Janneck, "Tÿcho: A framework for compiling stream programs," *ACM Trans. Embedded Comput. Syst.*, vol. 18, no. 6, p. 120, Jan. 2020. [Online]. Available: https://doi.org/10.1145/3362692
[38] J. Eker and J. Janneck, "CAL language report," Dept. Elect. Eng. Comput. Sci., Univ. California Berkeley, Berkeley, CA, USA, Rep. ERL Technical Memo UCB/ERL M03/48, 2003.
[39] E. Bezati, "High-level synthesis of dataflow programs for heterogeneous platforms: Design flow tools and design space exploration," Dept. Doctoral Degree Sci., EPFL, Lausanne, Switzerland, 2015. [Online]. Available: http://infoscience.epfl.ch/record/207992
[40] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet, "Orcc: Multimedia development made easy," in *Proc. 21st ACM Int. Conf. Multimedia*, 2013, pp. 863–866.
[41] *Orcc Source Code Repositoy*. Accessed: Apr. 2021. [Online]. Available: http://github.com/orcc/orcc
[42] *Caltoopia*. Accessed: Apr. 2021. [Online]. Available: https://github.com/Caltoopia
[43] S. Casale-Brunet and M. Mattavelli, "Execution trace graph of dataflow process networks," *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 4, no. 3, pp. 340–354, Jul.–Sep. 2018.
[44] *TURNUS Source Code Repositoy*. Accessed: Apr. 2021. [Online]. Available: http://github.com/turnus
[45] S. Casale-Brunet, M. Mattavelli, and J. W. Janneck, "TURNUS: A design exploration framework for dataflow system design," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Beijing, China, 2013, p. 654.
[46] *NVIDIA CUDA Compute Unified Device Architecture*. Accessed: Apr. 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[47] *CAL Exelixi Backends Source Code Repositoy*. Accessed: Apr. 2021. [Online]. Available: https://bitbucket.org/exelixi/exelixi-backends
[48] A. Bloch, E. Bezati, and M. Mattavelli, "Programming heterogeneous CPU-GPU systems by high-level dataflow synthesis," in *Proc. IEEE Workshop Signal Process. Syst. (SiPS)*, Coimbra, Portugal, 2020, pp. 1–6.
[49] E. Bezati, S. Casale-Brunet, R. Mosqueron, and M. Mattavelli, "An heterogeneous compiler of dataflow programs for Zynq platforms," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, Brighton, U.K., May 2019, pp. 1537–1541.
[50] *Orcc-Apps Source Code Repositoy*. Accessed: Apr. 2021. [Online]. Available: https://github.com/orcc/orc-apps
[51] M. Michalska, S. Casale-Brunet, E. Bezati, and M. Mattavelli, "High-precision performance estimation of dynamic dataflow programs," in *Proc. IEEE 10th Int. Symp. Embedded Multicore Many-core Systems-on-Chip (MCSOC)*, Lyon, France, 2016, pp. 101–108.
[52] S. Casale-Brunet, E. Bezati, and M. Mattavelli, "High level synthesis of Smith-Waterman dataflow implementations," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, New Orleans, LA, USA, 2017, pp. 1173–1177.

AURELIEN BLOCH received the B.S. and M.S. degrees in computer science from École Polytechnique Fédérale de Lausanne (EPFL) in 2015 and 2018, respectively, where he is currently pursuing the Ph.D. degree with EPFL SCI-STI-MM Group. His current research at EPFL includes methodologies and tools for designing complex software systems targeting heterogeneous architectures in the context of signal processing and digital media.

SIMONE CASALE-BRUNET (Member, IEEE) received the B.S. degree (Hons.) in electrical engineering and the M.S. degree (Hons.) in mechatronics engineering from the Politecnico di Torino, Italy, in 2008 and 2010, respectively, and the Ph.D. degree in electrical engineering from the Ecole Polytechnique Federale de Lausanne (EPFL), Switzerland, in 2015, where he is a Research Scientist with EPFL SCI-STI-MM Group. His current research activities at EPFL include methodologies for specification, modeling and synthesis of complex systems, architectures for digital media coding and processing, applications of combinatorial optimization to signal processing, and the compression and processing of genome sequencing data.

MARCO MATTAVELLI (Member, IEEE) received the Ph.D. degree from the Ecole Polytechnique Federale de Lausanne (EPFL), in 1996. He started his research activity with the Philips Research Laboratories, Eindhoven, in 1988, on channel and source coding for optical recording, electronic photography, and signal processing of HDTV. In 1991, he joined EPFL. He has been the Chairman of a Group of MPEG ISO/IEC Standardization Committee. He has authored more than 200 publications. His current research activities at EPFL include methodologies for specification, modeling and syntesis of complex systems, architectures for digital media coding and processing, applications of combinatorial optimization to signal processing, and the compression and processing of genome sequencing data. For his work, he received four ISO/IEC Awards. He has served as an invited editor and associate editor for several IEEE conferences and journals.