Journal of
Low Power Electronics and Applications

Article

Dynamic SIMD Parallel Execution on GPU from High-Level Dataflow Synthesis †

Aurelien Bloch *, Simone Casale-Brunet and Marco Mattavelli

EPFL SCI-STI-MM, École Polytechnique Fédérale de Lausanne, CH-1015 Lausanne, Switzerland; simone.casalebrunet@epfl.ch (S.C.-B.); marco.mattavelli@epfl.ch (M.M.)
* Correspondence: aurelien.bloch@epfl.ch

† This paper is an extended version of our paper published in 14th IEEE MCSoC 2021.

Abstract: Developing and fine-tuning software programs for heterogeneous hardware such as CPU/GPU processing platforms comprise a highly complex endeavor that demands considerable time and effort of software engineers and requires evaluating various fundamental components and features of both the design and of the platform to maximize the overall performance. The dataflow programming approach has proven to be an appropriate methodology for reaching such a difficult and complex goal for the intrinsic portability and the possibility of easily decomposing a network of actors on different processing units of the heterogeneous hardware. Nonetheless, such a design method might not be enough on its own to achieve the desired performance goals, and supporting tools are useful to be able to efficiently explore the design space so as to optimize the desired performance objectives. This article presents a methodology composed of several stages for enhancing the performance of dataflow software developed in RVC-CAL and generating low-level implementations to be executed on GPU/CPU heterogeneous hardware platforms. The stages are composed of a method for the efficient scheduling of parallel CUDA partitions, an optimization of the performance of the data transmission tasks across computing kernels, and the exploitation of dynamic programming for introducing SIMD-capable graphics processing unit systems. The methodology is validated on both the quantitative and qualitative side by means of dataflow software application examples running on platforms according to various different mapping configurations.

Keywords: heterogeneous systems; GPU programming; source-to-source compiler; parallel computing; SIMD; RVC-CAL; dynamic dataflow programs

check for updates

Citation: Bloch, A.; Casale-Brunet, S.; Mattavelli, M. Dynamic SIMD Parallel Execution on GPU from High-Level Dataflow Synthesis. *J. Low Power Electron. Appl.* **2022**, *12*, 40. https://doi.org/10.3390/jlpea12030040

Academic Editors: Sanghamitra Roy and Andrea Acquaviva

Received: 10 March 2022
Accepted: 11 July 2022
Published: 17 July 2022

1. Introduction

Heterogeneous computing systems are becoming increasingly popular in industry and academia, thanks in part to the fact that transistor shrinkage has slowed down. Trends such as the scale-out of data centers are starting to reach technical limitations. Here, in fact, energy consumption becomes a severe issue and is now a serious subject of study and optimization. Thus, the ever-increasing need for more powerful computing systems drives the development of more diverse specialized platform designs, which include processing architectures (e.g., CPU arrays, GPUs, FPGAs) that are able to be more efficient at performing specialized computing operations. This trend is also driven by the industry, in which major semiconductor companies (e.g., Nvidia, Intel) are releasing more advanced heterogeneous hardware that complies with increasingly stringent energy policies [1]. GPUs offer the ability to achieve massive, yet energy-efficient parallelism. However, exploiting this type of parallelism requires that the code be carefully structured to address the limitations of the SIMT execution model. Several methods have been developed in the literature, to avoid the situation in which adapting the code to specific platforms results in being an operation that has to be performed fully manually. For example, a hybrid scheduling approach, which the authors call inspector–executor, applied to the problem of six tree traversal algorithms is illustrated in [2]. The GPU partially executes a task and sends execution information to the CPU, then the CPU uses this information to perform the scheduling. These techniques have been effectively applied in problems such as community detection in weighted networks [3] and in the multi-product dynamic lot sizing problem [4]. All these works, however, start with implementations specifically targeted for hybrid CPU/GPU platforms. An increasingly used methodology is to employ high-level dataflow programming languages that are not dependent on the target platform. This type of high-level programming enables portable development approaches that are architecture-independent and can be efficiently leveraged to analyze and explore the design space. In fact, dataflow application software inherently reveals the intrinsic parallelism available in the data being processed by the algorithms. This is done in a systematic fashion by exploring the design configuration of the actors' networks—such as the mapping of computational kernel to the processing element or the allocation of communication buffers. It is then possible to identify which high-performance design points simultaneously satisfy the design constraints [5–7]. The exploration of the design space is effectively achieved without the necessity to test every design point on the hardware system; instead, it relies on a simulation and performance estimation model. These simulation models typically use profiling measures obtained from a sample set of (partial) executions of the application obtained on the platform. In other words, this method greatly reduces the effort of exploring configuration points, as not all configuration points are tested on the real platform, but only at the simulated level. Such an approach significantly facilitates and speeds up portability to different heterogeneous platforms and, by using specific algorithms, allows obtaining results close to optimum [5]. When using high-level and platform-independent dataflow design techniques and targeting CPU/GPU platforms, the main challenge is that the computational nodes of the dataflow networks (also called actors) are formed with distinct sequential functions that do not strictly correspond to the Single-Instruction–Multiple-Data (SIMD) nature of the typical architecture of GPUs. The appropriate use of resources is a prerequisite of the exploitation of a SIMD-type parallelization, which then results in a dramatic improvement in the performance of the entire application.

There are several novelties introduced by this work. The first is the definition of a new methodology to exploit the intra-kernel parallelization available using parallel CUDA constructs for the execution of actions of dataflow actors. The second is the extension of both the design space exploration model defined by the authors of this work and the extension of the open-source toolbox capable of synthesizing low-level code for heterogeneous CPU and GPU platforms. To this end, the methodology already defined in [8] was significantly extended allowing automatically synthesizing a C++/CUDA parallel version for every actors' actions, all taking full advantage of SIMD parallelization techniques. All the innovative contributions of this article can be summarized as follows:

- The definition of a new FIFO communication channel to improve the communication performance between two computing elements even on different hardware (i.e., GPU and CPU). This was made possible by exploiting the architecture and programming APIs available with the new GPU implementations;
- A new methodology for GPU partitioning of actors synthesized with the CUDA formalism. This new methodology completely removes the need to use a CPU kernel (actively waiting) to control the execution of GPU-mapped actors;
- The dynamic programming model of actors in a dataflow program is leveraged to exploit intra-kernel parallelization and take full advantage of the SIMD architecture of GPUs;
- A low-level code-synthesizing methodology to leverage dynamic actor reprogramming, where internal actions are implemented leveraging SIMD parallelization.

The article is structured as follows: In Section 2 a summary of the dataflow computational model is illustrated. The main components, their properties, and the main advantages of this computational model are defined. In addition, an overview of the dataflow programming language used in this paper, in this case RVC-CAL, is given. Subsequently, the state-of-the-art of techniques implementing and optimizing dataflow applications in parallel CPU and GPU architectures is discussed in Section 3. The main contributions of this work are illustrated in Section 4 and supported by the experimental results of Section 5. Here, the approach is validated by making use of two case studies that differ in computational complexity and the nature of parallelization. An FIR filter and a JPEG decoder are used: both designs are implemented using RVC-CAL as the programming language. The results focus on demonstrating how the SIMD optimizations proposed in this work are very effective. Section 6 concludes the paper and provides future research directions.

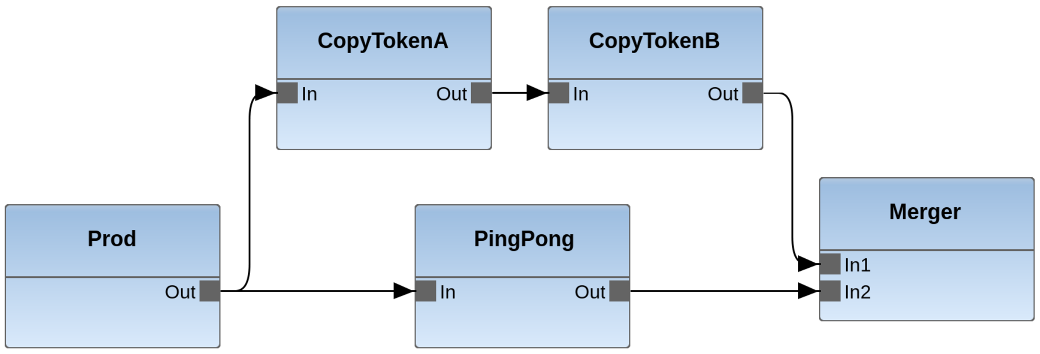
2. Dataflow Model of Computations

A dataflow program is a network of computational kernels, also referred to as actors. These actors can only communicate with each other by exchanging data through point-to-point transmission vectors, called buffers. Buffers depict the routes along which the tokens (data emitted) can flow. Tokens, which are atomic units of data, are placed and removed from the buffers according to the computational rules of individual actors, called firing rules.

The work presented in this paper addresses the Dynamic Dataflow (DDF) class of dataflow programs. In fact, this is the most expressive computation class defining the execution of a dataflow program. This class allows defining a dynamic token consumption/production rate, so that it is not predefined, but depends on the value of the network state and/or on the value carried by the tokens. As a consequence, dataflow networks of the DDF type are not analyzable at compile time [9].

Over the past decade, a large number of different programming languages have emerged with the aim of modeling the semantics of dataflow programs [10]. Imperative programming languages were augmented by including parallel directives (e.g., Java, Python, C/C++), and actual native dataflow languages were newly specified (e.g., Esterel, Ptolemy). Considering these variegated language extension environments, RVC-CAL [11] can be considered the only fully formally standardized programming language following the dataflow model, which is capable of modeling DDF networks. An RVC-CAL actor is defined as a collection of atomic methods, known as actions, together with a collection of encapsulated state variables for which access and modification are prohibited from neighboring actors in the same network. When an actor is running, a single action is selected at a time each time. That is, each actor cannot execute several actions at the same time, but only one. The selection of the action to execute can be done considering the input value of the tokens and/or the value of the internal variables of the actor. The synthesis from this high-level programming language to low-level code for heterogeneous platforms can be done using many open-source compilers, such as Open RVC-CAL Compiler (ORCC) [12,13], Tÿcho [14], StreamBlocks [15], and Caltoopia [16].

As an example, Figure 1 shows an RVC-CAL dataflow program. Figure 1a depicts the graphical representation of the program, where the blue boxes are the actors, while the directed arcs represent the buffers. Specifically, this network consists of five actors (i.e., *Merger*, *PingPong*, *CopyTokensB*, *CopyTokensA*, and *Prod*) and five buffers. Figure 1b shows the RVC-CAL implementation of the Producer actor: this is composed of a single action that produces a token for each execution (firing), during which it increments the internal variable *counter* each time. A guard condition prevents the action from being followed more than four times. In the PingPong actor code shown in Figure 1d, the schedule construct is used to define an internal finite state machine (FSM), which defines how the execution of action *pp1* must be alternated with the execution of action *pp2*.



(a) Illustration of the RVC-CAL dataflow network example.

```
actor Prod () ==> int Out:
int cnt := 0;

produce: action ==> Out:[cnt]
guard cnt < 4
do  cnt := cnt + 1; end
end
```

(b) Source code of the Prod.cal actor

```
actor CopyToken (String cpName) int In ==> int Out:
copy: action In:[token] ==> Out:[token] end
end
```

(c) Source code of the CopyToken.cal actor

```
actor PingPong () int In ==> int Out:

ping: action In:[token] ==> Out:[token]
do println("Ping:" + token); end

pong: action In:[token] ==> Out:[-token]
do println("Pong:" + token); end

schedule fsm s_ping:
s_ping(ping) --> s_pong;
s_pong(pong) --> s_ping;
end
end
```

(d) Source code of the PingPong.cal actor

Figure 1. *Cont.*

```
actor Merger () int In1 , int In2 ==> :
int cnt := 0;

merge: action In1:[ token1 ], In2:[ token2 ] ==>
do
println("Merger(" + cnt + "):" + token1 + ";" + token2);
cnt := cnt + 1;
end
end
```

(**e**) Source code of the Merger.cal actor

Figure 1. Example illustrating the main properties of the RVC-CAL program and how it is structured. This example contains five actors (i.e., *Prod*, *CopyTokenA*, *CopyTokenB*, *PingPong*, and *Merger*).

3. Related Work

Increasing attention has been paid over the past few years to dataflow programming in the field of programming multicore and heterogeneous computing platforms [17–22]. As outlined by the various studies that can be found in the literature, one of the most prominent use cases is the execution of dataflow programs on GPU platforms. In this direction, refer for example to [17–22]. Specific to dynamic programs and implemented on GPU platforms using RVC-CAL, the works presented in [23] and in [24] show two different approaches on how to use OpenCL APIs to implement, partition, and subsequently execute RVC-CAL SW programs. In the first case, an OpenCL implementation is generated from the Intermediate Representation (IR) provided by the Orcc compiler [13], whereas in the second, the IR of the Distributed Application Layer (DAL) [25]) is used for the final synthesis. In both works, the first step is to classify each actor that composes the program (in other words, it is determined whether an actor is static, cyclo-static, or dynamic); then, only the narrowest Static Dataflow Actors (SDFs) (i.e., those for which execution and scheduling behavior can be computed during the compile-time design stage) are executed on the GPU platform, while the dynamic actors [26] are mapped to the available CPU cores. A similar approach is illustrated in [27], where RVC-CAL dataflow programs are first transformed into a high-level SYCL [28] representation. SYCL is a cross-platform C++ abstraction layer that allows source code for heterogeneous processors to be expressed using the OpenCL API. The approach illustrated in [27] is also based on using the SIMD nature of GPUs to run multiple consecutive executions of the same action in parallel when there are data available for its execution.

The main limitations of the available state-of-the-art solutions described above are mainly due to the fact that only the computational part of the actors (actions firings) is performed on the GPUs, while the actor scheduling part (action selection) is performed on the CPUs. This implementation approach has two drawbacks. The first drawback is that, to the best of the authors' knowledge, in the presented design, they are scheduling only the static actors on the GPU; the dynamic actors are executed only on the CPU. The second drawback is that the CPU is constantly working to schedule the next tasks for the GPU. The work presented in this paper aims at removing these limitations, thus allowing having a scheduling choice made directly on the GPU. This solution releases the CPU from computational tasks and allows it to use its additional computational resources to execute other parts of the program concurrently with the GPU. In addition, the amount of information exchanged between the GPU and CPU is largely reduced. Moreover, in this approach, the mapping of GPU actors can be applied to any type of execution models, including DPNs, and is not limited to static actors or actor networks. In other words, this approach makes it possible to map a dataflow DPN network program onto a GPU platform independently of its dynamic or non-dynamic behavior. Table 1 resumes the main differences of the main comparable solutions.

Table 1. Summary of related papers' comparison.

	This Work	**[23]**	**[24]**	**[27]**
static	CPU/GPU	CPU/GPU	CPU/GPU	CPU/GPU
cyclo-static	CPU/GPU	CPU	CPU	CPU
dynamic	CPU/GPU	CPU	CPU	CPU

4. Design Process and Development

In the following section, it is explained how the methodologies developed in previous research (considering also the ones designed and implemented by the authors of this paper) were extended to more efficiently support the systematic partitioning of DPN software applications. To achieve this goal, the tool set illustrated in Figure 2 was utilized. Starting from the top, the figure shows the high-level representation of the dataflow program (CAL), the network setting (XDF), the configuration files that provide partitioning information, and buffer sizes (BXDF). These represent the input to the ORCC compiler, which uses the Exelixi CUDA backend described in [8,29,30] to automatically write the respective CUDA/C++ low-level software. This code must then subsequently be compiled using Nvidia CUDA Compiler (NVCC) to obtain the program binary. Furthermore, the following section describes a new design for the FIFO buffers that are utilized for communication between neighboring actors. Along with this, a novel scheduling strategy for actors executing on GPUs is presented. Lastly, it is shown how SIMD parallelization methodologies are used to accelerate the runtime of dataflow actions to improve execution performance.

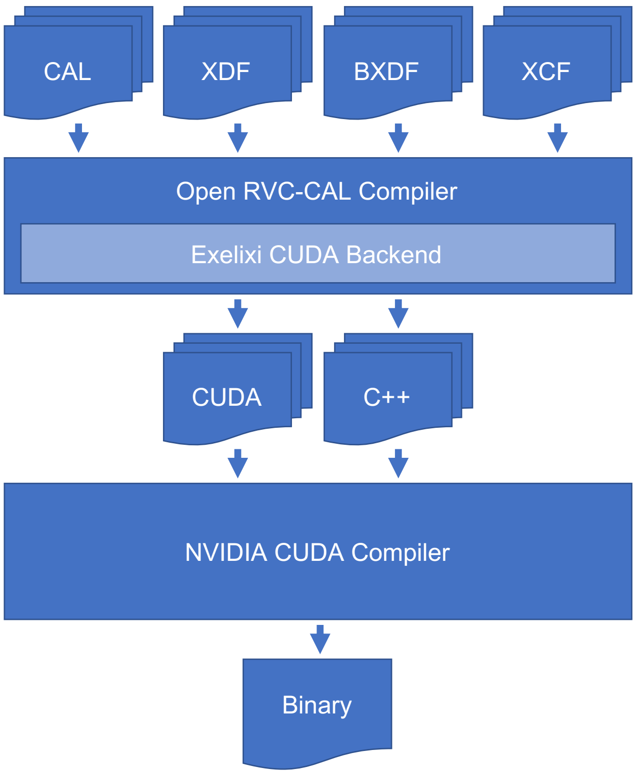


Figure 2. Workflow of the design, optimization, and implementation of a dynamic dataflow application using RVC-CAL as the programming language on heterogeneous CPU and GPU platforms.

4.1. GPU/CPU Data Communication

In a previous work of the authors of this paper [8], the data transfer between GPUs and CPUs was developed by means of a *CudaHostFIFO*. This specific communication channel is depicted in Figure 3 with a blue ②. Such a FIFO was particularly created to move data across the GPU and CPU boundary. The implementation is based on a twin FIFO buffer that is allocated on each platform using GPU RAM and RAM that is being mirrored using data copying and synchronization utilizing the CUDA APIs. The primary drawback with the presented method is the fact that an extensive number of calls to the CUDA runtime libraries is being made, therefore raising the runtime overhead.

In this article, a novel method for a cross-hardware FIFO buffer was designed by using the capabilities offered by the new CUDA APIs and architecture capacities summarized in Figure 3 with a blue ③. With this goal in mind, the memory allocated on the CPU platform is a pinned memory. This signifies that the operating system does not swap allocations or transfer them to another physical memory location. These memory allocations are subsequently registered in the virtual address memory management of the GPU from which translated pointers are obtained. With this approach, different pointers are dereferenced if the data are manipulated from the GPU system or the CPU system. The main benefit of this method is that it removes the necessity of any software API or synchronization calls. In fact, memory accesses are performed automatically by the hardware platform. It is important to note that this solution does not change the rest of the computation model or the way FIFO buffers are used.

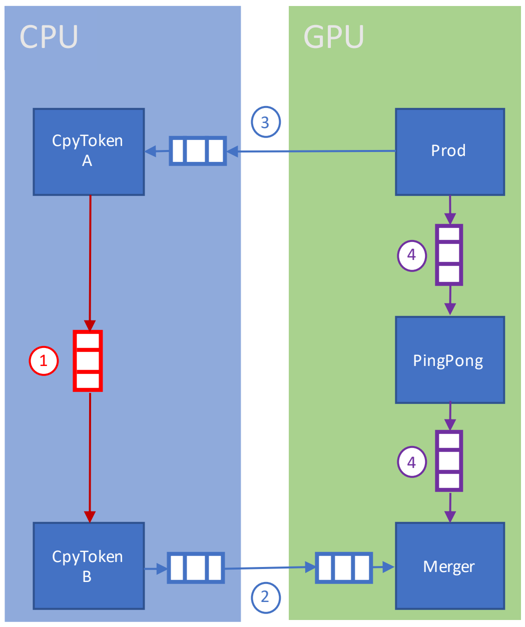


Figure 3. Partitioning model and implementation of a program between the GPU and CPU. The FIFOs used are of three different types: with 1 is indicated the host FIFO, with 2 a host-to-device FIFO, with 3 a device-to-host FIFO, and with 4 a device FIFO. In this particular example, the program illustrated in Figure 1 is considered.

4.2. GPU Partitions

In a previous work of the authors of this paper [8], a CPU thread was used exclusively to schedule all the actors that were mapped to the GPU (i.e., GPU partitions). This entails running each actor's kernel intended to be executed on GPUs in parallel and spinning until they terminate. If any actors made computational advancements, then every actor is executed again until no actors perform forward processing for a determined elapsed time. Only from this moment, the GPU partitions are considered to have completed their execution and the CPU thread is freed. This is an active-wait (spin–wait)-type system, which is neither energy nor computationally efficient. Its advantages, however, are the ease of implementation and the possibility to know at any time the complete state of the entire dataflow network.

The goal of this work is to make the execution of each GPU-mapped actor autonomous. In other words, using the same formalism described in [8], it acts as if each CUDA partition contains only one actor. Every actor is supposed to terminate itself by monitoring the termination reached by the application software. This indicates that the GPU partitions could be considered during the entire time the application program is executing as totally independent. So as to reach this goal, after all the instantiation and initialization phases, the main program thread waits for all CPU and GPU partitions to stop before terminating the application. The main mechanism used for this is that each partition (CPU or GPU) keeps track of the progress made by its actors. If, after a fixed amount of time, no actors have made any progress (i.e, no actions were fired), the partition terminates.

Regarding in more detail how the new GPU partition design is implemented, the low-level scheduling generated by the CAL backend (which we call *actionSelection*) was implemented as a long-running kernel. Figure 1 presents a simplified example of how this function of *actionSelection* was implemented. From Line 25, it can be noticed that the *status* array is set every time an action is fired. This array is accessible by every other CUDA actor and is utilized to notify the other actors, indicating that one of them just performed further processing. The *checkStatus* function-call in Line 30 is in charge of verifying the status of the other actors. If zero actors have produced any work after a certain fixed period of time (*waitPeriod*), the actors terminate. The software application is closed after all CPU-side actors and all GPU-side actors are done executing.

4.3. SIMD Parallelization

Generating efficient low-level SIMD-type code is a prerequisite to fully exploit the hardware available in modern GPUs. Without this, on *Nvidia* GPUs for instance and because a CUDA core is a SIMD architecture composed of 32 threads, a single thread out of the 32 possible will be utilized, wasting a substantial part of the potential resources. In order to reach this objective, though, it is mandatory to identify what are the limitations of the computation model used to program the platform. In the specific case of RVC-CAL, the intrinsic behavior of an actor is to consume one or more tokens from the input buffers, execute the computational part defined by the action, modify the internal state, and then, produce one or more tokens in the output buffers. This approach makes the implementation not yet ready for SIMD-like parallelization. However, it is possible to improve the actor execution performance if the order of tokens within the actor communication channels is preserved and the actor state dependencies of the model execution are respected.

Unlike previous works, where the *actionSelection* was executed on a single CUDA thread and the actions were simple function calls, in this work, a new method is used. The dynamic parallelism that is required to execute the *actionSelection* function is obtained by using a second dedicated kernel for the actions. This approach allows obtaining a SIMD parallel execution using several instances of the same action at the same time. As an illustrative example of the approach developed in this paper, a summary of the implementation is shown in Listing 1. Here, an improved implementation of *actionSelection* is proposed. It can be seen from Line 23 how, for this particular example, 512 actions are executed in parallel. Another point worth noticing is how the handling of read and write addresses in FIFOs is handled within the *actionSelection* function instead of in the action implementation itself. This makes possible the *actionSelection* to be called only once and sequentially. Moreover, it can be observed that in Line 21, the read address is obtained by providing as a FIFO sizing parameter the number of tokens that will be used at each call (in the specific case of the proposed example, this number is 64 tokens for the number of threads). Proper sizing is particularly important to ensure that memory is aligned so that a specific amount of data can be accessed in consecutive addresses. It is possible to have DPN networks in which sequential actors and parallelized actors coexist while maintaining the sequential semantics of the FIFO buffers. This can be achieved by implementing a token id and thread id mapping to be used within an action in order to index the read and/or write addresses.

Listing 1. Simplified implementation of a *CUDA* action selection function for an actor of the *idct* design illustrated in Figure 4.

```
void actionSelection(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);
}
```

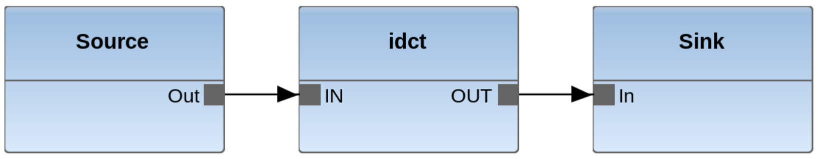


Figure 4. Illustration of the test RVC-CAL IDCT dataflow network.

4.4. Dynamic SIMD Parallelization

In this section, a methodology developed to implement applications by a number of threads that evolve at runtime is described.

Implementation

The dynamic SIMD parallelization optimization is based on generating a number of parallel executions of the same action that dynamically evolve over the runtime of the application, with the objective of maximizing the performance and the GPU resource utilization. To achieve such an objective, during the code generation phase, each action with a *parallel* flag gets assigned a pair of integer $\{bl, th\}$ with bl being the number of CUDA thread blocks and th the number of threads per block. From there, everything needs to be carefully parametrized with this dynamic pair of variables, that is the action CUDA kernel launch parameter, the pre-allocation of the FIFO buffer token slots for the corresponding input or output ports, and the test for available space for writing and available tokens to be read. Finally, the dynamic change needs to be synchronized with the end of the main inner loop to avoid size mismatches.

It has to be noted that the maximum amount of thread that a FIFO can access in parallel is correlated with the size of the FIFO *threshold* and the number of consumed/produced tokens per action firing. Indeed, for a FIFO buffer of size *"size"*, a memory of size "$size + threashold$" is allocated. This is done because to achieve efficient execution performance, the SIMD threads need to access consecutive memory locations. Thus, the threshold should be at least $Nb_{thread} * Nb_{tokens}$ with Nb_{thread} being the maximum number of SIMD threads an action can be executed with and Nb_{tokens} being the number of tokens an action produces/consumes to/from this FIFO buffer. When using the static SIMD mode, this concern is not a relevant issue as the Exelixi CUDA backend generates the proper size according to the flag set by the developer, but for the dynamic SIMD case, such a number needs to be sufficiently large to generate memory sizes enabling effective parallelization opportunities.

Figure 5 shows an example of an internal representation of a FIFO buffer of size 12 tokens ($t1 - t12$) where an action consumes 3 tokens per firing and is executed with 4 SIMD threads ($th1 - th4$) in parallel. In the example, the FIFO buffer needs a *threshold* of minimum size $3 * 4 = 12$.

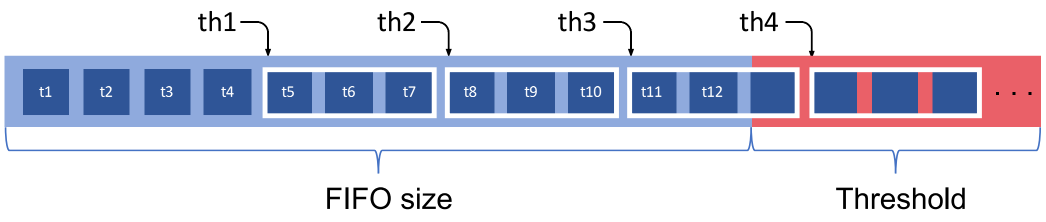


Figure 5. Representation of the SIMD parallel read/write of a FIFO buffer.

5. Experimental Evaluation

5.1. Experimental Hardware and Software Platform

The processing platforms used to run the experiments have the following specification:

- System 1: GeForce GTX 1660 SUPER Nvidia GPU coupled with an Intel Skylake i5-6600. The GPU graphic co-processor is equipped with 6 GB of memory, whereas the CPU main processing platform is equipped with 16 GBytes of DDR4 RAM.
- System 2: GeForce RTX 3080 Ti Nvidia GPU coupled with an AMD Threadripper 3990X. The GPU graphic co-processor is equipped with 12 GB of memory, whereas the CPU main processing platform is equipped with two 256 GBytes of DDR4 RAM.

They both run CUDA version 11.6.2 used for the graphic software library.

5.2. SIMD Parallelization Capabilities' Evaluation

Two application programs were selected with the objective of evaluating the new methodology introduced in this work. When a new technology is used to generate executable code from a high-level dataflow program, two elements need to be considered. The first element is to verify that the generated executable code is semantically correct and equivalent to the dataflow source code. The second element is obviously to measure the obtained performance increase versus an alternative.

5.2.1. Experiments with an IDCT Application

The parallelization results obtainable by applying the new approach described in the paper to a computationally intensive actor are described in this section. So as to isolate the transformation and better evaluate the potential results, the test application is simply composed by a network of three actors. Two actors provide the necessary input *Source* and output *Sink* data flow, and the central actor contains the implementation of a compliant *idct* algorithm version (refer to Figure 4 for an illustration of the dataflow network). For this simple network, two partitioning configurations were tested. The first is a CPU-only mapping, employing one thread for each actor and a total of three threads. A second partitioning is a GPU-only mapping for which the CUDA kernel of the action that is executed inside the *idct* actor is executed on a grid of two blocks employing 512 threads per block. Both test configurations were implemented with a buffer of the same size. The result of the experiment is that the GPU configuration, when compared to the configuration in which all actors are mapped only on the CPU platform, can complete the data processing tasks in less than half the time. The detailed results can be seen in Table 2. Such results demonstrate that if a sufficient throughput of data can be supplied to the CUDA-implemented actors, the proposed methodology does indeed provide a relevant performance boost over implementations relying only on multicore CPU processing.

Table 2. Speedup results with statistics for the the RVC-CAL IDCT application using 10 executions.

	CPU				**GPU**				**Speedup**
	Min	**Mean**	**Max**	**Var**	**Min**	**Mean**	**Max**	**Var**	
System 1	9.23	9.31	9.42	2.90×10^{-3}	4.36	4.60	4.77	1.40×10^{-2}	2.02
System 2	8.50	12.6	18.0	1.39×10^{1}	5.72	5.73	5.74	3.22×10^{-5}	2.18

5.2.2. RVC-CAL JPEG Decoder

Figure 6 reports the dataflow network of the JPEG decoder application program that was chosen for the experiments. The network is composed by six actors. The JPEG decoder is a well-known application, and the corresponding dataflow source code is publicly available in the following web repository [31]. In this second experiment, the *idct2d* actor parallelized as in the tests described above was used in the dataflow network implementing the full JPEG decoding algorithm. In this experiment, the application program is essentially used to validate the semantic correctness of the generated code for a generic dataflow program. The application is sufficiently complex, and it was not explicitly designed and optimized for exploring efficient GPU parallelization options. In terms of performance, Table 3 reports the experimental results in terms of the obtained performance, respectively when the *idct2d* actor is sequentially executed on the GPU (such an implementation is consistent with the methodology previously developed and reported in [8]) and when all other actors processing tasks are executed on the CPU partition. The second experiment corresponds to the improved methodology in which the *idct2d* actor is executed in parallel on the GPU platform. It can be observed that this experiment employing the parallel GPU implementation results outperforms by a speedup of 9.67 or 18.3 the sequential implementation of the first experiment depending on the hardware system.

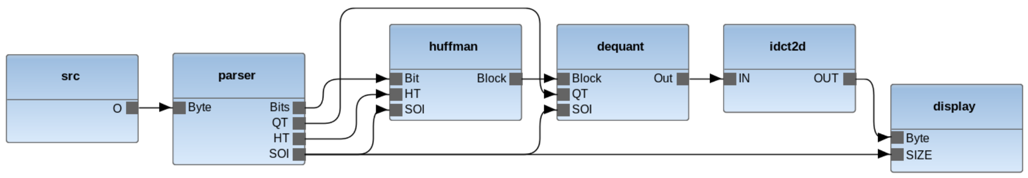


Figure 6. RVC-CAL JPEG decoder dataflow network: actors and communication FIFO buffers.

Table 3. Frame rate and speedup results for the RVC-CAL JPEG decoder.

	Frame Rate (image/s)								Speedup
	Sequential GPU				Parallel GPU				
	Min	Mean	Max	Var	Min	Mean	Max	Var	
System 1	0.24	0.24	0.24	7.63×10^{-8}	2.28	2.28	2.28	4.06×10^{-7}	9.44
System 2	0.28	0.29	0.29	2.38×10^{-7}	5.16	5.19	5.24	8.17×10^{-4}	18.20

5.3. CPU/GPU Data Exchange Performance

This sub-section reports the performance of two application programs employed to assess the improvements obtained by only changing the inter-partition communication (CPU/GPU and GPU/CPU) implementations by using the new FIFO implementation introduced in the previous sections of this work.

5.3.1. RVC-CAL FIR Filter

Figure 7 reports the dataflow network of a simple FIR filter application program. The network is composed of 13 actors. The corresponding source code is publicly available in the web repository accessible at this URL [31].

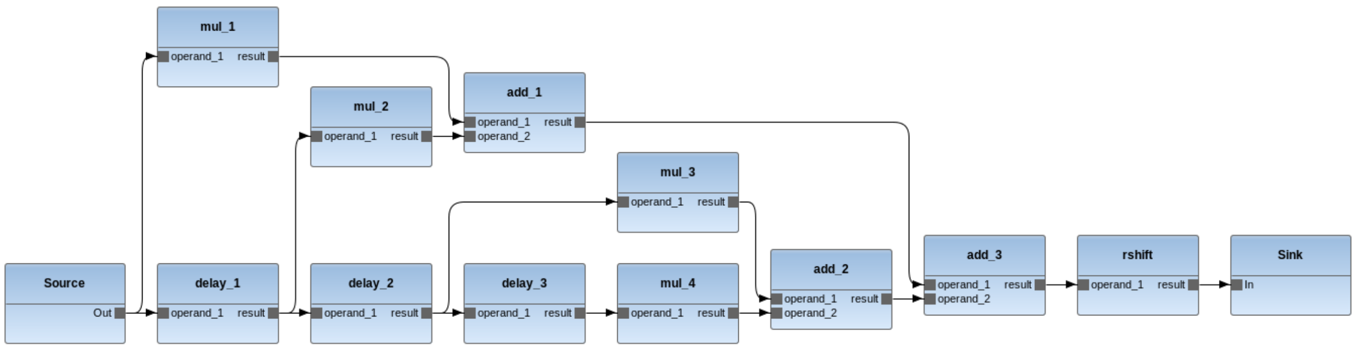
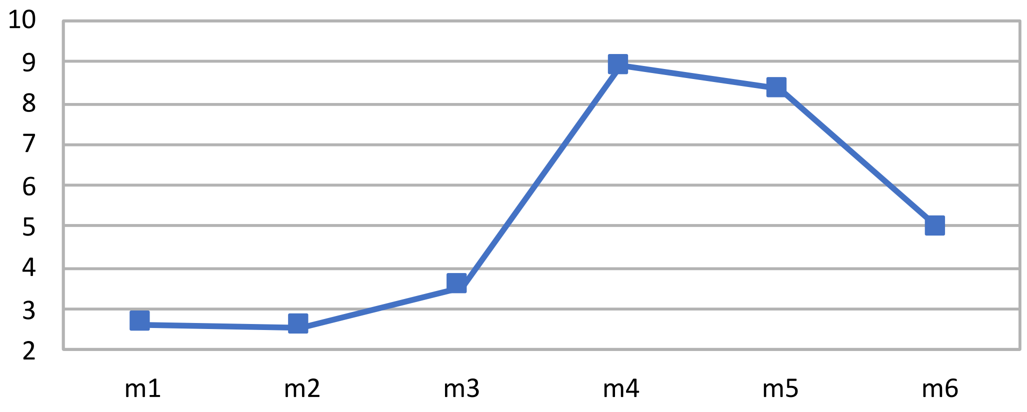
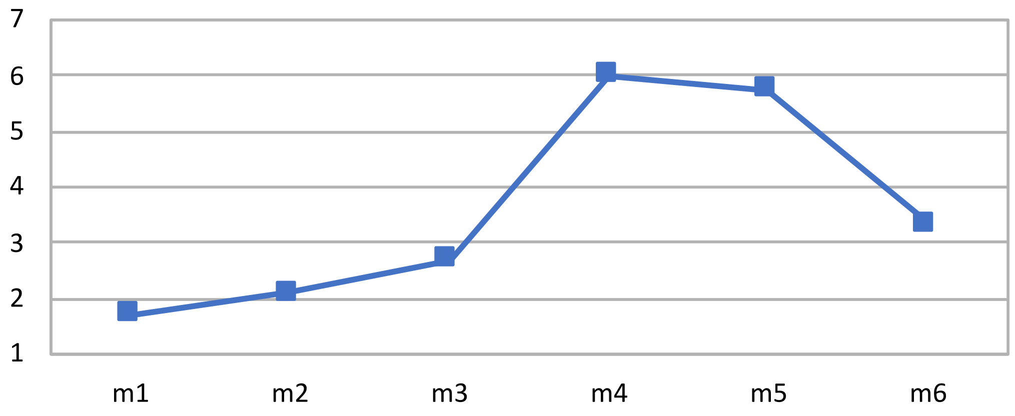


Figure 7. RVC-CAL FIR filter dataflow network: actors and communication FIFO buffers.

Figure 8, Tables 4 and 5 reports the performance speedup that the new implementation of the FIFO buffer provides versus the state-of-the-art implementation. Since the objective is to measure the performance of CPU/GPU data throughput, different configurations/mappings of a relatively complex data flow network of actors implementing simple internal processing are an appropriate validation test. Therefore, for this experiment, the overall performance of an FIR application was evaluated. The performance of six randomly chosen mappings (see Table 6) on the two different systems of the FIR network actors for the CPU or GPU platform are reported. The results were compared to the performance of the implementation using the previous inter-platform communication mechanism and the same mapping/partition. It can be observed that, in this case, the improvement of the processing speed ranges from 1.70 up to 8.91.



(**a**) System 1



(**b**) System 2

Figure 8. Speedup results for the RVC-CAL FIR filter on the two different hardware systems. 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.

Table 4. Statistics of the results depicted in Figure 8a using 10 executions.

	New FIFO				**Old FIFO**				**Speedup**
	Min	**Mean**	**Max**	**Var**	**Min**	**Mean**	**Max**	**Var**	
m1	1.36	1.37	1.38	1.00×10^{-4}	3.56	3.60	3.63	1.23×10^{-3}	2.63
m2	1.31	1.34	1.35	4.33×10^{-4}	3.40	3.42	3.45	8.33×10^{-4}	2.56
m3	1.37	1.38	1.41	5.33×10^{-4}	4.90	4.91	4.92	1.33×10^{-4}	3.55
m4	5.73	5.76	5.80	1.43×10^{-3}	51.03	51.27	51.65	1.09×10^{-1}	8.91
m5	5.99	6.16	6.26	2.26×10^{-2}	51.23	51.30	51.37	4.90×10^{-3}	8.32
m6	5.79	5.80	5.82	3.00×10^{-4}	28.55	28.63	28.71	6.43×10^{-3}	4.94

Table 5. Statistics of the results depicted in Figure 8b using 10 executions.

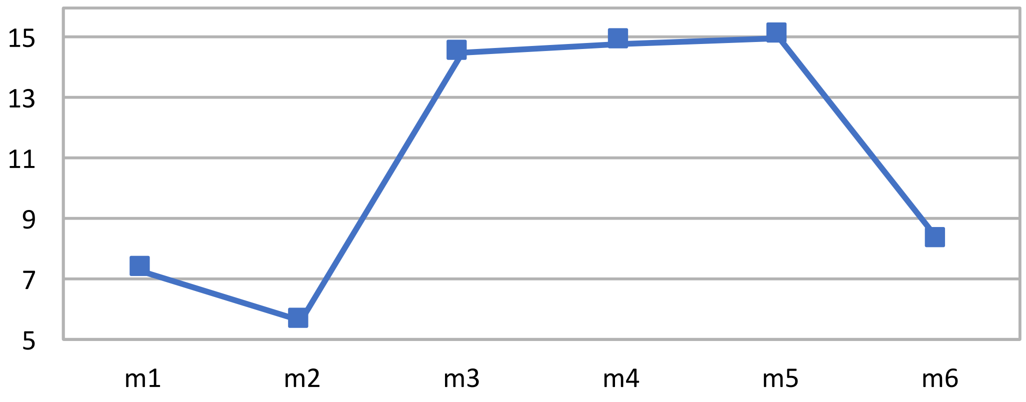
	New FIFO				Old FIFO				Speedup
	Min	Mean	Max	Var	Min	Mean	Max	Var	
m1	1.25	1.26	1.27	1.33×10^{-4}	2.11	2.13	2.17	1.03×10^{-3}	1.70
m2	1.25	1.28	1.31	9.33×10^{-4}	2.65	2.67	2.68	3.00×10^{-4}	2.09
m3	1.30	1.31	1.32	1.00×10^{-4}	3.48	3.52	3.56	1.63×10^{-3}	2.68
m4	4.98	5.04	5.17	1.20×10^{-2}	30.10	30.24	30.42	2.72×10^{-2}	6.00
m5	5.23	5.26	5.31	1.73×10^{-3}	30.00	30.20	30.51	7.52×10^{-2}	5.74
m6	5.15	5.19	5.26	3.70×10^{-3}	17.16	17.31	17.53	3.72×10^{-2}	3.34

Table 6. The different mapping settings of the FIR implementation that are used in the results.

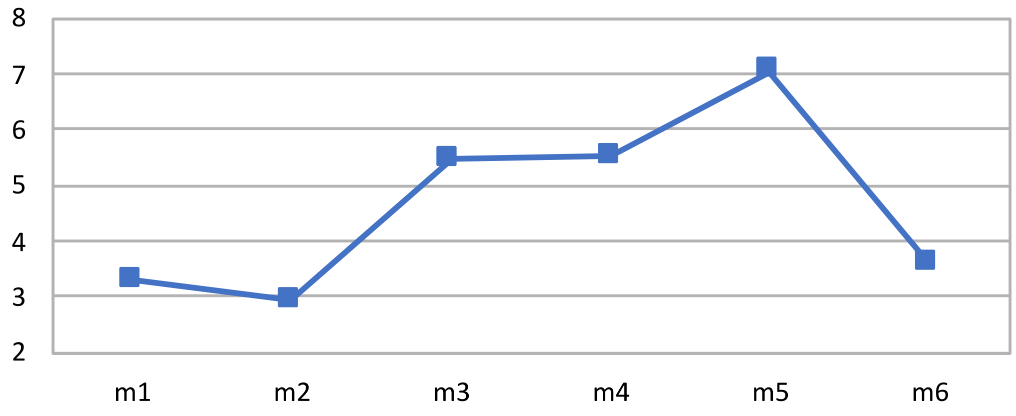
	CPU	GPU
m1	Source, Sink	delay_1, delay_2, delay3, mul_1, mul_2, mul_3, mul_4, add_1, add_2, add_3, rshift
m2	Source, Sink, delay_1, mul_1, mul_2, add_1, add_2, rshift	delay_2, delay3, mul_3, mul_4, add_2
m3	Source, Sink, mul_1, add_3, rshift	delay_1, delay_2, delay3, mul_2, mul_3, mul_4, add_1, add_2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

5.3.2. RVC-CAL JPEG Decoder

Like the results reported in the previous sub-sections, Figure 9, Tables 7 and 8 reports reports the speedup obtained by the new FIFO implementation in the case of six random configurations of the JPEG decoder already introduced in Section 5.2.2. For this software app, the performance of six arbitrary partitions (see Table 9) on the two different systems of the actors for the GPU or CPU platform is introduced and analyzed alongside the results relative to the performance of the implementation employing the prior dataflow GPU/CPU data communication buffers developed by the authors. It can be observed that, in this test experiment, the speedup improvement of the overall application program execution is in the range 2.94- up to 15.01-times faster.



(**a**) System 1



(**b**) System 2

Figure 9. Speedup results for the RVC-CAL JPEG decoder on the two different hardware systems. 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.

Table 7. Statistics of the results depicted in Figure 9a using 10 executions.

	New FIFO				**Old FIFO**				**Speedup**
	Min	**Mean**	**Max**	**Var**	**Min**	**Mean**	**Max**	**Var**	
m1	2.07	2.07	2.07	1.33×10^{-4}	14.98	15.08	15.15	7.11×10^{-2}	7.31
m2	1.88	1.89	1.89	5.33×10^{-4}	10.47	10.58	10.65	8.14×10^{-2}	5.64
m3	5.03	5.10	5.23	1.14×10^{-1}	73.04	73.70	74.51	5.00	14.26
m4	5.57	5.59	5.62	5.63×10^{-3}	82.61	82.72	82.81	9.31×10^{-2}	14.73
m5	9.57	9.60	9.62	6.70×10^{-3}	143.39	143.96	144.43	2.50	15.01
m6	7.70	7.72	7.75	7.23×10^{-3}	62.96	63.36	63.64	1.14	8.21

Table 8. Statistics of the results depicted in Figure 9b using 10 executions.

	New FIFO				**Old FIFO**				**Speedup**
	Min	**Mean**	**Max**	**Var**	**Min**	**Mean**	**Max**	**Var**	
m1	1.72	1.72	1.72	4.30×10^{-5}	5.44	5.67	5.90	4.84×10^{-1}	3.29
m2	1.31	1.31	1.32	2.52×10^{-4}	3.81	3.86	3.94	4.42×10^{-2}	2.94

Table 8. *Cont.*

	New FIFO				Old FIFO				Speedup
	Min	Mean	Max	Var	Min	Mean	Max	Var	
m3	4.56	4.57	4.57	2.44×10^{-4}	24.93	24.95	24.99	8.83×10^{-3}	5.46
m4	5.12	5.13	5.14	7.46×10^{-4}	27.59	28.35	29.35	7.41	5.52
m5	8.44	8.45	8.46	5.37×10^{-4}	58.56	59.51	61.25	2.03×10^{1}	7.04
m6	6.93	6.94	6.95	6.13×10^{-4}	23.62	24.87	25.52	1.06×10^{1}	3.58

Table 9. The different mapping settings of the JPEG decoder implementation that are used in the results.

	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

6. Conclusions

In this work, two new design methodologies for the synthesis of dynamic dataflow programs to be executed on CUDA GPU platforms were introduced. Experimental results reporting the obtained performance improvements were also provided for the RVC-CAL dataflow application software.

The article improves and extends state-of-the-art methods in some essential parts of dataflow program synthesis for heterogeneous CPU/GPU platforms. One innovation is a new implementation of the buffer and related protocols interconnecting the GPU and CPU processing platforms. The data exchange in terms of throughput performance was improved versus state-of-the-art results. A second innovation is a new implementation of the GPU executable software, which can now execute without any control from the CPU executable, a feature that removes any processing or control need on the CPU side. Both design innovations were fully supported by computer-assisted compilations of the executable on both the GPU and CPU sides and demonstrated for the RVC-CAL dataflow language. The innovations were obtained by relying on an appropriate implementation of dynamic parallelism for the intra-kernel parallel execution of actions. Improvements of the overall performance were obtained by a dynamic grid configuration on the GPU platform. The semantic correctness of the dataflow model of execution and the correctness of the systematic compilation for both the CPU and GPU platform of executable code were verified by synthesizing real-sized application programs.

Further extensions of the work presented here include the investigation of an improved tool support providing the automatic selection of actors with a high potential for GPU parallelization. The current implementation requires an explicit selection to be provided by the designer. Another aspect that deserves further investigation is the development of good heuristics on how to dynamically choose at runtime the number of parallel threads that actions can execute to maximize resource utilization and overall application performance.

Author Contributions: Conceptualization, A.B.; methodology, A.B.; software, A.B.; validation, A.B., S.C.-B., and M.M.; investigation, A.B.; writing—original draft preparation, A.B.; writing—review and editing, A.B., S.C.-B., and M.M.; supervision, M.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Platzer, M.; Sargent, J.; Sutter, K. *Semiconductors: U.S. Industry, Global Competition, and Federal Policy (R46581)*; Technical Report; USA Congressional Research Service: New York, NY, USA, 2020.
2. Liu, J.; Hegde, N.; Kulkarni, M. Hybrid CPU-GPU Scheduling and Execution of Tree Traversals. *SIGPLAN Not.* **2016**, *51*, 2. [CrossRef]
3. Souravlas, S.; Sifaleras, A.; Katsavounis, S. Hybrid CPU-GPU Community Detection in Weighted Networks. *IEEE Access* **2020**, *8*, 57527–57551. [CrossRef]
4. Antoniadis, N.; Sifaleras, A. A hybrid CPU-GPU parallelization scheme of variable neighborhood search for inventory optimization problems. *Electron. Notes Discret. Math.* **2017**, *58*, 47–54. [CrossRef]
5. Michalska, M.; Casale-Brunet, S.; Bezati, E.; Mattavelli, M. High-precision performance estimation for the design space exploration of dynamic dataflow programs. *IEEE Trans. Multi-Scale Comput. Syst.* **2017**, *4*, 127–140. [CrossRef]
6. Savas, S. Hardware/Software Co-Design of Heterogeneous Manycore Architectures. Ph.D. Thesis, Halmstad University Press, Halmstad, Sweden, 2019.
7. Goens, A.; Khasanov, R.; Castrillon, J.; Hähnel, M.; Smejkal, T.; Härtig, H. Tetris: A multi-application run-time system for predictable execution of static mappings. In Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems, St. Goar, Germany, 12–13 June 2017; pp. 11–20.
8. Bloch, A.; Bezati, E.; Mattavelli, M. Programming Heterogeneous CPU-GPU Systems by High-Level Dataflow Synthesis. In Proceedings of the 2020 IEEE Workshop on Signal Processing Systems (SiPS), Coimbra, Portugal, 20–22 October 2020; pp. 1–6.
9. Bhattacharyya, S.; Deprettere, E.; Theelen, B. Dynamic Dataflow Graphs. In *Handbook of Signal Processing Systems*; Springer: Cham, Switzerland, 2013; pp. 905–944.
10. Johnston, W.; Hanna, J.; Millar, R. Advances in dataflow programming languages. *ACM Comput. Surv.* **2004**, *36*, 1–34. [CrossRef]
11. *23001-4:2011*; Information Technology—MPEG Systems Technologies—Part 4: Codec Configuration Representation. ISO: Geneva, Switzerland, 2011.
12. Yviquel, H.; Lorence, A.; Jerbi, K.; Cocherel, G.; Sanchez, A.; Raulet, M. Orcc: Multimedia Development Made Easy. In Proceedings of the 21st ACM International Conference on Multimedia, Barcelona, Spain, 21–25 October 2013; pp. 863–866.
13. Orcc Source Code Repository. Available online: http://github.com/orcc/orcc (accessed on 9 March 2022).
14. Cedersjö, G.; Janneck, J.W. Tÿcho: A framework for compiling stream programs. *ACM Trans. Embed. Comput. Syst.* **2019**, *18*, 1–25. [CrossRef]
15. Bezati, E.; Emami, M.; Janneck, J.; Larus, J. StreamBlocks: A compiler for heterogeneous dataflow computing (technical report). *arXiv* **2021**, arXiv:2107.09333.
16. Siyoum, F.; Geilen, M.; Eker, J.; von Platen, C.; Corporaal, H. Automated extraction of scenario sequences from disciplined dataflow networks. In Proceedings of the 2013 Eleventh ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2013), Portland, OR, USA, 18–20 October 2013; pp. 47–56.
17. Boutellier, J.; Nyländen, T. Design flow for GPU and multicore execution of dynamic dataflow programs. *J. Signal Process. Syst.* **2017**, *89*, 469–478. [CrossRef]
18. Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. Tensorflow: A system for large-scale machine learning. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), Savannah, GA, USA, 2–4 November 2016; pp. 265–283.
19. Sbîrlea, A.; Zou, Y.; Budimlíc, Z.; Cong, J.; Sarkar, V. Mapping a data-flow programming model onto heterogeneous platforms. *ACM SIGPLAN Not.* **2012**, *47*, 61–70. [CrossRef]
20. Gautier, T.; Lima, J.V.; Maillard, N.; Raffin, B. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, Cambridge, MA, USA, 20–24 May 2013; pp. 1299–1308.
21. Schor, L.; Tretter, A.; Scherer, T.; Thiele, L. Exploiting the parallelism of heterogeneous systems using dataflow graphs on top of OpenCL. In Proceedings of the 11th IEEE Symposium on Embedded Systems for Real-Time Multimedia, Montreal, QC, Canada, 3–4 October 2013; pp. 41–50.

22. Lin, S.; Liu, Y.; Plishker, W.; Bhattacharyya, S.S. 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, Sankt Goar, Germany, 23–25 May 2016; Association for Computing Machinery: New York, NY, USA, 2011; pp. 20–29. [CrossRef]
23. Lund, W.; Kanur, S.; Ersfolk, J.; Tsiopoulos, L.; Lilius, J.; Haldin, J.; Falk, U. Execution of dataflow process networks on OpenCL platforms. In Proceedings of the 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Turku, Finland, 4–6 March 2015; pp. 618–625.
24. Boutellier, J.; Nylanden, T. Programming graphics processing units in the RVC-CAL dataflow language. In Proceedings of the 2015 IEEE Workshop on Signal Processing Systems (SiPS), Hangzhou, China, 14–16 October 2015; pp. 1–6.
25. Schor, L.; Bacivarov, I.; Rai, D.; Yang, H.; Kang, S.H.; Thiele, L. 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, Tempere, Finland, 7–12 October 2012; pp. 71–80.
26. Lee, E.A.; Messerschmitt, D.G. Synchronous data flow. *Proc. IEEE* **1987**, *75*, 1235–1245. [CrossRef]
27. Rafique, O.; Krebs, F.; Schneider, K. Generating Efficient Parallel Code from the RVC-CAL Dataflow Language. In Proceedings of the 2019 22nd Euromicro Conference on Digital System Design (DSD), Kallithea, Greece, 28–30 August 2019; pp. 182–189.
28. SYCL. Available online: https://www.khronos.org/sycl/ (accessed on 9 March 2022).
29. CAL Exelixi Backends Source Code Repository. Available online: https://bitbucket.org/exelixi/exelixi-backends (accessed on 9 March 2022).
30. Bezati, E.; Casale-Brunet, S.; Mosqueron, R.; Mattavelli, M. An Heterogeneous Compiler of Dataflow Programs for Zynq Platforms. In Proceedings of the ICASSP 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Brighton, UK, 12–17 May 2019; pp. 1537–1541. [CrossRef]
31. Orcc-Apps Source Code Repository. Available online: https://github.com/orcc/orc-apps (accessed on 9 March 2022).