Exploiting the Parallelism of Heterogeneous Systems using Dataflow Graphs on Top of OpenCL

Lars Schor, Andreas Tretter, Tobias Scherer, and Lothar Thiele
Computer Engineering and Networks Laboratory
ETH Zurich, 8092 Zurich, Switzerland
firstname.lastname@tik.ee.ethz.ch

***Abstract*—Programming heterogeneous systems has been greatly simplified by OpenCL, which provides a common low-level API for a large variety of compute devices. However, many low-level details, including data transfer, task scheduling, or synchronization, must still be managed by the application designer. Often, it is desirable to program heterogeneous systems in a higher-level language, making the developing process faster and less error-prone. In this paper, we introduce a framework to efficiently execute applications specified as synchronous dataflow graphs (SDF) on heterogeneous systems by means of OpenCL. In our approach, actors are embedded into OpenCL kernels and data channels are automatically instantiated to improve memory access latencies and end-to-end performance. The multi-level parallelism resulting from the hierarchical structure of heterogeneous systems is exploited by applying two techniques. Pipeline and task parallelism are used to distribute the application to the different compute devices and data-parallelism is used to concurrently process independent actor firings or even output tokens in a SIMD fashion. We demonstrate that the proposed framework can be used by application designers to efficiently exploit the parallelism of heterogeneous systems without writing low-level architecture dependent code.**

***Keywords*—Automatic synthesis, dataflow languages, heterogenous systems, OpenCL, SDF graphs**

I. Introduction

The ever increasing computational requirements of real-time multimedia and scientific applications, coupled with the thermodynamic laws limiting the effectiveness of sequential architectures, has brought hardware designers to conceive systems with a high degree of parallelism. Such systems integrate a wide variety of compute devices including general-purpose processors, graphics processing units (GPUs), or accelerators. This enables software designers to perform their tasks on the best-suited device.

Programming heterogeneous computing systems, i.e., systems with different types of compute devices, can often be challenging, for instance because different devices require different code or support different (low-level) services. To provide a common interface for programming heterogeneous systems, the open computing language (OpenCL) [1] has been proposed. Programs following the OpenCL standard can run on any OpenCL-capable platform satisfying their resource requirements, and many hardware vendors, including Intel, AMD, NVIDIA, and STMicroelectronics, provide native support for OpenCL. Still, application designers have to manage many low-level details including data exchange, scheduling, or process synchronization, which makes programming heterogeneous systems difficult and error-prone. Thus, it would be desirable to have a higher-level programming language as significant areas of the design process could be automated, making the complete design process simpler, less error-prone, and more understandable.

Dataflow graphs have emerged as a promising paradigm for programming parallel systems. Essentially, an application specified as a dataflow graph is a set of autonomous actors that communicate through first-in first-out (FIFO) channels. When an actor fires, it reads tokens from its input channels and writes tokens to its output channels. As a large set of streaming applications, such as audio and video codecs, signal processing applications, or networking applications, can be naturally expressed as dataflow graphs, they enjoy great popularity in research and industry. Using them as model of computation allows the designer to explicitly specify the parallelism of an application and, at the same time, to separate computation from communication. Moreover, the use of a dataflow graph based design flow, such as the one introduced in this paper, allows designers to focus on the application-specific parts, with the architecture-specific implementation being automatically synthesized by the design flow.

In the past, efficiently programming dataflow graphs for heterogeneous systems was challenging due to hardware specific runtime and coding environments. We propose to meet this challenge by running applications on top of OpenCL. This has the advantage that the design flow can be used for any platform supporting OpenCL. Another challenge is to fully exploit the multi-level parallelism offered by hierarchically organized heterogeneous systems. A heterogeneous system might not only be composed of many compute devices; each of them might also be able to process multiple threads in parallel. Even more, there are devices that are only able to fully exploit their performance if the same instructions are concurrently applied to multiple data sources. To address this issue, we propose to use pipeline, task, and data parallelism to exploit the different levels of parallel hardware. Pipeline and task parallelism are leveraged by distributing the application to the different compute devices. Pipeline parallelism is achieved by assigning each actor of a chain to a different compute device and task parallelism is achieved by executing independent actors on different compute devices. Finally, data parallelism is used to exploit the parallelism offered by the individual compute devices: independent actor firings will be scheduled concurrently if the compute device is able to process multiple threads in parallel and multiple output tokens will be calculated in a SIMD fashion.

Following these ideas, the contribution of this paper is a design flow for executing applications specified as synchronous dataflow (SDF) [2] graphs on heterogeneous systems using OpenCL. SDF graphs are a restricted version of dataflow

programs in which actors read and write a fixed number of tokens per firing. First, we propose an architecture-independent application programming interface (API) for programming parallel applications as SDF graphs. Second, the corresponding runtime-system and program synthesis backend has been implemented for the distributed application layer (DAL) [3], which allows multiple (dynamically interacting) applications to efficiently execute on parallel systems. The communication interface abstracts low-level implementation details from the application designer in the sense that the memory location of the FIFO channels is optimized to improve memory access latencies and end-to-end performance. Seamless integration of I/O operations is provided by the ability to execute actors as native POSIX threads and to automatically transfer tokens between them and actors running on top of OpenCL. Finally, a detailed performance evaluation is carried out to support our claims. In particular, the overhead of the runtime-system is measured, the proposed concepts for exploiting the different levels of parallelism are compared with each other, and the end-to-end throughput is evaluated for various systems.

The remainder of the paper is structured as follows. In the next section, related work is reviewed. In Section III, a short overview on OpenCL is given. In Section IV, the proposed approach is summarized. In Section V, the proposed API is defined. In Sections VI and VII, details of our approach are discussed. Finally, experimental results are presented in Section VIII.

II. Related Work

While OpenCL has recently become very popular, mainly due to its non-proprietarity and the large number of supported platforms, people also struggled with managing many low-level details in OpenCL, and thus tried to find ways to simplify programming OpenCL-capable platforms. Maestro [4] is an extension of OpenCL that provides automatic data transfer between host and device as well as task decomposition across multiple devices. While tremendously simplifying the task of the programmer, Maestro introduces new restrictions as, for instance, that tasks have to be independent of each other. The task-level scheduling framework detailed in [5] extends OpenCL by a task queue enabling a task to be executed on any device in the system. Furthermore, dependencies between tasks are resolved by specifying a list of tasks that have to be completed before a new task is launched. Nonetheless, the burden of organizing data exchange is still left to the designer and no automatic design-flow is provided to efficiently design applications in a high-level programming language.

dOpenCL (distributed OpenCL) [6] is an extension of OpenCL to program distributed heterogeneous systems. The approach abstracts the nodes of a distributed system into a single node, but the programmer is still responsible for managing many low-level details of OpenCL.

The high-level compiler described in [7] generates OpenCL code for applications specified in Lime [8], a high-level Java compatible language to describe streaming applications. Lime includes a task-based data-flow programming model to express applications at task granularity, similar to dataflow graphs. The work in [7] particularly focuses on generating optimal OpenCL code out of the given Java code. In contrast to the high-level language based approach of Lime, we propose the use of a model-based design approach enabling actor-to-device mapping optimization and verification. Furthermore, our high-level specification enables the use of a lightweight runtime-system without the need of a virtual machine.

The model-based design of heterogeneous multi-processor systems using dataflow graphs is subject to several research projects. Even though not targeting OpenCL-capable platforms, the approach presented in [9] is closely related to our work. They introduce a formal dataflow language that can be used for both software and hardware synthesis. In contrast, the focus of our work is the efficient exploitation of the parallelism offered by applications specified as dataflow graphs.

Another approach to designing heterogeneous systems is to program the individual devices in separate languages. In this context, the execution of dataflow graphs on systems with CPUs and GPUs has been studied using NVIDIA's CUDA framework [10] for executing actors on the GPU. For instance, the multi-threaded framework proposed in [11] integrates both POSIX threads and CUDA into a single application. In contrast to our work, the approach is primarily concerned with overlapping computation and computation, but does not optimize the actual memory placement. KPN2GPU, a tool to produce fine-grain data parallel CUDA kernels from a dataflow graph specification, is described in [12]. An automatic code synthesis framework taking dataflow graphs as input and generating multi-threaded CUDA code is described in [13]. However, they assume that a separate definition of the actor is given for the CPU thread implementation and the GPU kernel implementation. In contrast, our API abstracts low-level details enabling the same definition to be used for CPU and GPU devices. Furthermore, Sponge [14] is a compiler to generate CUDA code from the StreamIt [15] programming model. Sponge only exploits the coarse-grained task parallelism of the dataflow graph while in our approach, the firing of an actors is additionally fragmented to fully exploit the multi-level parallelism of today's heterogenous systems. In addition, our approach generates OpenCL code enabling the same framework to be used for a wider range of heterogeneous platforms than the above described CUDA-targeting frameworks.

III. OpenCL Background

OpenCL defines a couple of new terms and concepts, which the reader may be unfamiliar with. For the sake of completeness, a short overview on these shall be given here; for a detailed documentation, however, we refer to [1].

Computational resources are organized hierarchically in OpenCL. There are *devices*, which consist of several *compute units* (CUs); those again are groups of one or more *processing elements* (PEs). As an example, a system with one CPU and one graphics card might provide two OpenCL devices: the CPU and the GPU of the graphics card. The different cores of the CPU would then each be one CU basically consisting of one PE. Each cluster of the GPU would be a CU with typically dozens of PEs. The devices are controlled by the *host*, which is a native program executed on the target machine[1].

The code that runs on the PEs is referred to as *kernels*. Essentially, a kernel is a function written in a C-like programming language called OpenCL C. It should perform a specific task with a well-defined amount of work and then return. All memory portions that it can use to communicate with the other kernels are provided as kernel arguments. When a kernel is

[1]On a PC, this means that the CPU may represent the host and a device at the same time; usually, this is implemented by having different OS threads.

executed on a PE, this execution instance is called a *work-item*. When the same kernel is instantiated multiple times at once (e.g., to achieve SIMD execution on GPUs), some of these work-items can be gathered to *work-groups*, which are allowed to share memory for intermediate calculations.

There are two major classes of memory types in OpenCL: global memory and different local memory types. Global memory can be accessed by the host as well as the devices, whereas local memory can only be used by the work-items as an intermediary storage. Note that there is no specification as to where the memory types are physically mapped. In case of a GPU, the global memory would typically reside in the graphics card DDR memory and the local memory in the GPU's fast scratch-pad memories, whereas on a CPU both types just represent the normal RAM of the PC. All memory types have in common that they are limited to the scope of a *context*, i.e., a user-defined set of devices. If, within a context, a global memory reference is passed from one device to another one with a different implementation of global memory, the OpenCL framework will automatically take care of the necessary copying. However, as this is done by the OpenCL backend (the *driver*), which is implemented by the hardware manufacturers, a context may not contain devices of different manufacturers (e.g., Intel CPU and NVIDIA graphics card).

The mapping and scheduling of the work-items is done in a semi-automatic way. The host creates *command queues* for each device and on these queues, it can place commands, namely the instantiation of a kernel or a data transfer to or from a global memory. The framework will decide on which PE a work-item is executed and when it will be scheduled. The host can influence this by choosing between *in-order* command queues, which execute the commands strictly in the order in which they were enqueued, and *out-of-order* queues, which may consider later commands if the first command is blocked.

While initially conceived for PCs and general purpose GPU programming, OpenCL is today supported by many more platforms. Examples include Intel's Xeon Phi accelerator, AMD's Accelerated Processing Unit (APU), and STMicroelectronics' STHORM platform [16].

IV. Problem and Approach

In this paper, a model-based design approach to programming heterogeneous systems in a systematic manner is considered. The proposed approach enables the system designer to efficiently execute SDF graphs on OpenCL-compatible heterogeneous systems. The goal is to maximize the end-to-end throughput of an application by leveraging the parallelism offered by heterogeneous systems. Other performance metrics as, for instance, response time, power consumption, or real-time guarantees are not regarded.

Clearly, the goal could be reached by compiling SDF graphs natively for a specific target platform without the additional layer introduced by OpenCL. Even though the advantages of a model-based design approach could be retained, such an approach would only support a small subset of heterogeneous systems and optimizing the application might be time-consuming. Using OpenCL and its runtime compiler enables the application to automatically take advantage of the latest enhancements of modern processors to maximize the throughput. On the other hand, in comparison with programming applications directly in OpenCL, the proposed model-based design approach offers a way to explicitly leverage various kinds of parallelism, enables functional verification, and allows to efficiently distribute the work between the different compute devices. Another advantage of the proposed model-based design approach is that the application-independent parts can be implemented once and reused in all applications.

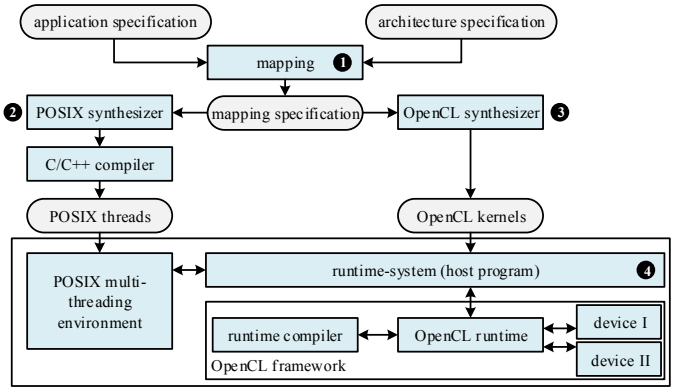


Fig. 1. Proposed high-level design flow to execute SDF graphs on heterogeneous systems using OpenCL.

A. Proposed Design Flow

The proposed design flow maps an SDF graph onto an OpenCL-compatible system in a number of steps, as illustrated in Fig. 1. The design flow's input is a dataflow graph specified in a high-level, architecture-independent API and an abstract specification of the target architecture. In Step 1, the actors are assigned to the available compute devices and potentially parallelized for running on multiple CUs and PEs. OpenCL C compliant actors are mapped onto OpenCL devices and the remaining actors (e.g., actors using file I/O or recursive functions) are executed as native POSIX threads (Step 2). Actors that have been assigned to OpenCL devices are synthesized in Step 3 to OpenCL kernels. Finally, in Step 4, OpenCL kernels are launched at runtime by the runtime-system that synchronizes the actors.

The focus of this paper is on the specification and efficient execution of SDF graphs on top of OpenCL-compatible heterogeneous systems. Thus, we do not detail Steps 1 and 2 on how to partition the SDF graph onto the available compute devices and how to synthesize actors as POSIX threads. An excellent survey on current trends in application partitioning and mapping is provided in [17]. More details on executing dataflow graphs on top of a POSIX-compliant multi-threading environment is given, e.g., in [3].

Next, we will give an overview on how parallelism is exploited before detailing the proposed high-level API, the software synthesizer, and the runtime-system.

B. Exploiting the Parallelism of Heterogeneous Systems

The key to efficient program execution on heterogeneous systems is to exploit the parallelism they offer. OpenCL supports three levels of hardware parallelism, namely different devices, CUs, and PEs. We take account of these by leveraging different kinds of application parallelism on each of them. More specifically, pipeline and task parallelism are used to distribute the dataflow graph to the different devices. Afterwards, data parallelism is used to exploit the parallelism offered by the individual devices. Independent actor firings are concurrently executed on different CUs of the same device and, to achieve SIMD execution on different PEs, independent output tokens

are calculated in parallel. To illustrate how data parallelism is leveraged, we take the actor shown in Figure 2 as an example. Per firing, it reads six tokens and writes three tokens. Output tokens are independent of each other, although depending on the same input tokens. In our framework, multiple firings might be executed in parallel on different CUs and each output token is calculated on a different PE.

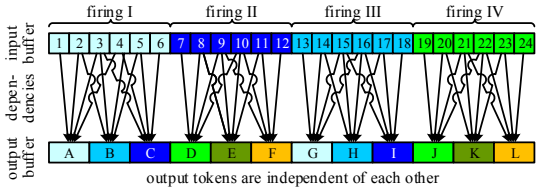


Fig. 2. Exemplified actor behavior where the calculation of each output token is independent of the other tokens even though three output tokens depend on the same input data.

This concept can be implemented in OpenCL by mapping actors, firings, and output tokens onto equivalent OpenCL constructs. The basic idea is that a work-item calculates one or more output tokens of a firing so that all work-items gathered to a single work-group calculate together all output tokens of a firing. All work-items belonging to the same work-group have access to the same input tokens but write their output tokens to different positions in the output stream. This allows for SIMD execution.

In addition, if the topology of the dataflow graph allows it, multiple firings will be calculated in parallel by having multiple work-groups. Thus, work-items from different work-groups access different input tokens and write their output tokens to different memory positions. While this could also be achieved by adding a splitter distributing the work and a merger collecting the data, the communication and management overhead in our approach is less since multiple invocations can be combined into one invocation. This particularly has an impact if certain input or output channels are mapped onto memory other than the device's memory. The memory required by all firings belonging to the same kernel invocation is then copied as one block. Note that the number of work-groups per kernel invocation might be limited by the topology of the graph, e.g., if the actor is part of a cycle.

When launching a kernel, the OpenCL runtime assigns each work-group to a CU and each work-item to a PE of its work-group's CU. Note that in OpenCL, the number of work-groups and work-items is not bounded by the number of CUs and PEs. OpenCL can handle more work-groups than available CUs and more work-items than available PEs. In fact, OpenCL might use them to improve the utilization by switching the context if work-items of a particular work-group are stalled due to memory contention. Figure 3 illustrates the different levels of hardware and software parallelism and how they are linked.

In order to fully leverage the parallelism offered by heterogeneous systems, design decisions have to be taken in all steps of the proposed design flow. First of all, the specification of the SDF graph should include details about the output tokens. In Step 1, besides binding actors to compute devices, decisions must be taken about the number of work-groups and work-items per work-group. The number of work-groups highly depends on the characteristics of the device, e.g., a loss of performance must be expected if the number of work-groups

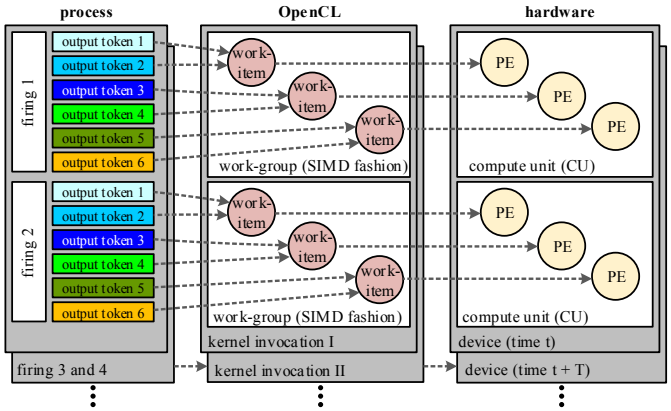


Fig. 3. Illustration of the different levels of hardware and software parallelism when executing an actor on top of an OpenCL device.

is not a multiple of the number of CUs. Similarly, the number of work-items might depend on the number of available PEs. In Step 3, the OpenCL kernel is synthesized based on these numbers and the output tokens are distributed to the work-items. At start-up, the runtime-system forwards the parallelization directives to the OpenCL framework, which builds and optimizes the kernel to leverage the latest enhancements of the device. The runtime-system is further in charge of reducing the communication overhead, e.g., by selecting a good memory location for the FIFO channels and combining data transfers.

V. High-Level System Specification

In this section, we describe the high-level specification we propose for automatic program synthesis. It is illustrated in Fig. 4, based on an example.

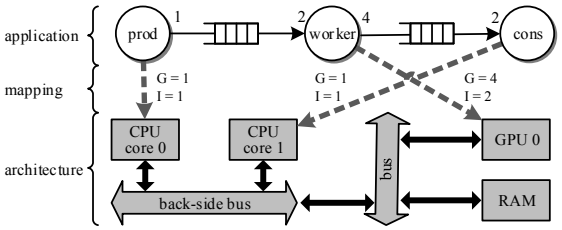


Fig. 4. Example system illustrating the proposed high-level specification. G refers to work-groups and I refers to work-items per work-group.

Application specification. In this work, we consider applications that can be represented as an SDF graph. An SDF graph is basically a set of autonomous actors that communicate through FIFO channels by reading/writing tokens from/to a channel. Reading a value from a FIFO channel is always destructive, i.e., the value is removed from the FIFO channel. Each actor of a graph adhering to the SDF model of computation produces and consumes a fixed number of data tokens per firing, which is called the token rate. In Fig. 4, the token rate is specified by a number next to the ports. In addition, we introduce the notion of *blocks*. A block is defined as a group of consecutive output tokens that are jointly calculated and do not depend on any other output tokens. In the example illustrated in Fig. 2, a single output token forms a block as all output tokens are independent of each other. Clearly, all blocks of an output port have to be of the same size and that size must be a fraction of the port's token rate. This last kind of fragmentation can typically not be leveraged by an ordinary dataflow graph

specification; however, we claim that the proposed extension is natural, as in practice, actors are often specified such that one firing calculates multiple output tokens that require the same input data even though they could be calculated independently of each other. The discrete Fourier and cosine transforms, block-based video processing algorithms [18], and sliding window methods often used for object detection [19] are just a few examples of algorithms that exhibit this property.

SDF graphs have the advantage that the application behavior is specified independently of the topology. We employ the XML format shown in Listing 1 to specify the topology. Each actor has a set of ports representing the connections to the FIFO channels. Besides its type, a port has a field `rate` specifying the token rate, i.e., the number of tokens produced or consumed per firing, and each output port has a field `blocksize` specifying the size of a corresponding block in number of tokens.

Listing 1. Specification of the SDF graph illustrated in Fig. 4.

```
<graph>
  <actor name="prod">
    <port type="out" name="out1" rate="1" blocksize="1"/>
    <src type="c" location="prod.c"/>
  </actor>
  <actor name="worker">
    <port type="in" name="in1" rate="1"/>
    <port type="out" name="out1" rate="4" blocksize="2"/>
    <src type="c" location="worker.c"/>
  </actor>
  <actor name="cons">
    <port type="in" name="in1" rate="2"/>
    <src type="c" location="cons.c"/>
  </actor>

  <channel capacity="8" tokensize="4" name="channel1">
    <sender actor="prod" port="out1"/>
    <receiver actor="worker" port="in1"/>
  </channel>
  <channel capacity="32" tokensize="1" name="channel2">
    <sender actor="worker" port="out1"/>
    <receiver actor="cons" port="in1"/>
  </channel>
</graph>
```

The behavior of an actor is specified in C/C++ and follows the ideas illustrated in Listing 2; however, the application designer is free to optimize the code using OpenCL C constructs. Providing the application designer with the ability to write the code in C/C++ has various advantages over OpenCL C, e.g., it hides low-level details of OpenCL and provides the opportunity to not only execute the actor as OpenCL kernel, but also as POSIX thread. The FIRE procedure represents one firing of the actor and is repeatedly executed. Additionally, there is an INIT procedure, which is called once at start-up of the application in order to produce potential initial tokens. We leverage the concept of windowed FIFOs [20] for the FIFO channel interface. The basic idea of windowed FIFOs is that actors directly access the FIFO buffer avoiding (expensive) memory copy operations. A window for reading is acquired by using CAPTURE, which returns a pointer to the requested memory. Similarly, a buffer reference for writing is obtained by using RESERVE, potentially together with a block identifier (`blk`, the default is to use the first block). Finally, the FOREACH directive allows for a parallel calculation of the individual blocks of an output port. Note that, although not shown here, it is possible to include multiple ports in one FOREACH statement if they take the same number of blocks per firing.

Listing 2. Implementation of the actor "worker" using the proposed API.

```
void INIT() {
  initialization();
  FOREACH(blk IN PORT_out1) {
    TOKEN_OUT1_t *wbuf = RESERVE(PORT_out1, blk);
    createinittokens(wbuf, blk); // write initial tokens to wbuf
  }
}

void FIRE() {
  preparation();
  TOKEN_IN1_t *rbuf = CAPTURE(PORT_in1);
  FOREACH(blk IN PORT_out1) {
    TOKEN_OUT1_t *wbuf = RESERVE(PORT_out1, blk);
    manipulate(rbuf, wbuf, blk); // read from rbuf, write to wbuf
  }
}
```

Architecture specification. The described approach targets heterogeneous platforms that are OpenCL-capable and can be managed by a single host. A system may consist of hardware components of different vendors with individual OpenCL drivers. The extension to distributed systems with multiple host controllers is straightforward, e.g., by connecting the different nodes by an MPI interface as presented in [3]. For the high-level specification of the architecture, we employ an XML format specifying the available devices and their OpenCL identifiers.

Mapping specification. Finally, a mapping of the actors onto the devices is required for program synthesis. The mapping is generated in Step 1 of the design flow illustrated in Fig. 1.

The mapping particularly decides on the distribution of the actors to the architecture by assigning each actor to a compute device, see Listing 3. In addition, the mapping defines the number of work-items per work-group that should be instantiated for an actor and the number of work-groups used to gather the work-items. The mapping may also specify a work distribution pattern for every output port. This setting indicates how the output blocks are assigned to the work-items, e.g., consecutive blocks to the same work-item or to different work-items for simultaneous access.

Listing 3. XML specification of the mapping based on the example of the "worker" actor shown in Fig. 4.

```
<binding>
  <actor name="worker">
    <port name="out1" work-dist="strided"/>
  </actor>
  <device name="gpu_0"/>
  <workgroups count="4"/>
  <workitems count="2"/>
</binding>
```

VI. OpenCL Synthesizer

Having identified the actors that are mapped onto OpenCL devices in Step 1, Step 3 is to synthesize these actors into OpenCL kernels by performing a source-to-source code transformation. In the following, we illustrate this step based on the "worker" actor shown in Listing 2.

As described in Section III, an OpenCL kernel specifies the code that runs on one PE. Thus, the basic idea of the OpenCL synthesizer is to replace the high-level communication procedures with OpenCL-specific code so that the FIRE procedure calculates a certain number of output blocks. The number of output blocks depends on the number of work-items per work-group as specified by the mapping illustrated in Listing 3. When launching the kernel, the runtime-system

Listing 4. Embedding the actor shown in Listing 2 into an OpenCL C kernel. Newly added lines are marked by 02 and modified lines by 07.

```
01  // declare helper variables according to dataflow graph specification in Listing 1
02  const int TOKEN_IN1_RATE = 1;
03  const int TOKEN_OUT1_RATE = 4, BLOCK_OUT1_SIZE = 2;
04  const int BLOCK_OUT1_COUNT =
05                      TOKEN_OUT1_RATE / BLOCK_OUT1_SIZE;

07  __kernel void INIT(__global TOKEN_OUT1_t *out1) {
08    int gid = get_group_id(0); // work-group id
09    int lid = get_local_id(0); // work-item id
10    int lsz = get_local_size(0); // work-item count
11    initialization();
12    for (int blk=lid; blk<BLOCK_OUT1_COUNT; blk+=lsz) {
13      __global TOKEN_OUT1_t *wbuf1 = out1 +
14          gid*TOKEN_OUT1_RATE + blk*BLOCK_OUT1_SIZE;
15      createinittokens(wbuf, blk); // write initial tokens to wbuf
16    }
17  }

19  __kernel void FIRE(__global TOKEN_IN1_t *in1,
20                     __global TOKEN_OUT1_t *out1) {
21    int gid = get_group_id(0); // work-group id
22    int lid = get_local_id(0); // work-item id
23    int lsz = get_local_size(0);   // work-item count
24    preparation();
25    __global TOKEN_IN1_t *rbuf = in1 + gid*TOKEN_IN1_RATE;
26    for (int blk=lid; blk<BLOCK_OUT1_COUNT; blk+=lsz) {
27      __global TOKEN_OUT1_t *wbuf1 = out1 +
28          gid*TOKEN_OUT1_RATE + blk*BLOCK_OUT1_SIZE;
29      manipulate(rbuf, wbuf, blk); // read from rbuf, write to wbuf
30    }
31  }
```

will specify the number of work-groups and work-items so that one or more firings are concurrently calculated by one kernel invocation. Listing 4 illustrates how an actor is synthesized into an OpenCL kernel:

- The INIT and FIRE procedures are declared as kernels with one parameter per output channel being added to INIT and one parameter per input and per output channel being added to FIRE (Lines 07 and 19 f.).
- Constant helper variables are declared (Lines 02 to 05).
- CAPTURE is replaced with a pointer to the head of the corresponding FIFO channel, using the work-group id to select the correct region (Line 25).
- FOREACH is implemented as a loop iterating over the block identifiers (Lines 12 and 26). For each work-item, its ID determines the iteration that it computes. In Listing 4, the solution for a strided work distribution is shown. Note that the loop might not be executed by all work-items, in particular if the number of work-items is larger than the number of blocks. The number of work-items might be larger than the number of blocks if, for instance, an actor has multiple output ports with each having a different number of blocks.
- RESERVE is replaced with a pointer to the block being written in the current iteration (Lines 13 f. and 27 f.).

VII. Runtime-System for SDF Graphs

The runtime-system's task is to dispatch the OpenCL kernels to the connected devices by providing an implementation of the high-level API proposed in Section V. This includes two basic functionalities, namely a framework to synchronize the actors and a memory-aware implementation of the FIFO channels. Their implementation is the key to an efficient execution of SDF graphs on top of OpenCL. Thus, the runtime-system must try to maximize the utilization of OpenCL devices and to reduce communication latencies. To this end, we leverage two properties of OpenCL:

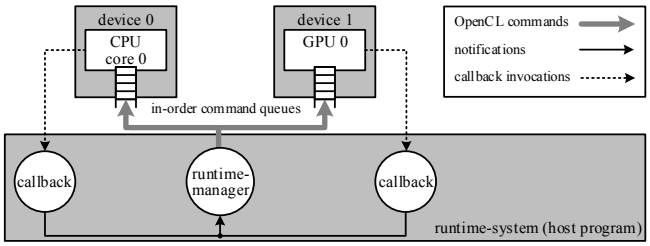


Fig. 5. Structure of the actor synchronization and invocation framework.

In-order command queues. The utilization of a device can be maximized by always having commands in the command queue. In-order queues allow for an efficient specification of dependencies as they guarantee a fixed execution order of work-items. Unlike all other methods of specifying execution orders, they can be autonomously interpreted by the devices without time-consuming interventions of the host.

Hierarchical memory system. Work-items have to use global memory for data exchange. The physical location of these, however, depends on the devices and on the OpenCL directives used by the host. It is therefore important to keep track of these locations and to take influence such that the best-suited memory is used whenever possible. The objective is to keep the number of memory copy transactions as low as possible, reducing latencies and maximizing the end-to-end throughput.

Based on these considerations, we describe the two basic functionalities of the runtime-system in the following.

A. Actor Synchronization

As each kernel invocation executes a specific amount of work and then returns, a mechanism is required to repeatedly reinvoke the kernel with different input data. However, a kernel can only be invoked if enough tokens and space are available on each input and output channel, respectively. Thus, the basic idea of the actor synchronization and invocation framework is to monitor the FIFO channels and, depending on their fill level, to (re)invoke the kernels.

Figure 5 illustrates the structure of the actor synchronization and invocation framework. On start-up, the host creates an in-order command queue for each device and starts the runtime-manager. The runtime-manager then monitors the fill level of the FIFO channels and invokes OpenCL kernels. Callbacks triggered by certain command execution states are used to keep this information up to date. For instance, the completion of a kernel may trigger a callback function notifying the runtime-manager that a certain amount of tokens has been produced or consumed (see below).

The aim of the runtime-manager is to maximize the utilization of the individual devices by avoiding empty command queues. It might even have multiple kernel invocations of the same actor simultaneously enqueued, in particular as long as data are available in the input channels and buffer space is available in the output channels. Using in-order command queues is advantageous as the runtime-manager can enqueue commands already before completion of other commands they depend on. For instance, if two actors having a common FIFO channel are mapped onto the same device, the runtime-manager can update the number of available tokens of the FIFO channel immediately after enqueuing the source actor, even though no tokens have been produced yet.

B. FIFO Communication

The communication service has two tasks to accomplish, namely data transfer and, indirectly, actor invocation. While data transfer should happen ideally without even involving the host, the host must know the state of the channels, which determines whenever an actor can be fired. Therefore, a distributed FIFO channel implementation is considered where the FIFO channel's fill level is managed by the host and only updated by the runtime-manager. The memory buffer, on the other hand, may be allocated in device memory.

In the following, we discuss FIFO communication between actors mapped onto a single device, between actors mapped onto different devices, and between an actor running on top of OpenCL and an actor executing as a POSIX thread.

FIFO communication on single device. If both the source and the sink actor are mapped onto the same device, a buffer is allocated in the global memory of the corresponding device. In each firing, the source actor gets a pointer to the current tail and the sink actor to the current head of the virtual ring buffer. The ring buffer is implemented as a linear buffer in the device memory and OpenCL's sub-buffer functionality is used to split the buffer into individual tokens. As an in-order command queue is used, the runtime-manager can also update the tail and head pointers upon enqueuing the kernels. The FIFO channel implementation and the communication protocol are illustrated in Fig. 6.

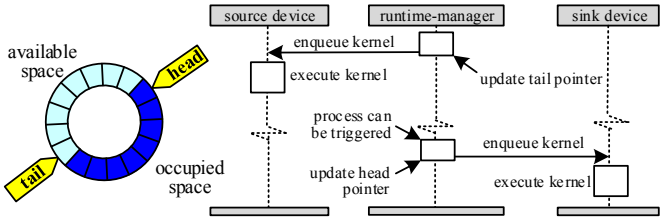


Fig. 6. FIFO channel implementation on single devices (left) and corresponding communication protocol (right).

FIFO communication between devices. While the FIFO communication implementation on a single device requires no memory copies, it is typically required to transfer the data from one device to the other one if two actors mapped onto different devices communicate with each other. In the following, we consider the general case where host accessible memory (HAM) has to be used for this data transfer. The overall idea is to allocate the entire buffer in both global memories of the involved devices and in the HAM. The runtime-manager keeps track of all three memories by having three tail and three head pointers. Preallocating the memory is particularly advantageous as expensive memory allocation operations are avoided at runtime.

The communication protocol is illustrated in Fig. 7. The basic principle underlying the protocol is that data are forwarded as soon as possible from the source to the sink in packets of reasonable size. The protocol works as follows. After enqueuing the command for launching the source kernel, the runtime-manager also enqueues a command for updating the host memory with the newly written data. As in-order command-queues are installed between host and devices, it is ensured that the memory update command is only executed when the kernel execution is completed. Once the memory update command is completed, a callback tells the runtime-manager to update the fill level of the FIFO channel. The runtime-manager then checks if the channel contains enough data for a kernel invocation of the sink actor. If this is the case, it enqueues a command to update the device memory with the newly available data. Once the sink fulfills all execution conditions, i.e., there are enough tokens and space available on each input and output channel, the runtime-manager enqueues the kernel into the command queue, initiating a new firing of the actor. As soon as the kernel execution is completed, a callback tells the runtime-manager that tokens have been consumed.

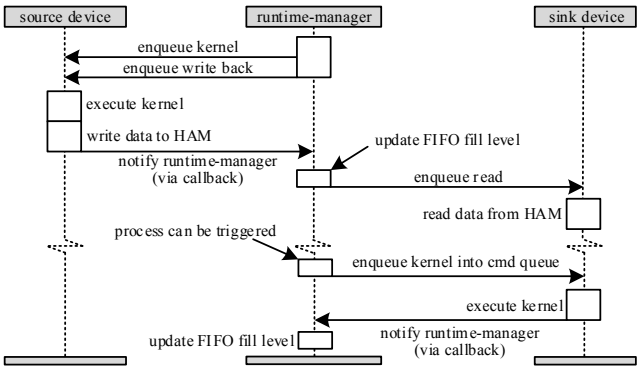


Fig. 7. Communication protocol for data exchange between two devices.

A special case of this situation is if one of the devices is the CPU. In that case, the global memory of the device is identical with the HAM and thus only two buffers are needed.

FIFO communication between device and POSIX thread. The situation that one of the actors is executed as a POSIX thread is similar to the situation where one of the OpenCL kernels is executed on the CPU device. The actor directly accesses the buffer in the HAM.

VIII. Experimental Results

In this section, we evaluate the performance of the proposed framework. The goal is to answer the following questions. *a*) What is the overhead introduced by the proposed runtime-system and can we give guidelines when to execute an actor as OpenCL kernel or as a POSIX thread? *b*) How expensive is communication between actors, both when mapped onto the same device and onto different devices? *c*) Does the proposed framework offer enough flexibility for the programmer to efficiently exploit the parallelism offered by GPUs? — To answer these questions, in the following, we evaluate the performance of synthetic and real-world applications on two different heterogeneous platforms.

A. Experimental Setup

The considered target platforms are outlined in Table I. Both CPUs have hyper-threading deactivated and support the advanced vector extensions (AVX). Applications benefit from AVX in the sense that several work-items might be executed in a lock-step via SIMD instructions. The Intel SDK for OpenCL Applications 2013 is used for Intel hardware. OpenCL support for the graphics cards is enabled by the NVIDIA driver 319.17 and by the AMD Catalyst driver 13.1. If not specified otherwise, the used compiler is G++ 4.7.3 with optimization level O2 and the default optimizations are enabled in OpenCL.

B. Overhead of the Runtime-system

First, we quantify the overhead introduced by OpenCL and by the proposed actor synchronization and invocation

TABLE I. PLATFORMS USED TO EVALUATE THE PROPOSED FRAMEWORK.

ID	CPU	*cores*	GPU(s)	*clusters*	*PEs*	*memory bandwidth*	operating system
A	Intel Core i7-2600K at 3.4 GHz	4	NVIDIA GeForce GTX 670	7	1344	192.2 GB/s	Arch Linux with kernel 3.7.6-1
			AMD Radeon HD 7750	8	512	72.0 GB/s	
B	Intel Core i7-2720QM at 2.2 GHz	4	NVIDIA Quadro 2000M	4	192	28.8 GB/s	Ubuntu Linux 12.04 with kernel 3.5.0-28

mechanism. For this purpose, we have designed a synthetic SDF application called PRODUCER-CONSUMER that consists of two actors connected by a FIFO channel. Both actors access the FIFO channel at the same token rate. While the only task of the sink actor is to read the received tokens, the source actor first performs some calculations before writing the result to the output channel. This result consists of multiple output tokens that can be calculated in parallel using multiple work-items. The number of calculations is varied to change the execution time per firing of the source actor. This also changes the invocation period and its inverse, i.e., the invocation frequency as the SDF graph's invocation interval is only limited by the execution time of the source actor. For brevity, we only report the results for platform A; however, the results for platform B exhibit similar trends.

Overhead of the OpenCL framework. To quantify the overhead of the OpenCL framework, we synthesized both actors of the PRODUCER-CONSUMER application either for OpenCL or POSIX. When synthesizing the application for POSIX, either no optimization, optimization level O2, or optimization level O3 with setting `march=native` is used. When `march=native` is set, G++ automatically optimizes the code for the local architecture. When synthesizing the application for OpenCL, the number of work-groups is set to one so that each actor is executed on exactly one core.

Figure 8 shows the speed-up of the OpenCL implementation and the optimized POSIX implementations versus the execution time of the unoptimized POSIX implementation, with the number of calculations in the source actor being varied. The x-axis represents the invocation period of the unoptimized POSIX implementation. As the kernel invocation overhead in OpenCL is virtually independent of the kernel's amount of work, the POSIX implementation performs better for small invocation periods. On the other hand, the overhead is less crucial for longer invocation periods and OpenCL implementations achieve even higher speed-ups than the optimized POSIX implementations. This may be due to OpenCL's ability to utilize the CPU's vector extension so that four work-items are executed in SIMD fashion. That assumption is supported by the fact that the CPU we use is only able to execute four work-items in parallel and distributing the work to five work-items is counter-productive. Note that G++ also makes use of the AVX commands when the corresponding option is enabled, which is why the speed-up of the O3 implementation is always higher than the speed-up of the OpenCL implementation with one work-item.

Overhead of the runtime-system. To evaluate the overhead caused by the proposed runtime-system, we synthesize the PRODUCER-CONSUMER application for OpenCL with one work-item per actor and measure its execution time for two different actor synchronization mechanisms. The first mechanism is called "manager" and corresponds to the proposed actor synchronization mechanism presented in Section VII, i.e., a runtime-manager monitors the FIFO channels and creates new kernel instances. A larger channel capacity allows the runtime-manager to have multiple firings of the same actor simultaneously enqueued in the command queue, which can lead to a higher utilization of the device. The second mechanism is called "callback" and is a naive approach that uses the callback functions of OpenCL (e.g., kernel execution finished, data transfer completed) for enqueueing further commands.

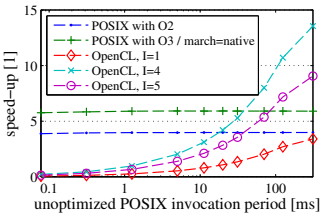


Fig. 8. Speed-up of the OpenCL and the optimized POSIX implementations versus the unoptimized POSIX implementation. I refers to the number of work-items.

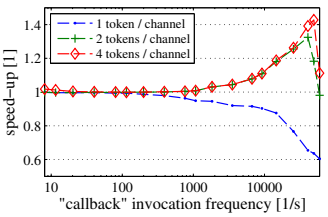


Fig. 9. Speed-up of the PRODUCER-CONSUMER application when using the "manager" mechanism relative to the execution time when using the "callback" mechanism.

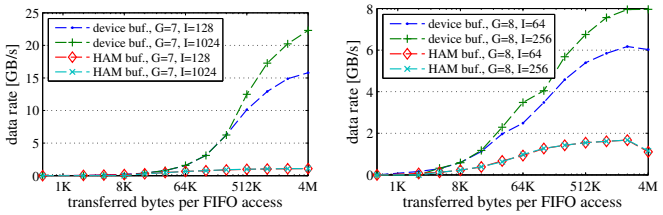
Figure 9 shows the speed-up of the PRODUCER-CONSUMER application when using the "manager" mechanism relative to the execution time when using the "callback" mechanism. Both actors are mapped onto the CPU and the number of calculations in the source actor is varied. The x-axis represents the invocation frequency when using the "callback" mechanism. As expected, no speed-up is achieved for low invocation frequencies. For higher invocation frequencies, the "manager" mechanism is slower than the "callback" mechanism if the FIFO channel has a capacity of one token. In this case, both mechanisms can enqueue a new firing only if the previous firing is completed. However, the feedback loop is larger for the "manager" mechanism as it also includes the runtime-manager. If the FIFO channel has a capacity of more than one token, the "manager" mechanism can enqueue a new firing in parallel to the execution of the old one so that the "manager" mechanism is faster than the "callback" mechanism. Finally, for very large invocation frequencies, the firing completes earlier than the "manager" mechanism can enqueue a new firing so that the speed-up declines again.

Overall, the results demonstrate that the proposed runtime-manager performs considerably better than a naive actor invocation mechanism. Furthermore, we claim that OpenCL is not only useful for executing actors on GPUs, but also on CPUs if data parallelism can be efficiently leveraged.

C. Intra- and Inter-Device Communication

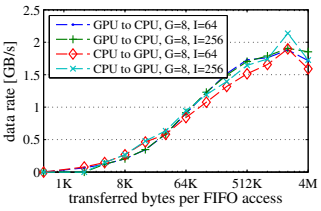
Next, we evaluate the communication costs for different types of FIFO channel implementations. The goal is to show that mapping the memory buffers in a sub-optimal manner onto the distributed memory architecture may affect the overall performance of the application.

For this purpose, we measure the data transfer rate between two actors mapped onto either the same device or different devices. Our test application is designed such that in each firing, the producer actor generates one token, which it fills with a simplistic integer sequence. It then sends the token to

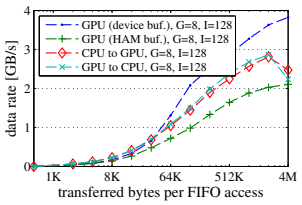


(a) Platform A, NVIDIA GPU.

(b) Platform A, AMD GPU.



(c) Platform A, AMD GPU and CPU.



(d) Platform B.

Fig. 10. Data rate for different actor mappings and channel implementations.

the consumer actor that reads the values. This set-up is selected to ensure that the workload per transmitted byte is independent of the size of a token. Figure 10 shows the aggregated data rate for different actor mappings and channel implementations, where the size of a single token is varied between $512\,\text{bytes}$ and $4\,\text{MB}$. The channel size is fixed to $32\,\text{MB}$.

In the first set-up, the data transfer rate between two actors mapped onto the same device is measured for two different FIFO implementations. The results are shown in Figs. 10a and 10b for the NVIDIA and the AMD GPU, respectively. First, we use the optimized FIFO implementation, which keeps the data in the global memory of the device ("global buf."). Second, we used a naive FIFO implementation, which transfers data through HAM ("HAM buf."). The number of work-groups (G) per actor is set to the number of CU and the number of work-items per work-group is indicated by I. Having more work-items might lead to higher data transfer rates as more PEs can concurrently read and write. The observed peak data rate is $20.30\,\text{GBytes/s}$ when both actors are mapped onto the NVIDIA GPU and $7.96\,\text{GBytes/s}$ when both actors are mapped onto the AMD GPU. The data transfer rate is considerably lower if the memory buffer is allocated in the HAM. In this case, the observed peak data rate is $1.09\,\text{GBytes/s}$ (both actors are mapped onto the NVIDIA GPU) and $1.67\,\text{GBytes/s}$ (both actors are mapped onto the AMD GPU).

The data transfer rate for the case that one actor is mapped onto the CPU and the other actor is mapped onto the AMD GPU is illustrated in Fig. 10c. "GPU to CPU" means that the producer actor is mapped onto the GPU and the consumer actor onto the CPU. For "CPU to GPU", it is vice versa. The observed peak data rate is $1.91\,\text{GBytes/s}$ when the producer actor is mapped onto the GPU and $2.14\,\text{GBytes/s}$ when the producer actor is mapped onto the CPU.

Finally, Fig. 10d shows a summary of the data transfer rates for platform B. The observed peak data rate is $3.82\,\text{GBytes/s}$ and measured when both actors are mapped onto the GPU.

We conclude that exploiting on-device communication without going through HAM is essential for the performance. Yet, we think that the communication costs still leave much room for improvement. In future work, we would like to include the local memory of a CU and to overlap local to global memory communication with computation.

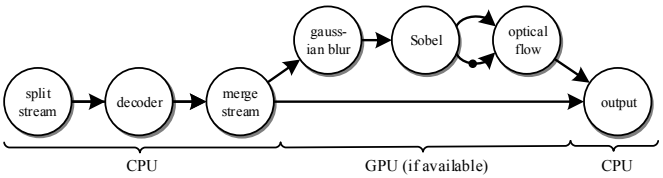


Fig. 11. SDF graph of the video-processing application.

D. Exploiting Data and Task Parallelism

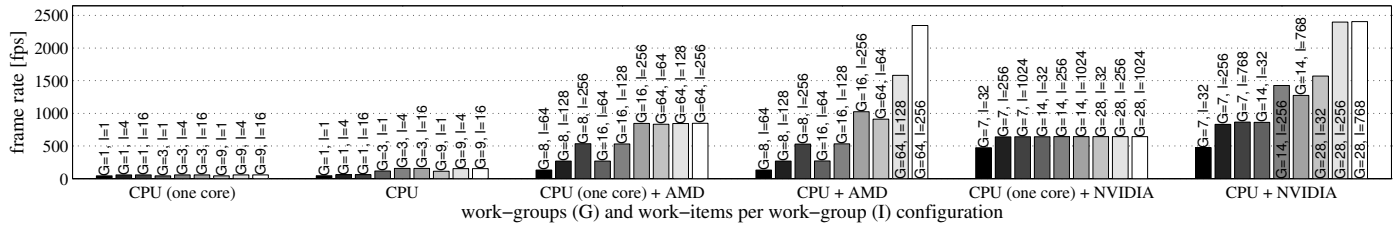
The results presented so far indicate that the proposed concepts of multi-level parallelism indeed lead to higher performance. Next, we will investigate this question further by comparing the performance of a real-world application for different mappings and degrees of parallelism.

For this purpose, a video-processing application has been implemented that decodes a motion-JPEG video stream and then applies a motion detection method to the decoded video stream, see Fig. 11 for the SDF graph. The MJPEG decoder can decode multiple video frames in parallel but cannot divide the output tokens into smaller pieces. The motion detection method is composed of a Gaussian blur, a gradient magnitude calculation using Sobel filters, and an optical flow motion analysis. Tokens transmitted between these three components correspond to single video frames, but in all filters, the calculation of an output pixel is independent of the other output pixels. A gray scale video of 320×240 pixels is decoded and analyzed in all evaluations. If a GPU is available, the Gaussian blur, the Sobel, and the optical flow actors will be mapped onto it. Otherwise, all actors will be mapped onto the CPU.

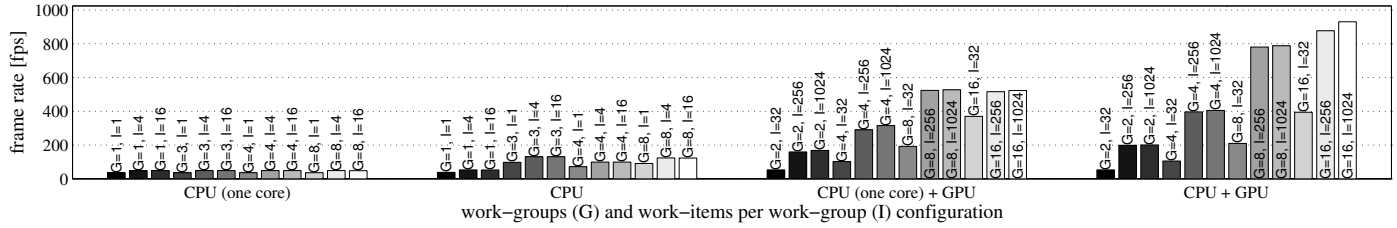
In what follows, we measure the frame rate of the application for different degrees of parallelism and configurations of the target platforms. While fixing the number of frames that are concurrently decoded to three, the number of work-groups (G) and the number of work-items (I) per work-group are varied for the Gaussian blur, the Sobel, and the optical flow actors. However, for the sake of simplicity, all three actors have the same number of work-groups and work-items.

Figure 12a shows the frame rates achieved with different configurations on target platform A. The highest performance ($2347\,\text{fps}$) is achieved with all CPU cores and a GPU device available. In that case, the bottleneck is not anymore the GPU, but the CPU that is not able to decode more frames. Similarly, the CPU is mostly the bottleneck if only one core is used to decode the frames. Mapping all actors onto a single core of the CPU leads to a maximum frame rate of $57\,\text{fps}$, thus a speed-up of almost $41x$ can be achieved when all cores are used together with a GPU device. The plot also shows that the GPU can highly leverage a large number of work-items. Finally, note that a different work distribution pattern is used for the calculation of the individual output pixels depending on whether the Gaussian blur, the Sobel, and the optical flow actors are mapped onto the CPU or the GPU. Mapping consecutive blocks to the same work-item works best for the CPU while consecutive blocks to different work-items works best for the GPU.

Figure 12b shows the frame rates for different configurations on target platform B. The peak performance ($931\,\text{fps}$) is achieved when all cores of the CPU and the GPU device are available. It constitutes a speed-up of $19x$ compared to the case where all actors are mapped onto one CPU core. The plot also shows that the number of work-groups should be aligned with the available hardware. We have found that the Intel OpenCL



(a) Evaluation platform A.



(b) Evaluation platform B.

Fig. 12. Frame rate of the SDF graph outlined in Fig. 11 for different degrees of parallelism and different configurations of the target platforms.

SDK version we used distributes the OpenCL kernels to only three cores, which is why a higher frame rate is obtained when executing three work-groups instead of four.

Overall, the results demonstrate that the proposed framework provides developers with the opportunity to exploit the parallelism provided by state-of-the-art GPU and CPU systems. In particular, speed-ups of up to $41x$ could be measured when outsourcing computation intensive code to the GPU.

IX. Conclusion

In this paper, we have presented a high-level programming framework to execute applications specified as SDF graphs on heterogeneous systems using OpenCL. The proposed high-level API abstracts low-level implementation details from the application designer so that the designer can focus on the application-specific parts. In this framework, actors are automatically embedded into OpenCL kernels and scheduled by a centralized runtime-manager. FIFO channels are instantiated by the runtime-system in a way that memory access latencies and end-to-end performance are improved. The multi-level parallelism typically offered by heterogeneous systems is individually exploited on each level. First, task and pipeline parallelism are used to distribute the application to the different compute devices. Afterwards, the parallelism offered by the individual compute devices is exploited by means of data parallelism. In particular, multiple firings of an actor are concurrently processed and independent output tokens are calculated in a lane of a SIMD machine. We demonstrated the viability of our approach by running synthetic and real-world applications on two heterogeneous systems consisting of a multi-core CPU and multiple GPUs, achieving speed-ups of up to $41x$ compared to execution on a single core.

In the future, we plan to apply the proposed design flow to other OpenCL-capable platforms, e.g., the STHORM platform [16] or the Xeon Phi accelerator.

Acknowledgment

This work was supported by EU FP7 project EURETILE, under grant numbers 247846, and by NanoTera.ch with Swiss Confederation financing.

References

[1] "The OpenCL Specification," Khronos OpenCL Working Group, 2012.
[2] E. Lee and D. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.
[3] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele, "Scenario-Based Design Flow for Mapping Streaming Applications onto On-Chip Many-Core Systems," in *Proc. CASES*, 2012, pp. 71–80.
[4] K. Spafford, J. Meredith, and J. Vetter, "Maestro: Data Orchestration and Tuning for OpenCL Devices," in *Euro-Par 2010 - Parallel Processing*, ser. LNCS. Springer, 2010, vol. 6272, pp. 275–286.
[5] E. Sun, D. Schaa, R. Bagley *et al.*, "Enabling Task-level Scheduling on Heterogeneous Platforms," in *Proc. GPGPU*, 2012, pp. 84–93.
[6] P. Kegel, M. Steuwer, and S. Gorlatch, "dOpenCL: Towards a Uniform Programming Approach for Distributed Heterogeneous Multi-/Many-Core Systems," in *Proc. IPDPSW*, 2012, pp. 174–186.
[7] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink, "Compiling a High-Level Language for GPUs: (via Language Support for Architectures and Compilers)," in *Proc. PLDI*, 2012, pp. 1–12.
[8] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures," in *Proc. OOPSLA*, 2010, pp. 89–108.
[9] E. Bezati *et al.*, "High-Level Dataflow Design of Signal Processing Systems for Reconfigurable and Multicore Heterogeneous Platforms," *Journal of Real-Time Image Processing*, pp. 1–12, 2013.
[10] "CUDA Programming Guide," NVIDIA, 2008.
[11] A. Balevic and B. Kienhuis, "An Efficient Stream Buffer Mechanism for Dataflow Execution on Heterogeneous Platforms with GPUs," in *Proc. DFM*, 2011, pp. 53–57.
[12] ——, "KPN2GPU: An Approach for Discovery and Exploitation of Fine-grain Data Parallelism in Process Networks," *SIGARCH Comput. Archit. News*, vol. 39, no. 4, pp. 66–71, 2011.
[13] H. Jung *et al.*, "Automatic CUDA Code Synthesis Framework for Multicore CPU and GPU Architectures," in *Parallel Processing and Applied Mathematics*, ser. LNCS. Springer, 2012, pp. 579–588.
[14] A. H. Hormati *et al.*, "Sponge: Portable Stream Programming on Graphics Engines," in *Proc. ASPLOS*, 2011, pp. 381–392.
[15] W. Thies *et al.*, "StreamIt: A Language for Streaming Applications," in *Compiler Construction*, ser. LNCS. Springer, 2002, pp. 179–196.
[16] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an Ecosystem for a Scalable, Modular and High-Efficiency Embedded Computing Accelerator," in *Proc. DATE*, 2012, pp. 983–987.
[17] A. K. Singh *et al.*, "Mapping on Multi/Many-Core Systems: Survey of Current and Emerging Trends," in *Proc. DAC*, 2013.
[18] M. Mattavelli, S. Brunetton, and D. Mlynek, "A Parallel Multimedia Processor for Macroblock Based Compression Standards," in *Proc. Int'l Conf. on Image Processing*, 1997, pp. 570–573.
[19] J. Keinert and J. Teich, *Design of Image Processing Embedded Systems Using Multidimensional Data Flow*. Springer, 2011.
[20] W. Haid, L. Schor, K. Huang, I. Bacivarov, and L. Thiele, "Efficient Execution of Kahn Process Networks on Multi-Processor Systems Using Protothreads and Windowed FIFOs," in *Proc. ESTIMedia*, 2009, pp. 35–44.