ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Semester Thesis
at the Department of Information Technology
and Electrical Engineering

High-Level Programming Framework for Executing Streaming Applications on Heterogeneous OpenCL Platforms

Spring Term 2014

Matthias Baer

Advisors: Andreas Tretter
Lars Schor
Professor: Prof. Dr. Lothar Thiele

Zurich
June 26, 2014

Abstract

As the computer industry is reaching more and more limits regarding processor speed and transistor size, they have to come up with complex new architectures and more efficient use of the available processing power. For application developers this can be a difficult task, because they have to be aware of low-level hardware properties and there are many pitfalls to circumvent.
The Distributed Application Layer (DAL) framework developed at the Computer Engineering Lab provides application designers with a tool to simplify their work-flow and still providing a high level of flexibility and more importantly: parallelism. Running an application on multiple processing units in parallel without synchronization issues, is one of the key features of the DAL framework.

In the past, a new branch of the DAL framework got developed including the capability of executing applications on dedicated hardware like Graphics Processing Units (GPUs) or coprocessors by using OpenCL.
Even though this implementation is working as intended, there are still some drawbacks which delayed the integration of this branch into the main DAL framework.

This thesis progresses with integrating the OpenCL capabilities and adds some new specifications which allow a more generalized definition for DAL applications. As part of this process, some new programming constructs are introduced and the framework is extended with a transformation tool. This addition allows to produce different source code depending on the desired mapping.

The final evaluations of the newly implemented changes in the DAL framework show that no performance loss can be determined when using the generalized version of an application.

Acknowledgements

First of all I would like to thank Prof. Dr. Lothar Thiele for giving me the opportunity to write this semester thesis in his research group.

A big thank you goes out to my advisors Andreas Tretter and Lars Schor for their ongoing support during this thesis. The weekly meetings were a big help and the fact that there was always someone to ask if there were questions was great. It was inspiring to take part in such a project where your work is valuable for other people.

Contents

List of Figures

List of Listings

1 Introduction

1.1 Motivation

Today, the demand for increasing computing power leads to more and more complex processor designs and integrated solutions like video decoders for high-definition video streams.
Alternatively, there are ways to distribute the computational load of an application onto multiple processing units and to run them in parallel in order to increase performance. But this task is difficult and requires a lot of knowledge about the participating units and handling of communication between them. Parallelism also introduces the need for synchronization of the processes and this gets more and more unmanageable, the bigger an application is. Therefore it is desirable to have a framework for application designers which reduces their workload by managing the low-level parallelization and distribution, but still provides enough flexibility to successfully exploit the available hardware.
Such a software development framework has been developed at the Computer Engineering Lab. The Distributed Application Layer (DAL) framework allows the user to specify a parallel application using dataflow graphs and map these graphs onto a multi-processor platform.
As part of a master thesis by Tobias Scherer [1], another part got added recently, which allows to use the capabilities of a single machine even further by performing parts of the computation on Graphics Processing Units (GPUs) or coprocessors. This is achieved by using OpenCL, which acts as an interface to the different supported hardware solutions.

However, this addition to the framework still had some gaps that needed to be closed before it could be used by the community:

- The implementation has been written on a separate branch of the DAL framework, which was independent of the main development and therefore had to be reintegrated.
- In order to write an OpenCL process, the designer had to learn some additional programming constructs and be aware of how the framework interacts with this process. This is contradictory to the idea that low-level tasks should be handled by the framework.
- The support of multiple machines is one of the goals of the DAL framework, but the current OpenCL implementation only supports a single platform.
- Because there is no tool available for specifying the underlying architecture of an OpenCL application, this can be difficult to do manually and requires some insight knowledge about the identification of the available processing units.
- OpenCL platforms normally provide a hierarchical memory structure, but the current implementation only exploits the top-level memory of an OpenCL device.

Those were some of the problems which needed to be solved in order to integrate the OpenCL additions into the main branch of the DAL framework.

1.2 Contributions

The contributions of this semester thesis are as follows:

- The integration of the OpenCL capability into the main branch of the DAL framework.
- The systematization of the language extensions proposed by Tobias Scherer.
- The implementation of a code transformation tool that transforms generic source code into OpenCL capable processes or POSIX threads, according to the specified mapping.

1.3 Outline

This thesis is structured as follows: In Chapter 2, some background information is given to explain the basic concepts used in this thesis. The proposed design flow is described in Chapter 3 and an evaluation can be found in Chapter 4. Finally, Chapter 5 draws a conclusion and provides a short outlook.

2 Background

2.1 DAL Framework

A common approach to perform computations more efficiently today, is to distribute the workload to several independent machines, where each one performs another part of the calculation. At the Computer Engineering Lab, the Distributed Application Layer (DAL) framework [2] has been developed, which provides the user with a high-level interface to write distributed applications.

2.1.1 Kahn Process Networks

The underlying concept of the DAL framework relies on Kahn Process Networks (KPNs), which were first introduced by Gilles Kahn [3]. Process networks are modelled as directed graphs and used to describe applications by defining its separate processes and the relations between them. A graphical representation of a simple process network can be seen in Figure 2.1.

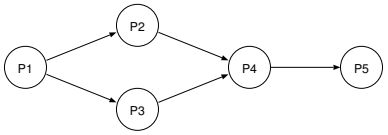


Figure 2.1: Simple process network with five processes and five channels.

All processes in a KPN are running independently of each other and communicate via First-In First-Out (FIFO) channels. Writing to such a channel has to be always successful because they are of infinite size, but reading can block the process if there is no data available. The data blocks exchanged between the processes are called **tokens** and their size is fixed for each channel. KPNs are deterministic [3], meaning that the same sequence of input tokens has to lead to the same output.

2.1.2 Mapping

By describing an application as a KPN, each process can be executed on a different processor as long as there exists a channel between them which they can use to communicate.
Defining which process runs on which processor is called **mapping** and can greatly influence the efficiency of an application.

The DAL framework uses KPNs to describe an application and it also allows to specify a mapping of the processes onto a multi-processor platform as seen in Figure 2.2.

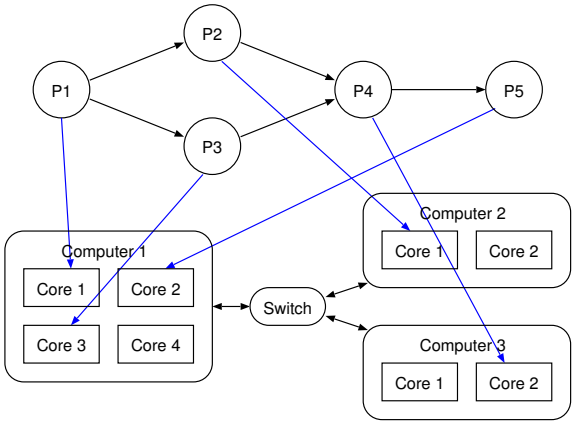


Figure 2.2: Process network with an architecture consisting of three machines (one quad-core and two dual-core) and some random mapping.

2.1.3 Synchronous Data Flow

In order to specify process networks which can be handled by OpenCL (see Section 2.2), Tobias Scherer proposed in his master thesis [1] to use the concept of Synchronous Data Flow (SDF) graphs. This is a special case of a KPN and makes the following restrictions: In each execution of a process, also called *firing*, the process uses and produces a fixed amount of tokens

and it is stateless, meaning that the result of a computation only depends on the current inputs of the process. In the context of an SDF, the processes are often called *actors*.
Figure 2.3 shows a graphical representation of an SDF, where the numbers indicate the amount of tokens consumed or generated in one cycle.

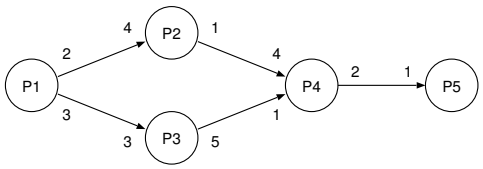


Figure 2.3: Simple Synchronous Data Flow graph with random token rates.

2.2 OpenCL

Another method to exploit the processing capability of a machine is to use the existing processing units like Graphics Processing Units (GPUs) or coprocessors for computations. This means that these units are no longer dedicated to one specific task like processing image data, but can also handle general purpose computations (more or less efficiently). In order to do so, several different languages have been developed, including NVIDIA's proprietary CUDA framework [4] and the OpenCL [5] GPU computing language.

OpenCL is an open standard adopted by many hardware vendors and is widely used for parallel computing. Its main concept consists of two parts: The **host**, which is responsible to distribute the work to the second part, the **devices**. Each device consists of one or more **compute units**, which are then further divided into **processing elements**. This hierarchy is illustrated in Figure 2.4.

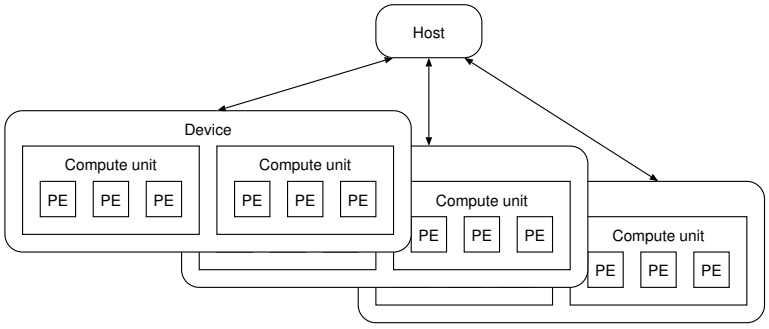


Figure 2.4: OpenCL hierarchy.

In software, the function to execute on an OpenCL device is called **kernel**. Each kernel has to know how much data it can read and write in one execution cycle, before it gets called. This is why the introduction of SDF graphs for OpenCL applications is reasonable, because the number of tokens which are being produced and consumed is known in advance.

A kernel can be executed on multiple processing elements and compute units to exploit the parallelism of an OpenCL device. If the used processing elements belong to the same compute unit, the executing kernels are called **work-items** and can make use of *data-parallelism* by running the same kernel code on distinct data. Different compute units can also contribute to data-parallelism if they execute the same kernel or they can establish *task-parallelism* when running different kernels. The collection of related work-items on a single compute unit is called a **work-group**. [6]

2.2.1 Blocks

The paper *"Exploiting the Parallelism of Heterogeneous Systems using Dataflow Graphs on Top of OpenCL"*[7] introduces another notion called **blocks**. In a case where some output tokens of a process need to be calculated altogether, they are grouped into a block. Figure 2.5 shows an example where each execution cycle consumes four tokens and also produces four tokens. Each output token depends on more than one input token and they need to be calculated in groups of two.

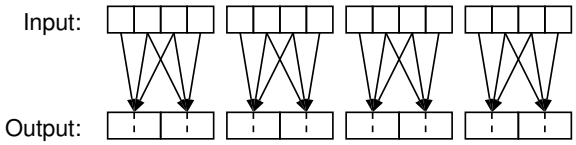


Figure 2.5: Input-output relation with two blocks per execution cycle.

Depending on the architecture, there are several methods in which order to write the output blocks to maximize the efficiency of the memory accesses. The most straightforward option is to write the blocks following each other by the same work-item, illustrated in Figure 2.6.

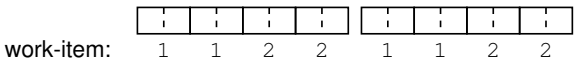


Figure 2.6: Example *block* access with two available work-items. The numbers indicate which work-item writes which output block. Each block consists of two tokens and 8 tokens have to be written in one execution cycle.

The other method mostly used for GPUs, is the *strided* method. The output blocks written by one work-item are spaced equally by the number of actual work-items in a work-group. Figure 2.7 shows this method with the same setup as above.

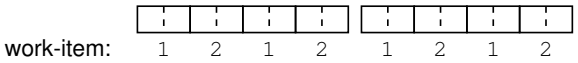


Figure 2.7: Example *strided* access.

3 Design Flow

3.1 Overview

In order to successfully run an application with the DAL framework, the designer has to know how to specify its behaviour.

The following sections describe the steps necessary to create and run an application with the resulting framework of this thesis. We assume a simple program with a process network illustrated in Figure 3.1:

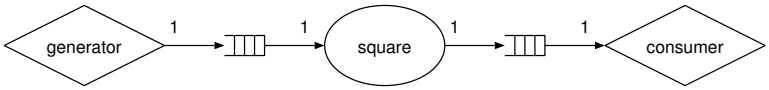


Figure 3.1: Simple producer - consumer Synchronous Data Flow (SDF) graph.

The **generator** process is generating some floating point numbers and forwarding them to the **square** process via a FIFO buffer. This process is squaring the given number and outputs it to another FIFO, which connects to the **consumer** process. There the received number is printed on the screen.

3.2 XML Specification

There are several specifications needed to execute an application with the DAL framework. Three of those are written in Extensible Markup Language (XML) and describe the **process network**, the underlying **architecture** of the executing machine and the **mapping** of the processes to the available hardware. The additions added by Tobias Scherer and adapted by this thesis are highlighted in the description by using *italic font*.

3.2.1 Process Network

The specification for the process network describes the connections between the different processes. It consists of three different parts:

- The description of the different processes in the network where their input and output ports are defined together with the location of the underlying source code file. Listing 3.1 shows an example for the square process in our simple program.

Listing 3.1: Example *square* Process.

```
1 <process name="square" type="local">
2   <port type="input" name="in" tokenrate="1"/>
3   <port type="output" name="out" tokenrate="1" blocksize="1"/>
4   <source type="c" location="square.c"/>
5 </process>
```

  - The `process` tag specifies a process with a given `name` and some `type`. The `type` attribute does not influence on the behaviour described in this thesis.
  - The `port` tag adds an `input` or `output` port with `name` to the process according to the `type` attribute. *The* ***`tokenrate`*** *attribute defines the number of tokens consumed/produced in each execution cycle. Output ports can also define a* ***`blocksize`*** *which refers to the number of tokens in one output block (see Section 2.2.1).*
  - The `source` tag is required to specify the `location` of the source code for the process. The `type` attribute defines the source code language and can be: `c`, `c++` *or* ***`openclc`*** *(see below).*

  The `openclc` type is used to directly specify OpenCL kernel code without performing any further transformations. However, this is not required for the new framework, as OpenCL kernels can be generated from `c` type source code files.

- The definition of the channels used in the process network. They are generally defined as FIFOs with one input and one output port. Additionally they specify the size of a single token, as well as how many tokens the channel can store, because the concept of an infinite FIFO size is not applicable to reality. An example channel is shown in Listing 3.2 with a token size of 4 bytes, which corresponds to a `float`.

Listing 3.2: Example FIFO Channel.

```
<sw_channel type="fifo" size="10" tokensize="4" name="C1">
  <port type="input" name="in"/>
  <port type="output" name="out"/>
</sw_channel>
```

  - The `sw_channel` tag specifies a software channel with a given `name` and some `type`. The `type` can be either `fifo` or `wfifo` (windowed-FIFO [8]). With the `size` attribute, the maximum number of stored tokens within the channel can be specified *and with* ***`tokensize`*** *the size of a single token can be set in bytes.*
  - The `port` tag simply allows to add an `input` or `output` port with `name` to the channel according to the `type` attribute.

- The connection between the processes and the channels. Listing 3.3 describes the connection between the FIFO and the square process.

Listing 3.3: Example Channel to Process Connection.

```
<connection name="channel-square">
  <origin name="C1">
    <port name="out"/>
  </origin>
  <target name="square">
    <port name="in"/>
  </target>
</connection>
```

  - The `connection` tag specifies a connection with a given `name` between a channel and a process and vice versa.
  - The `origin` tag selects the channel/process with `name` as the origin of the connection.
  - The `target` tag selects the channel/process with `name` as the target of the connection.
  - The `port` tag selects the port with `name` from the origin/target which needs to be connected.

The complete process network XML file describing Figure 3.1 can be found under Appendix B as Listing B.1.

3.2.2 Architecture

In order to define a mapping for an application, the framework has to know about the available hardware. This is currently done manually by providing an architecture XML file and has a similar structure as the process network:

- The type of each processing unit needs to be specified, together with its capabilities. Listing 3.4 describes one core of a Central Processing Unit (CPU) which can handle Portable Operating System Interface (POSIX) code as well as OpenCL kernels. Each core of a CPU is treated like a separate processor, because it is an individual processing unit. The additional GPU described, is only capable of executing OpenCL kernels.

Listing 3.4: Example Processing Units.

```
1  <processor name="core_0" type="RISC" id="0">
2    <port name="port1"/>
3    <capability name="posix" identifier="0"/>
4    <capability name="opencl" identifier="CPU_4098_Intel(R)Core(TM)
         i7-2600KCPU@3.40GHz_DEV1"/>
5  </processor>
6
7  <processor name="gpu" type="GPU" id="1">
8    <port name="port1"/>
9    <capability name="opencl" identifier="GPU_4098_Capeverde_DEV1"/
         >
10 </processor>
```

  - The `processor` tag specifies a processing element with a given `name` and some `type`. The `type` can be one of the following: `RISC`, `DSP` or `GPU` and is used to check the accuracy of a defined mapping later. The `id` attribute is used to internally assign a core identifier.
  - The `port` tag adds an interface with `name` to the processor which allows communication.
  - *The* ***`capability`*** *tag specifies whether the processor is capable of handling* ***`posix`*** *code or* ***`opencl`*** *kernels according to the* ***`name`*** *attribute. The* ***`identifier`*** *attribute is for* ***`posix`*** *capabilities a simple number, but for* ***`opencl`*** *it is more complex and explained below.*

  The OpenCL `identifier` has the following structure:
  `<type>_<id>_<name>_DEV<number>`

  `<type>` is the device type which can be either `CPU`, `GPU` or `ACC` (accelerator) at the moment.

  `<id>` is the device identifier.

`<name>` is the device name without any spaces or underscores.

`<number>` is the number of the device, if there is more than one with the same name.

For an AMD Radeon HD 7750 [9] GPU (Codename: *Cape Verde*), the `identifier` would read as: `GPU_4098_Capeverde_DEV1`.

- The shared memory of the machine is simply defined by enumerating the ports used to connect to the processing units as seen in Listing 3.5. The memory of a GPU does not need to be specified, because OpenCL takes care of distributing the data properly.

Listing 3.5: Example Shared Memory.

```
1 <shared name="localhost">
2   <port name="port1"/>
3   <port name="port2"/>
4 </shared>
```

  - The `shared` tag specifies a shared memory with a given `name`.
  - The `port` tag adds an interface with `name` to the shared memory which allows communication.

- The actual connections between the processing units and the shared memory are similar to the connections of the process network. Listing 3.6 shows the connection between the GPU and the shared memory.

Listing 3.6: Example Link between GPU and Memory.

```
1 <link name="link_1">
2   <end_point_1 name="gpu">
3     <port name="port1"/>
4   </end_point_1>
5   <end_point_2 name="localhost">
6     <port name="port2"/>
7   </end_point_2>
8 </link>
```

  - The `link` tag specifies a link with a given `name` between a processor and a shared memory.
  - The `end_point_1` and `end_point_2` tags select the processor/shared memory with `name` as one of the end points of the connection.
  - The `port` tag selects the port with `name` from the processor/shared memory which needs to be connected.

In addition, Listing B.2 describes the architecture of a simple quad-core processor with an additional GPU.

3.2.3 Mapping

The actual mapping of the processes to the available processing units is specified in a separate XML file. This is also used to define some specific properties for a process depending on its mapping. Listing 3.7 shows an example binding for the square process onto the GPU. It is defined to use the OpenCL version of the process, which results into a transformation of the source code into an OpenCL kernel if it is not yet specified as such. There are also two mapping specific attributes given: the number of work-items per work-group and the number of work-groups.

Listing 3.7: Example Mapping for the *square* Process.

```
1 <binding name="square">
2   <process name="square"/>
3   <processor name="gpu"/>
4   <target>
5     <opencl workitems="256" workgroups="1">
6       <port name="out" access="strided"/>
7     </opencl>
8   </target>
9 </binding>
```

- The `binding` tag specifies a binding with `name`.
- The `process` tag selects the process to bind by using its `name`.
- The `processor` tag selects which processor to map onto depending on its `name`.
- *The* `target` *tag is used to select which implementation to use of the specified process.*
- *The tag within* `target` *can be either* `posix` *which has no further attributes and is the default, or it can be* `opencl` *which can specify the number of* `workitems` *per work-group and the number of* `workgroups`*.*
- *For* `opencl` *tags, it is possible to specify the output access method (see Section 2.2.1) in a* `port` *tag. The* `name` *attribute identifies the port and* `access` *can either be* `block` *or* `strided`*.*

A full example mapping for the squaring application can be found in Listing B.3.

3.3 Process Source Code

To describe the functionality of the individual processes, DAL uses C or C++. The paper *"Exploiting the Parallelism of Heterogeneous Systems using Dataflow Graphs on Top of OpenCL"*[7] proposes some changes to the source code definition in order to allow translation of a process description into either a POSIX thread or an OpenCL kernel. Those changes were adapted and implemented as part of this thesis and are described in this section.

3.3.1 Header File

Each process has to be provided with a header file like the one corresponding to the square process in Listing 3.8.

Listing 3.8: Example *square* Header File.

```
1  #ifndef SQUARE_H
2  #define SQUARE_H
3
4  #include <dal.h>
5
6  #define PORT_in  "in"
7  #define PORT_out "out"
8
9  typedef float TOKEN_in_t;
10 typedef float TOKEN_out_t;
11
12 #define TOKEN_in_RATE  1
13 #define TOKEN_out_RATE 1
14 #define BLOCK_out_SIZE 1
15 #define BLOCK_out_COUNT (TOKEN_out_RATE / BLOCK_out_SIZE)
16
17 typedef struct _local_states {
18 } Square_State;
19
20 void square_init(DALProcess *);
21 int square_fire(DALProcess *);
22 void square_finish(DALProcess *);
23
24 #endif
```

The properties of the header file are as follows:

- It has to include the *dal.h* header file which declares some of the data types and functions used in a process. Other header files can be included, but may cause problems when transforming the process into an OpenCL kernel.
- Each port of the process has to be defined as `PORT_<name> "<name>"` where `<name>` represents the name of the port used in the XML specification.

- The tokens sent and received by a process have to be defined as a new type `TOKEN_<name>_t` with `<name>` again matching the port name for this token. The size of this data type should match the token size specified in the XML file.

- There need to be some additional parameters defined, depending on the port type. Input ports only have to set its token rate as `TOKEN_<name>_RATE` (`<name>` always being the port name) and output ports have to define its token rate as well as its block size (`BLOCK_<name>_SIZE`) and the number of blocks (`BLOCK_<name>_COUNT`) which is usually simply `(TOKEN_<name>_RATE / BLOCK_<name>_SIZE)`. For more information about blocks, see Section 2.2.1.

- If the process is not intended to be used as an OpenCL kernel, it can store some internal state variables inside the `_local_states` structure. Its name should be `<Name>_State` with `<Name>` being the camel case variant of the process name.

- Finally, the header should contain the declaration of the three process functions `<name>_init`, `<name>_fire` and `<name>_finish` where `<name>` is the process name. All three processes require a pointer to a `DALProcess struct` as an argument.
  The *init* and *finish* functions are called on creation and destruction of the process and should not return any value. The *fire* function is called repeatedly and contains the main functionality of the process. It has to return an integer number indicating whether it is able to process more data (`0`) or if it should not be called anymore (`1`). This is only applicable for processes later used as POSIX threads and the return value is ignored in OpenCL kernels, because they cannot return any value at all. For compatibility reasons though, the return value is still required.

3.3.2 Source File

The actual source file of a process defines its behaviour and Listing 3.9 shows an example for the square process.

Listing 3.9: Example *square* Source File.

```
1  #include "square.h"
2
3  void square_init(DALProcess *p)
4  {
5  }
6
7  int square_fire(DALProcess *p)
8  {
9    TOKEN_in_t *rbuf = (TOKEN_in_t *)DAL_read_begin(PORT_in, sizeof(
       TOKEN_in_t), TOKEN_in_RATE, p);
10
11   DAL_foreach (blk : PORT_out)
12   {
13     TOKEN_out_t *wbuf = (TOKEN_out_t *)DAL_write_begin(PORT_out,
         sizeof(TOKEN_out_t), TOKEN_out_RATE, BLOCK_out_SIZE, blk, p);
14     *wbuf = rbuf[blk] * rbuf[blk];
15     DAL_write_end(PORT_out, wbuf, p);
16   }
17
18   DAL_read_end(PORT_in, rbuf, p);
19
20   return 0;
21 }
22
23 void square_finish(DALProcess *p)
24 {
25 }
```

Source files have to include their header file at the beginning and may also include other header files. System headers can be included too, but they are ignored for OpenCL kernels.

The *init*, *fire* and *finish* functions declared in the header file need to be defined here and their return values and arguments have to be the same as described in the section above.
There are several functions provided by DAL to interact with the other processes via the corresponding ports and they are listed below:

- `void *DAL_read_begin(int port, int tokensize, int tokenrate, DALProcess *p)`
  This function is used to read data of size `tokensize` from a `port` with a given `tokenrate`.
  - `int port`: should be one of the `PORT_<name>` constants defined in the header file
  - `int tokensize`: usually simply uses the `sizeof` operation on the `TOKEN_<name>_t` type

  - `int tokenrate`: should be one of the `TOKEN_<name>_RATE` constants matching the used port
  - `DALProcess *p`: used for compatibility reasons and equals the `DALProcess` pointer given to the calling function

  The return value of `DAL_read_begin` should be assigned to a variable of type `TOKEN_<name>_t` according to the port used.

- `void DAL_read_end(int port, void *buf, DALProcess *p)`
  This function ends the read procedure for `port` and renders the `buf` pointer useless, therefore it should be called near the end of the function.

  - `int port`: has to be the same `PORT_<name>` constant as used for the corresponding `DAL_read_begin`
  - `void *buf`: the pointer which stored the data returned by `DAL_read_begin`
  - `DALProcess *p`: used for compatibility reasons and equals the `DALProcess` pointer given to the calling function

- `void *DAL_write_begin(int port, int tokensize, int tokenrate, int blocksize, int blk_ptr, DALProcess *p)`
  This function is used to write data of size `tokensize` to a `port` with a given `tokenrate` and a given `blocksize`.

  - `int port`: should be one of the `PORT_<name>` constants defined in the header file
  - `int tokensize`: usually simply uses the `sizeof` operation on the `TOKEN_<name>_t` type
  - `int tokenrate`: should be one of the `TOKEN_<name>_RATE` constants matching the used port
  - `int blocksize`: normally uses the `BLOCK_<name>_SIZE` constant matching the used port
  - `int blk_ptr`: used to define which block of the output port is being written
  - `DALProcess *p`: used for compatibility reasons and equals the `DALProcess` pointer given to the calling function

  The return value of `DAL_write_begin` should be assigned to a variable of type `TOKEN_<name>_t` according to the port used and is a pointer to the write buffer.

- `void DAL_write_end(int port, void *buf, DALProcess *p)`
  This function actually writes the data from the `buf` pointer to the `port`.
  - `int port`: has to be the same `PORT_<name>` constant as used for the corresponding `DAL_write_begin`
  - `void *buf`: the pointer to the write buffer returned by `DAL_write_begin`
  - `DALProcess *p`: used for compatibility reasons and equals the `DALProcess` pointer given to the calling function

Apart from these functions, there exists an other construct: the `DAL_foreach` loop. It is used to iterate through all output blocks (see Section 2.2.1) using the method specified in the mapping XML file. The syntax is as follows:

```
DAL_foreach (<block_id> : <port>) {
}
```

`<block_id>` has to be an unused variable name, because it is automatically transformed into a proper variable type.

`<port>` has to be one of the `PORT_<name>` output ports defined in the header file.

3.4 Framework Execution

After specifying all application specific properties and defining the behaviour of each process, the developer is able to launch the building process. The steps performed by the DAL framework to execute an application are listed below:

1. **Preparation**: The directory structure for the application is created and all sources and XML specifications are copied there.
2. **XML Validation**: Each XML file is checked for syntactical errors.
3. **XML Flattening**: Iterations in the XML specification are replaced by the actual information and variables are substituted in this process, resulting in an entirely static version of the specification.
4. **XML Checking**: All XML files are checked for logical errors, such as missing connections in the process network or impossible mappings.
5. **Controller Generation**: The controller which handles the distribution and connection of the individual processes is generated and the process sources are adapted.
6. **Process Generation**: Each process runs through some further transformations to ensure correct identification within DAL.
7. **Execution**: Finally, the application is executed, which is necessarily preceded by the compilation of the generated source files.

This thesis mostly changed the behaviour of the controller and process generation from step 5 and 6. For executing OpenCL applications, a new process generator got added which uses the SDF implementation from Tobias Scherer for the process network. It got simplified by removing unnecessary code duplicates and an OpenCL test was added which only executes when OpenCL is actually installed.

Another important step was to introduce a code transformation tool which handles the translation from the syntax explained in Section 3.3 into the desired target code. Because of the internal structure of the DAL framework, the transformation step got added to the controller generation, where all the necessary information is available.
The required source code adaptions depending on the specified mapping are explained below.

3.4.1 POSIX Transformation

In order to execute the application from Figure 3.1 with all processes as POSIX threads, there is essentially only one substitution to be made: the `DAL_foreach` construct has to be replaced by an actual `for` loop. All the other functions like `DAL_read_begin` and `DAL_write_begin` are simply implemented as a slightly adjusted version of the original DAL counterpart of these functions.

3.4.2 OpenCL Transformation

The more important transformations take place if a process has to be executed as an OpenCL kernel. The first change to be made is the replacement of the source file extension, because OpenCL kernels are declared to be `*.cl` files.
In order to actually transform a process into an OpenCL kernel, the *init*, *fire* and *finish* functions need to be changed, because all ports of a process have to be assigned as arguments to the kernel.
The `DAL_foreach` loop is replaced by the correct `for` implementation for either *strided* or *block* output mode (see Section 2.2.1).
The `DAL_read_begin` and `DAL_write_begin` functions are replaced with the corresponding pointer arithmetic by using preprocessor macros and the `DAL_read_end`/`DAL_write_end` functions are simply removed.

Listing 3.10 and 3.11 show the transformed header and source file of the square process from Listing 3.8 and 3.9 after the preprocessor has run.

Listing 3.10: Transformed *square* OpenCL Header File.

```
1  #ifndef SQUARE_H
2  #define SQUARE_H
3
4  #include <dalMacros.h>
5
6  #define PORT_in  "in"
7  #define PORT_out "out"
8
9  typedef __global float TOKEN_in_t;
10 typedef __global float TOKEN_out_t;
11
12 #define TOKEN_in_RATE  1
13 #define TOKEN_out_RATE 1
14 #define BLOCK_out_SIZE 1
15 #define BLOCK_out_COUNT (TOKEN_out_RATE / BLOCK_out_SIZE)
16
17 typedef struct _local_states {
18 } Square_State;
19
20 #endif
```

Listing 3.11: Transformed *square* OpenCL Source File.

```
1  #include "square.h"
2
3  __kernel void square_init(TOKEN_in_t *PORT_in_name,
4                            __constant uint *PORT_in_size,
5                            TOKEN_out_t *PORT_out_name,
6                            __constant uint *PORT_out_size)
7  {
8  }
9
10 __kernel void square_fire(TOKEN_in_t *PORT_in_name,
11                           __constant uint *PORT_in_size,
12                           TOKEN_out_t *PORT_out_name,
13                           __constant uint *PORT_out_size)
14 {
15   TOKEN_in_t *rbuf = (TOKEN_in_t *)(PORT_in_name + get_group_id(0) *
         TOKEN_in_RATE);
16
17   for (int blk = get_local_id(0); blk < BLOCK_out_COUNT; blk +=
         get_local_size(0))
18   {
19     TOKEN_out_t *wbuf = (TOKEN_out_t *)(PORT_out_name + get_group_id
           (0) * TOKEN_out_RATE + blk * BLOCK_out_SIZE);
20     *wbuf = rbuf[blk] * rbuf[blk];
21   }
22
23   return 0;
24 }
25
26 __kernel void square_finish(TOKEN_in_t *PORT_in_name,
27                             __constant uint *PORT_in_size,
28                             TOKEN_out_t *PORT_out_name,
29                             __constant uint *PORT_out_size)
30 {
31 }
```

4 Evaluation

4.1 Overview

In order to see whether or not the newly introduced transformation of the source code into either POSIX code or OpenCL kernels was successful, a comparison between the original code from Tobias Scherer and the new framework was done. This is explained in Section 4.3.1 and shows that no significant performance loss could be detected.
In Section 4.3.2 some further experiments are carried out to test the functionality and ease of use of the framework.

To see the benefit of the changed specification, Section 4.4 finally illustrates the simplicity of adapting an existing DAL application to the new syntax.

4.2 Hardware Setup

The machine used to run all of the applications had the following properties:

CPU: Intel Core i7-2600K [10] (4 Cores, 3.40 GHz, 8 MB Cache) with Hyper-Threading enabled.

GPU: AMD Radeon HD 7750 [9] (8 compute units each 256 processing elements).

Memory: 16 GB RAM

Operating System: Arch Linux 3.14.6-1-ARCH x86_64.

4.3 Performance Evaluation

The example application used to measure the performance, is illustrated in Figure 4.1 and represents a Motion JPEG (MJPEG) decoder with some additional filters.

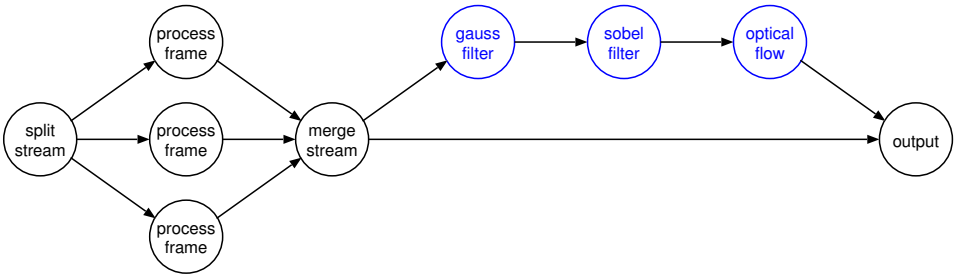


Figure 4.1: Video-processing process network with applied filters.

The three filter processes *gaussfilter*, *sobelfilter* and *opticalflow* are perfectly suitable for execution as OpenCL kernels, because they allow high data-parallelism. An existing version of the application already provided an OpenCL implementation of those filters and for good comparison, all three processes got rewritten with the newly introduced syntax.

4.3.1 Framework Comparison

In order to compare the performance of the original implementation from Tobias Scherer with native OpenCL kernel code to the generated version from the new syntax, the same mapping is used. By measuring the time between the first arriving output frame and the last one, an overall comparison of the two implementations is achieved. The setup time at the beginning of the application is ignored, although the OpenCL implementation might add some delay there.

As an initial benchmark, all three filters of the video-processing benchmark are mapped to the GPU and executed with 256 work-items per work-group and 64 work-groups. Table 4.1 lists the measurements of five separate executions for the original code from Tobias Scherer and the rewritten implementation of this thesis.

The results are very close and they show that the new implementation can keep up with the original OpenCL version.

	Original	New syntax
	2.156 s	2.151 s
	2.155 s	2.148 s
	2.161 s	2.148 s
	2.160 s	2.153 s
	2.164 s	2.147 s
Average:	2.159 s	2.149 s

Table 4.1: Five separate executions with all filters mapped on the GPU.

4.3.2 Mapping Comparisons

To perform further tests of the new framework, the video-processing application is executed with different mappings and variable OpenCL properties.

Figure 4.2 shows the results of those experiments where the advantage of using OpenCL with some elaborate mapping is clearly seen. For those measurements, all three filters are mapped on different cores of the CPU and every change can be done in the XML specification. In this example when using the CPU as OpenCL device, no performance improvement could be seen when increasing the number of work-items.

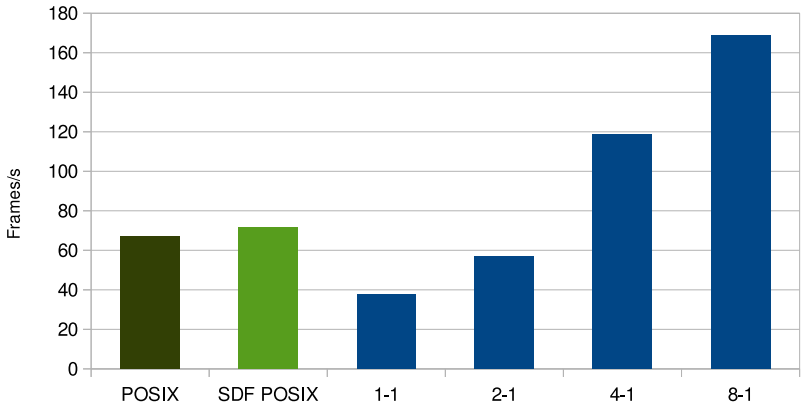


Figure 4.2: Different generators and mappings for the video-processing benchmark. On the far left, the application is generated with the standard POSIX generator. The second bar shows the result for the same mapping, but using the OpenCL generator where the channels are considered to be SDF FIFOs. The four remaining results are generated by the OpenCL generator as well, but now the actual OpenCL kernels are used. The numbers indicate the following: #work-groups-#work-items.

The performance when using the generated OpenCL kernels with very few work-groups is significantly worse than the POSIX version which essentially makes use of the same computational resources. The reason for this seems to be the overhead produced by the existing channel implementation. Each firing needs to allocate host memory and copies the required data into this memory location. By increasing the parallelism with the number of work-groups, this overhead has less impact because the number of memory allocations is reduced.

4.4 Adaptability

In order to see whether the newly introduced syntax and the corresponding specifications are easily applicable to existing DAL applications, another benchmark got rewritten. It is an ultrasound application developed by Harshavardhan Pandit [11] where the existing process network got slightly changed to conform to the SDF properties.

Applying the new syntax to one of the processes originally written as a POSIX thread, can be done within several hours. The result is an application which can run one of its processes on any OpenCL enabled device available.

5 Conclusion

5.1 Summary

The main purpose of this thesis was to enable OpenCL functionality within the DAL framework. This was achieved by integrating the OpenCL generator developed by Tobias Scherer into the current DAL version while maintaining full compatibility of already existing applications which use the framework.

To improve the ease of use of the existing OpenCL implementation, the process specification got extended with some more flexible functions for channel communication. Those functions together with a newly introduced loop construct also enable control over OpenCL specific data-parallelism.

This extension of the source code specification as well as some additions in the architecture and mapping specification, lead to the introduction of a code transformator. The transformator is capable of generating either POSIX code or OpenCL kernel code from a single source file, depending on the specified mapping.

In order to test the functionality of the transformator, the OpenCL part of an existing application got rewritten and the performance of the original implementation was compared to the newly generated version of the application. The evaluation then confirmed that no performance loss could be determined.

5.2 Outlook

Even though the OpenCL generator got added successfully and the code transformation is working properly, there are some limitations for application designers when using the new syntax. Some of the known barriers are listed below, together with possible solutions:

- By defining processes in a more generalized manner, some functions which make use of the underlying hardware architecture cannot be used, which leads to a slightly less efficient execution.

  This is considered to be a trade-off for generalizing process descriptions. It is difficult to solve and would possibly decrease the simplification aspect of the framework.

- The use of process internal variables which are stored between multiple executions are not allowed due to the fact that SDF graphs are stateless. Furthermore, OpenCL kernels are not able to keep data over multiple executions.

  This is a general problem when using SDF graphs and would require a conceptual change in the process network description. However, storing variables could be achieved by inserting feedback loops into the process network to simulate the desired behaviour.

- Processes need to specify each input and output port individually in the XML specification and in its source file, which disallows the use of port arrays where multiple ports are defined at once.

  This problem can be avoided by using the newly introduced parallelism and specifying the process network without such port arrays. The desired duplication of the processes can then be achieved by adapting the source code and determining an appropriate mapping.

- The idea of distributing an OpenCL application over several different machines does not yet apply to the present framework implementation.

  However, the flexibility of the framework should allow fast and straightforward insertion of this functionality as future work.

A

List of Acronyms

CPU	Central Processing Unit.
DAL	Distributed Application Layer.
DSP	Digital Signal Processor.
FIFO	First-In First-Out.
GPU	Graphics Processing Unit.
KPN	Kahn Process Network.
MJPEG	Motion JPEG.
POSIX	Portable Operating System Interface.
RISC	Reduced Instruction Set Computer.
SDF	Synchronous Data Flow.
XML	Extensible Markup Language.

B

Example Source Files

Listing B.1: Example Process Network XML

```
<processnetwork name="Squaring">
  <process name="generator" type="io">
    <port type="output" name="out"/>
    <source type="c" location="generator.c"/>
  </process>
  <process name="consumer" type="io">
    <port type="input" name="in"/>
    <source type="c" location="consumer.c"/>
  </process>
  <process name="square" type="local">
    <port type="input" name="in" tokenrate="1"/>
    <port type="output" name="out" tokenrate="1" blocksize="1"/>
    <source type="c" location="square.c"/>
  </process>

  <sw_channel type="fifo" size="10" tokensize="4" name="C1">
    <port type="input" name="in"/>
    <port type="output" name="out"/>
  </sw_channel>
  <sw_channel type="fifo" size="10" tokensize="4" name="C2">
    <port type="input" name="in"/>
    <port type="output" name="out"/>
  </sw_channel>

  <connection name="g-c">
    <origin name="generator">
      <port name="out"/>
    </origin>
    <target name="C1">
      <port name="in"/>
    </target>
  </connection>
  <connection name="c-s">
```

```
    <origin name="C1">
      <port name="out"/>
    </origin>
    <target name="square">
      <port name="in"/>
    </target>
  </connection>
  <connection name="s-c">
    <origin name="square">
      <port name="out"/>
    </origin>
    <target name="C2">
      <port name="in"/>
    </target>
  </connection>
  <connection name="c-c">
    <origin name="C2">
      <port name="out"/>
    </origin>
    <target name="consumer">
      <port name="in"/>
    </target>
  </connection>
</processnetwork>
```

Listing B.2: Example Architecture XML

```
<architecture name="Quad-core␣platform␣with␣GPU">
  <processor name="core_0" type="RISC" id="0">
    <port name="port1"/>
    <capability name="posix" identifier="0"/>
    <capability name="opencl" identifier="CPU_4098_Intel(R)Core(TM)i7
        -2600KCPU@3.40GHz_DEV1"/>
  </processor>
  <processor name="core_1" type="RISC" id="1">
    <port name="port1"/>
    <capability name="posix" identifier="1"/>
    <capability name="opencl" identifier="CPU_4098_Intel(R)Core(TM)i7
        -2600KCPU@3.40GHz_DEV1"/>
  </processor>
  <processor name="core_2" type="RISC" id="2">
    <port name="port1"/>
    <capability name="posix" identifier="2"/>
    <capability name="opencl" identifier="CPU_4098_Intel(R)Core(TM)i7
        -2600KCPU@3.40GHz_DEV1"/>
  </processor>
  <processor name="core_3" type="RISC" id="3" substitute="1">
    <port name="port1"/>
    <capability name="posix" identifier="3"/>
    <capability name="opencl" identifier="CPU_4098_Intel(R)Core(TM)i7
        -2600KCPU@3.40GHz_DEV1"/>
  </processor>

  <processor name="gpu" type="GPU" id="4">
    <port name="port1"/>
    <capability name="opencl" identifier="GPU_4098_Capeverde_DEV1"/>
  </processor>

  <shared name="localhost">
    <port name="port1"/>
    <port name="port2"/>
    <port name="port3"/>
    <port name="port4"/>
    <port name="port5"/>
  </shared>

  <link name="link_1">
    <end_point_1 name="core_0">
      <port name="port1"/>
    </end_point_1>
    <end_point_2 name="localhost">
      <port name="port1"/>
    </end_point_2>
  </link>
  <link name="link_2">
    <end_point_1 name="core_1">
      <port name="port1"/>
    </end_point_1>
    <end_point_2 name="localhost">
      <port name="port2"/>
    </end_point_2>
  </link>
  <link name="link_3">
    <end_point_1 name="core_2">
      <port name="port1"/>
    </end_point_1>
    <end_point_2 name="localhost">
      <port name="port3"/>
```

```
    </end_point_2>
  </link>
  <link name="link_4">
    <end_point_1 name="core_3">
      <port name="port1"/>
    </end_point_1>
    <end_point_2 name="localhost">
      <port name="port4"/>
    </end_point_2>
  </link>
  <link name="link_5">
    <end_point_1 name="gpu">
      <port name="port1"/>
    </end_point_1>
    <end_point_2 name="localhost">
      <port name="port5"/>
    </end_point_2>
  </link>
</architecture>
```

Listing B.3: Example Mapping XML

```
<mapping name="Mapping">
  <binding name="generator">
    <process name="generator"/>
    <processor name="core_2"/>
    <target>
      <posix/>
    </target>
  </binding>
  <binding name="square">
    <process name="square"/>
    <processor name="gpu"/>
    <target>
      <opencl workitems="256" workgroups="1">
        <port name="out" access="strided"/>
      </opencl>
    </target>
  </binding>
  <binding name="consumer">
    <process name="consumer"/>
    <processor name="core_2"/>
    <target>
      <posix/>
    </target>
  </binding>
</mapping>
```

C

Presentation Slides

High-Level Programming Framework for Executing Streaming Applications on Heterogeneous OpenCL Platforms

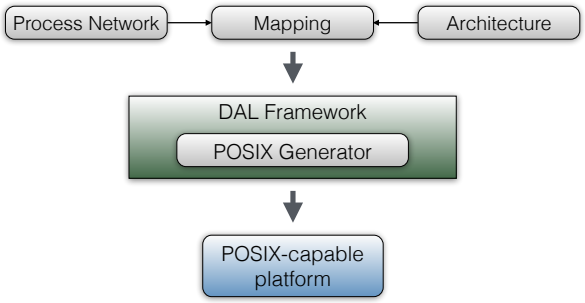
Semester Thesis of Matthias Baer
advised by Andreas Tretter and Lars Schor

Motivation

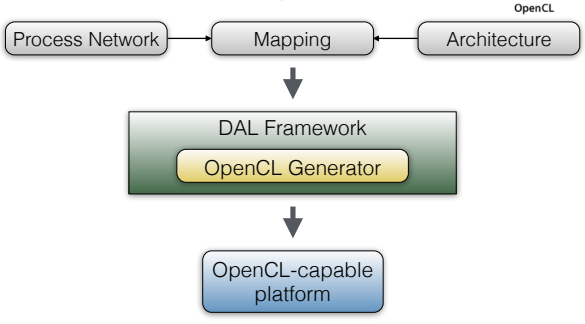
OpenCL

- Supports multiple devices (CPUs, GPUs, accelerators, …)
- Framework for low-level programming
- OpenCL-C language similar to C99
- Portable, but not performance portable

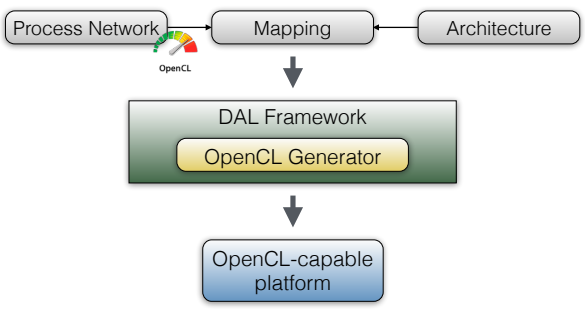
DAL Framework



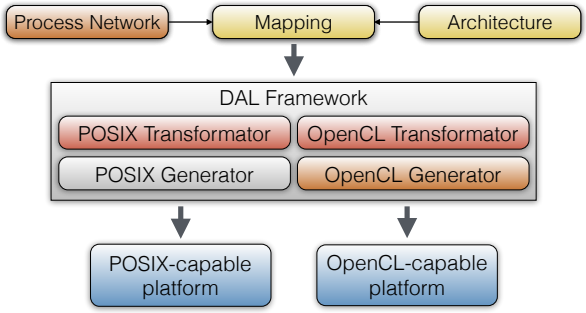
DAL Framework - OpenCL



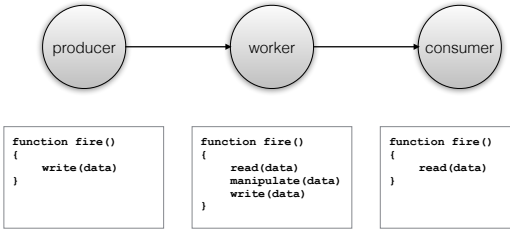
DAL Framework - OpenCL



Contributions



DAL Process Network Specification

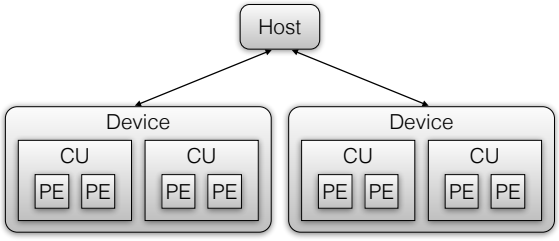



```
function fire()
{
    write(data)
}
```

```
function fire()
{
    read(data)
    manipulate(data)
    write(data)
}
```

```
function fire()
{
    read(data)
}
```

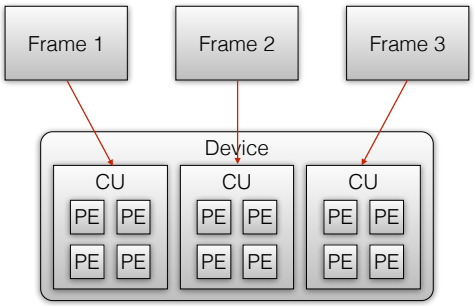

OpenCL Device Hierarchy



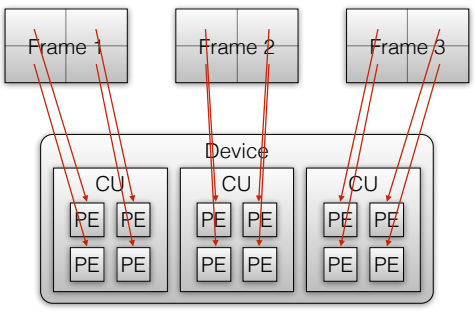
CU: Compute Unit

PE: Processing Element

OpenCL Execution Model



OpenCL Execution Model



OpenCL DAL Process

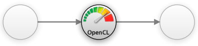

```
__kernel void fire(__global float *in,
                   __global float *out)
{
    int gid = get_group_id(0);
    int lid = get_local_id(0);
    int lsz = get_local_size(0);

    __global float *rbuf = in + gid*TOKEN_IN_RATE;

    for (int blk = lid; blk < BLOCK_OUT_COUNT; blk += lsz)
    {
        __global float *wbuf = out + gid*TOKEN_OUT_RATE
            + blk*BLOCK_OUT_SIZE;

        *wbuf = rbuf[blk] * rbuf[blk];
    }
}
```

OpenCL DAL Process




```
__kernel void fire(__global float *in,
                   __global float *out)
{
    int gid = get_group_id(0);
    int lid = get_local_id(0);
    int lsz = get_local_size(0);

    __global float *rbuf = in ...;

    for (int blk = ... _COUNT; blk += lsz)
    {
        __global ... out + gid*TOKEN_OUT_RATE
            + blk*B..._OUT_SIZE;

        *wbuf = rbuf[blk] * rbuf[blk];
    }
}
```

No high-level code

DAL Process

```
int fire(DALProcess *p)
{
    float i;
    DAL_read((void*)PORT_IN, &i, sizeof(float), p);

    i = i*i;

    DAL_write((void*)PORT_OUT, &i, sizeof(float), p);

    return 0;
}
```


Extended DAL Specification

```
int fire(DALProcess *p)
{
    float *rbuf = DAL_read_begin(PORT_IN, IN_RATE, p);

    DAL_foreach (blk : PORT_OUT)
    {
        float *wbuf = DAL_write_begin(PORT_OUT, OUT_RATE, blk, p);

        *wbuf = rbuf[blk] * rbuf[blk];

        DAL_write_end(PORT_OUT, wbuf, p);
    }

    DAL_read_end(PORT_IN, rbuf, p);

    return 0;
}
```


ETH zürich

Extended DAL Specification

```
int fire(DALProcess *p)
{
    float *rbuf = DAL_read_begin(PORT_IN, IN_RATE, p);

    DAL_foreach (blk : PORT_OUT)
    {
        float *wbuf = DAL_write_begin(PORT_OUT, OUT_RATE, blk, p);

        *wbuf = rbuf[blk] * rbuf[blk];

        DAL_write_end(PORT_OUT, wbuf, p);
    }

    DAL_read_end(PORT_IN, rbuf, p);

    return 0;
}
```

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory
Matthias Baer, ETH Zürich | 25.06.14 | 12

ETH zürich

Extended DAL Specification

```
int fire(DALProcess *p)
{
    float *rbuf = DAL_read_begin(PORT_IN, IN_RATE, p);

    DAL_foreach (blk : PORT_OUT)
    {
        float *wbuf = DAL_write_begin(PORT_OUT, OUT_RATE, blk, p);

        *wbuf = rbuf[blk] * rbuf[blk];

        DAL_write_end(PORT_OUT, wbuf, p);
    }

    DAL_read_end(PORT_IN, rbuf, p);

    return 0;
}
```

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory
Matthias Baer, ETH Zürich | 25.06.14 | 12

ETH zürich

Extended DAL Specification

```
int fire(DALProcess *p)
{
    float *rbuf = DAL_read_begin(PORT_IN, IN_RATE, p);

    DAL_foreach (blk : PORT_OUT)
    {
        float *wbuf = DAL_write_begin(PORT_OUT, OUT_RATE, blk, p);

        *wbuf = rbuf[blk] * rbuf[blk];

        DAL_write_end(PORT_OUT, wbuf, p);
    }

    DAL_read_end(PORT_IN, rbuf, p);

    return 0;
}
```

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory
Matthias Baer, ETH Zürich | 25.06.14 | 12

Transformation into POSIX

```
int fire(DALProcess *p)
{
    float *rbuf = DAL_read_begin(PORT_IN, IN_RATE, p);

    DAL_foreach (blk : PORT_OUT)
    {
        float *wbuf = DAL_write_begin(PORT_OUT, OUT_RATE, blk, p);

        *wbuf = rbuf[blk] * rbuf[blk];

        DAL_write_end(PORT_OUT, wbuf, p);
    }

    DAL_read_end(PORT_IN, rbuf, p);

    return 0;
}
```


Transformation into POSIX

```
int fire(DALProcess *p)
{
    float rbuf[IN_RATE];
    DAL_read((void*)PORT_IN, &rbuf, sizeof(float), p);
    for (int blk = 0; blk < BLOCK_OUT_COUNT; blk++)
    {
        float wbuf[OUT_RATE];
        *wbuf = rbuf[blk] * rbuf[blk];
        DAL_write((void*)PORT_OUT, &wbuf, sizeof(float), p);
    }

    return 0;
}
```


Transformation into OpenCL

```
int fire(DALProcess *p)
{
    float *rbuf = DAL_read_begin(PORT_IN, IN_RATE, p);

    DAL_foreach (blk : PORT_OUT)
    {
        float *wbuf = DAL_write_begin(PORT_OUT, OUT_RATE, blk, p);

        *wbuf = rbuf[blk] * rbuf[blk];

        DAL_write_end(PORT_OUT, wbuf, p);
    }

    DAL_read_end(PORT_IN, rbuf, p);

    return 0;
}
```


Transformation into OpenCL

```
int fire(DALProcess *p)
{
    __global float *rbuf = in + gid*TOKEN_IN_RATE;
    for (int blk = lid; blk < BLOCK_OUT_COUNT; blk += lsz)
    {
        __global float *wbuf = out + gid*TOKEN_OUT_RATE
            + blk*BLOCK_OUT_SIZE;
        *wbuf = rbuf[blk] * rbuf[blk];
    }

    return 0;
}
```

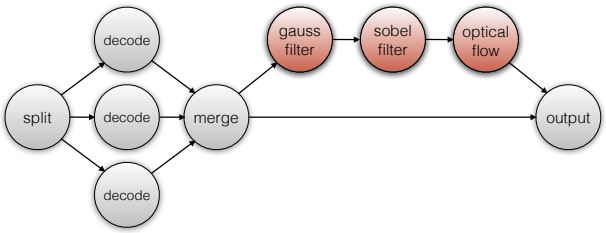

Evaluation

- Performance comparison
  - ➡ Handwritten OpenCL code **vs.** generated OpenCL code
  - ➡ Executing an application with different mappings
- Adaptability
  - ➡ Simplicity of converting an existing DAL application

Evaluation - Setup

CPU	Intel Core i7-2600K (4 Cores, 3.40 GHz, 8 MB Cache)
GPU	AMD Radeon HD 7750 (8 Compute Units each 256 Processing Elements)
Memory	16 GB RAM
Operating System	Arch Linux 3.14.6-1-ARCH x86_64

Evaluation - Application

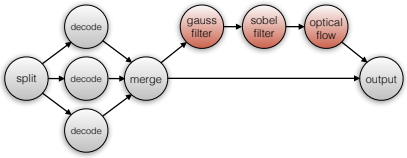


Evaluation - Comparison

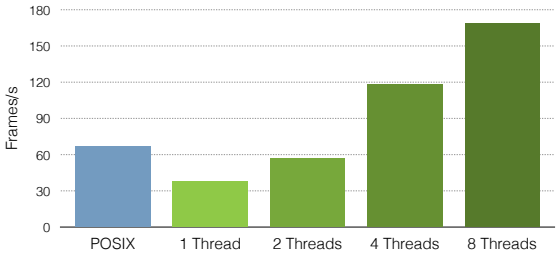
	Original OpenCL Implementation	New Syntax
GPU	2.159 s	2.149 s
CPU	24.62 s	24.64 s

Evaluation - Experiments

- All processes mapped to the CPU
  - ➡ The filters can be executed either as POSIX threads or using the OpenCL framework

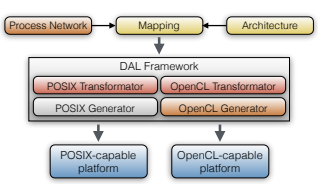


Evaluation - Experiments



Conclusion

- Integration of OpenCL generator into DAL
- Adaption of the specification
- Added a transformator to create POSIX or OpenCL code from one code base
- Comparison shows no significant performance loss



Bibliography

[1] Tobias Scherer, "Executing Process Networks on Heterogeneous Platforms using OpenCL," Master's thesis, ETH Zurich, May 2013.

[2] Lars Schor et al., "Scenario-Based Design Flow for Mapping Streaming Applications onto On-Chip Many-Core Systems," in *Proc. CASES'12*, 2012.

[3] Gilles Kahn, "The Semantics of a Simple Language for Parallel Programming," *Information Processing*, 1974.

[4] NVIDIA Corporation, "About CUDA," 2014. [Online]. Available: https://developer.nvidia.com/about-cuda

[5] Khronos Group, "OpenCL - The open standard for parallel programming of heterogeneous systems," 2014. [Online]. Available: http://www.khronos.org/opencl

[6] Aaftab Munshi, *The OpenCL Specification, Version 1.2*, 19th ed., Khronos OpenCL Working Group.

[7] Lars Schor, Andreas Tretter, Tobias Scherer and Lothar Thiele, "Exploiting the Parallelism of Heterogeneous Systems using Dataflow Graphs on Top of OpenCL," in *Proc. ESTIMedia*, 2013.

[8] Wolfgang Haid, Lars Schor, Kai Huang, Iuliana Bacivarov and Lothar Thiele, "Efficient Execution of Kahn Process Networks on Multi-Processor Systems Using Protothreads and Windowed FIFOs," in *Proc. ESTIMedia*, 2009.

[9] Advanced Micro Devices, Inc., "AMD Radeon™ HD 7700 Series Graphics Cards," 2014. [Online]. Available: http://www.amd.com/en-us/products/graphics/desktop/7000/7700

[10] Intel Corporation, "Intel® Core™ i7-2600K Processor," 2014. [Online]. Available: http://ark.intel.com/products/52214

[11] Harshavardhan Pandit, "Ultrasound 'To Go'," Bachelor's thesis, ETH Zurich, 2014.