ALMARVI

"Algorithms, Design Methods, and Many-Core Execution Platform for Low-Power Massive Data-Rate Video and Image Processing"

Project co-funded by the ARTEMIS Joint Undertaking under the
ASP 5: Computing Platforms for Embedded Systems
ARTEMIS JU Grant Agreement no. 621439

D3.3 Abstracting heterogeneous hardware architectures

Due date of deliverable: September 30, 2015

Start date of project: 1 April, 2014	**Duration:** 36 months
Organisation name of lead contractor for this deliverable:	Vector Fabrics BV
Author(s):	Jos van Eijndhoven, Toygar Akgün, Pekka Jääskeläinen, Zdenek Pohl
Validated by:	Pekka Toivanen
Version number:	1.2
Submission Date:	Oct. 10, 2015
Doc reference:	…
Work Pack./ Task:	WP 3, Task 3.3
Description: ***(max 5 lines)***	The report provides the description on how to interface and integrate application-level concurrency on multi-core processors and heterogeneous acceleration fabrics: a software layer abstracts from details in the target hardware architectures, enabling re-use through design methods, tools, and libraries.
Nature:	**R=Report**
Dissemination Level:	**CO** Confidential, only for members of the consortium (including the JU)

DOCUMENT HISTORY

Release	Date	Reason of change	Status	Distribution
V0.1	06/08/2015	Initial document organization	draft	CO
	24/08/2015	Chapter 3.4 UTIA (Jiri Kadlec and Zdek Pohk)	draft	CO
V0.2	17/08/2015	Updates from TUT	draft	CO
V0.3	29/09/2015	Updates from discussions in the project meeting	draft	CO
V1.0	1/10/2015	Draft for review	draft	CO
V1.1	5/10/2015	Updates from review	draft	CO
V1.2	6/10/2015	Updates from review	final	CO

Contents

Glossary

Abbreviation / acronym	Description
APU	Accelerated Processing Unit: Name used by AMD for their GPU subsystem, as part of their integrated CPU-GPU on-chip system architecture.
GPGPU	General-purpose computing on graphics processing unit
HSA	Heterogeneous System Architecture: a computer architecture that integrates heterogeneous processors, typically CPUs and GPUs, with shared-memory features.
POCL	Portable Computing Language: Open source implementation of the OpenCL standard.
SVM	Shared Virtual Memory: term used here to denote an on-chip architecture where the CPU and GPU's can cooperate by interacting on mutually shared memory, complete with virtual-memory addressing support.
TTA	Transport triggered architecture: a style of CPU micro-architecture and its instruction set, focusing on data transfers. Studied in Almarvi in WP3.
WG	Work Group: OpenCL term to denote a set of work items that will execute on the same compute unit and have access to a shared local memory

1 Introduction

The Almarvi project aims to develop an approach that allows for portable application software, across a range of modern high performance and energy efficient heterogeneous computing architectures. An important aspect of portable applications are the APIs and libraries used to express the application functionality. This report describes the selected APIs for the Almarvi project, which allow to interface and integrate heterogeneous acceleration fabrics: A software layer abstracts from details in the heterogeneous accelerators, enabling re-use through design methods and tools.

This report represents deliverable D3.3 which is part of Task 3.3 in WP3. Its content is aligned in particular with deliverable D3.1, which describes the newly developed hardware platforms, and deliverable D1.1 on use-case requirements. The partners involved in Task 3.3, the main contributors to this deliverable, are PHILIPS, TUDelft, VF, and TUT.

As discussed in WP4, the Almarvi software stack is mainly based on OpenCL. Choosing OpenCL as baseline set of APIs greatly helps in creating portable applications, both towards our novel Almarvi target platforms, as well as other (classic and novel) off-the-shelf 3rd party platforms. That is because:

- OpenCL is growing in popularity for programming heterogeneous devices, in particular in the embedded world.
- The OpenCL programming model is -in general- a good match with the application domains that Almarvi targets, on high-performance imaging and video analysis and mobile multi-media.

These advantages should outweigh the disadvantage of the rather verbose and low-level programming style. The OpenCL approach is further described in the next chapter 2. Although the project TUT partner is actively involved in an open-source and portable ‘pocl’ implementation, fully creating an OpenCL implementation for all Almarvi targets is beyond the capability (available manpower) for this project. Therefore the project chooses appropriate alternative concurrent programming model for other situations, also based on industry standards. These are described in chapter 3.

2 Almarvi baseline: OpenCL programming

As discussed in WP4, the Almarvi software stack is mainly based on OpenCL. Choosing OpenCL as baseline set of APIs greatly helps in creating portable applications, both towards our novel Almarvi target platforms, as well as other (classic and novel) off-the-shelf platforms. Although the project TUT partner is actively involved in the open-source and portable 'pocl' implementation, fully creating an OpenCL implementation for all Almarvi targets is beyond the capability (available effort) for this project. Therefore, we actively pursue the use of OpenCL in a few specific cases, and discuss good alternatives otherwise.

2.1 Classic GP-GPU style OpenCL

This matches the use-case where a relatively compact compute kernel is moved for its execution from the host to a (GPU-style) accelerator. The compute kernel is embedded in a loop which repeats the computations over a large index space. The **accelerator speedup** comes from distributing the index space across many concurrent processing elements. Since these processing elements can be rather minimal processors, and typically share instruction decode hardware among them, this scheme can also provide **energy savings**. Many good examples of this approach can be found on the internet. However, obviously, this approach is **not universally applicable**:

- The host and accelerator are heterogeneous environments with minimal coupling. The OpenCL (version 1.2) programming model requires explicit transfers of data from the host to the accelerator and vice versa, which is somewhat awkward to program and can cause severe performance degradation. Overlapping in time of data transfers, accelerator computations, and (other) host computations further add to the programming and performance analysis complexity. (For shared-memory operation see next section)
- Distributing the loop iterations over processing elements requires an almost perfect available parallelism between the loop instances: Absence of loop carried dependencies and highly uniform computations. This might be hard to achieve for some algorithms.

In Almarvi this approach is applied for some cases. Novel developments and results will be obtained from a) **creating algorithms** such that they are fit to this style of concurrency, and b) adopting **modern low-power** (embedded) GPU architectures.

Example Almarvi applications:

Mean-shift based segmentation, SLIC segmentation, optical flow estimation and thermal/day light camera image fusion applications are all accelerated on GPU using this style.

2.2 Modern OpenCL extensions

Pipes:

Energy-efficient computations generally come from application-specific processors that are tuned to do the required work with very little overhead. Applications must be partitioned between (at least) a generic 'host' part and the computation kernel for which the compute requirements are a good fit to the application-specific processor (ASIP). Often, and especially on the domain of the Almarvi applications, such application partitioning is made according to a data-flow architecture. In such an architecture, a process (or task) reads data items from an inbound queue, process the data, and pushes result data in an out-bound queue. Synchronization of the progress with surrounding tasks occurs implicitly through the queue accesses.

Such queues with built-in synchronization have been added in the 2.0 version of OpenCL as '**Pipes**'. This allows us to use OpenCL as API to create such systems, where the accelerators might not have abundant internal concurrency, but are just efficient in what they do. This mechanism is picked up, for example by Altera, providing an OpenCL interface to FPGA accelerator functions.

Unfortunately, OpenCL 2.0 does not allow to create such pipes for direct data communication between host code and its attached accelerator device, but (potentially) does allow to have such pipes between devices. This can be overcome by using a CPU device that runs in the host that feeds the other end of the pipe, if needed. Furthermore, unfortunately, the actual support for such pipes in OpenCL implementations today is not yet widespread. The Almarvi partners are encouraged to adopt these pipes where practically feasible. In addition, there is ongoing work to add a pipe implementation framework to *pocl* in order for the partners that utilize *pocl* as an OpenCL implementation framework to add support for pipe communication to their devices.

Shared virtual memory:

Shared virtual memory (SVM) is a technology where the host CPUs and the OpenCL target device can access mutually shared memory. This allows to pass data from the host to the device without copying, thus **gaining speed and energy**. Furthermore, this entails OpenCL device memory access through the host application virtual memory map, so that they can even share pointers, enabling shared access to complex data structures. The shared memory option is particularly attractive for embedded (integrated) devices, where the OpenCL device already physically shares the DRAM with the host CPU and can even have cache coherency across the caches between the CPUs and GPUs, thus making data sharing between the compute devices in the heterogeneous platform much more efficient. The Almarvi project might **take advantage** of this for some of its target architectures. In particular, the Intel HD 5XXX, 6XXX, various AMD 'APU's, and ARM MALI-T6XXX GPUs share the same memory space as their complementing CPUs and support zero copy operation.

Accelerators:

Next to the classical host CPUs and GP-GPUs that OpenCL addresses, the OpenCL 2 also provides generic Accelerator devices. Such devices are potentially a good match for the hardware accelerators that are created and used in a few project use cases utilizing Xilinx Zync devices. The description of OpenCL accelerator devices is rather scarce in the standard, which means that exploring this option will add to the generic knowledge on applying OpenCL 2 for a popular class of devices.

Example use in the Almarvi project:

Adopting the **pipe** abstraction of OpenCL 2.0 is being done by TUT for their TCE-based customized processor platforms. The idea is to provide efficient hardware-based streaming support for cases that do not need random access for the input/output. This will be combined with customized datapath to provide a template for highly energy efficient computation nodes.

SVM is one of the show off features for the Heterogeneous System Architecture (HSA). AMD's APU architecture is providing HSA support, which means it can provide coherent shared memory access for sharing data between the CPU and the GPU. This will be utilized via the SVM API of OpenCL 2.0 with the planned support for HSA devices in pocl. The goal is to demonstrate performance benefits via this abstraction on a commercial heterogeneous platform.

Other application developers in Almarvi can learn from these early examples.

3 Alternative concurrent processing APIs

In cases where the OpenCL APIs are not appropriate or not available for the target platform, alternative parallelization methods are to be used. The selected alternatives are well-recognized industry-standard solutions to maintain a good degree of portability of the application software across target platforms.

3.1 OpenMP

OpenMP is the dominant parallelization method for multi-threaded shared-memory concurrency on multi-core host processors (Intel/AMD X86, ARM Cortex A-series, IBM/Freescale PowerPC). OpenMP provides efficient concurrency with relatively little programmer effort (application rewrites). It achieves target independency through features from a standard runtime system such as deciding at runtime the number of threads to spawn. For this purpose (shared-memory multi-core host CPUs) the **OpenMP versions 3.0 or 3.1** are fine. The later OpenMP version 4.0 adds new features for heterogeneous targets and distributed memory. Actual support for those features on specific target platforms is not yet wide-spread, and we do **not adopt the 4.0 extensions for use in Almarvi**, in favor of OpenCL.

OpenMP is used to accelerate CPU implementation of mean-shift filter in Aselsan.

3.2 MCAPI

For newly-developed heterogeneous and energy-efficient hardware architectures in Almarvi, the relatively extensive software stacks like OpenMP or OpenCL might not (yet) be available. In such cases, software support might need to start with smaller light-weight solutions. Later, full OpenCL might be built on top of these. The prevalent industry-standard API for light-weight heterogeneous distributed-memory systems is MCAPI, as maintained by the multi-core association (MCA). In particular, MCAPI provides light-weight channel constructs to communicate messages between heterogeneous components.

MCAPI is not (yet) used in Almarvi. It will be checked whether it can play a glue-layer role in the implementation of an OpenCL-2 accelerator device.

3.3 Vectorization

Vectorization is a mechanism for data-level concurrency which is available on all host CPU (and many GPU) processors. Vectorization is often not discussed in the context of application concurrency, as it is treated very differently from multi-core parallelism. However, as it is widely available and provides significant **performance enhancements and energy savings** from plain C/C++ on host CPUs. The Almarvi project does explore these features and proposes a default software approach. The main challenge of vectorization is exploiting its advantages while maintaining portability of application software.

- OpenCL does provide a software API for explicit vectorization. In this approach, the programmer manually specifies the vector types and expresses the application in terms of these vector operations. Some GPUs do rely on such vectorization to achieve fast and efficient processing. The same method (the same application source code) can be used to achieve vector processing on host CPUs, when the host CPUs are selected as target OpenCL device. In practice, the programmer chooses the vector length (as part of the vector type), and with that choice creates a dependency on the target architecture. Typically, today's commercial GPUs feature a short vector length of just 4 bytes, and there for these vectors provide speedup only in case of short data types (chars and shorts). Mostly for that reason, expressing explicit vectorization is **not popular** among OpenCL programmers. In contrast, host CPUs typically have hardware support for longer vectors: 16-byte vectors in ARM neon and 32-byte vectors in Intel avx. In general, (OpenCL-) compilers have difficulty in exploiting vectors of the target hardware if the (OpenCL-) source code does not have vectors or specifies too narrow vectors. As exception, the proprietary Intel compiler does quite a good job in automatically exploiting the vector capability of the Intel CPUs, even if the source code does not express that.
- The more *performance portable* alternative for utilizing vector instructions from OpenCL kernels is the implicit vectorization. In case of OpenCL this means mapping multiple work-items of a work-group to execute in vector lanes of the vector instruction. For example, an 8 wide SIMD instruction could then execute 8 work-items from the same work group (WG) in parallel. When implicit vectorization is desired, the OpenCL kernels are written in a style where the work-items usually handle a single scalar data item

which when combined by an implicit vectorizer then produce vector-based computation. As this involves automatic vectorization, it on the other hand helps "performance portability"; the same kernel can potentially utilize various vector widths more efficiently, but, on the other hand it is more fragile; it depends on the compiler's capabilities whether the vector instructions are utilized or not. The latest OpenCL kernels compilers from Intel and AMD can support implicit vectorization with a lot of kernels. In addition pocl has an implicit WG vectorization capability that works for many cases, but, like always is the case with compiler based vectorization there is room for improvement. In ALMARVI the *pocl's* implicit WG-vectorization is used to provide a degree of performance portability across the various width vector SIMD instructions available in TCE custom cores.

- Historically, vectorization for (X86-) CPUs was done by **assembly-level** programming. This allowed manual selection of the (most optimal) vector instructions, but also required manual managing of the vector register allocation. The manually created short fragments of assembly code were integrated in surrounding C code through 'asm' statements. This approach should **not be used anymore**, as it is time consuming and creates code that is strongly target dependent.
- Nowadays, programmer manual (explicit) vectorization for Intel and ARM CPUs is often done by calling **vector 'intrinsics'** embedded in the C/C++ code. These 'intrinsics' syntactically look like function calls, but have a very close relation with the underlying processor instructions, and are translated by the compiler to indeed just invoke the intended vector instructions without creating function-call overhead. This allows the programmer to select optimal vector instructions, but relieves the programmer from manual (vector-) register allocation. Also, because the compiler is responsible for creating the final assembly, the portability of the code is improved in comparison with manual assembly programming. For many situations, this is currently a **viable approach**.
- Recent versions of C/C++ compilers for popular host CPU's (X86, ARM) have really improved in auto-vectorization. So, the application source code expresses merely loops with scalar processing of array elements. With proper compiler optimization flags, the compilers can automatically create vector instructions. Next to choosing a high optimization level, this requires that the target architecture is specified to the compiler. Furthermore, critical inner loops might need some rewriting to make them eligible for auto-vectorization. In effect, **good results can be obtained** this way in many cases, while maintaining **optimal portability** of the (standard, scalar) C code. This approach should be **preferred for application kernels that execute on host CPUs**. While there has been a lot of work in the parallelization of sequential C loops in the past decades, it is a fact that a sequential programming language such as C loses information which might make a potential parallel loop serialized in the eyes of the compiler. The alias analysis and loop dependence analysis might fail to see the parallelism that is actually there, preventing efficient auto-vectorization of loops. This is one of the points where implicit parallelization of OpenCL work-groups brings benefits; the multiple work-item execution are actually parallel loops which can be more easily mapped to parallel hardware like SIMD instructions by the compiler because of the explicit parallelism information.

As mentioned above, auto-vectorization is used and explored today in the *pocl* compiler for OpenCL kernels. At the time of writing this, the Almarvi use-cases do not actively test compiler auto-vectorization for compute kernels in plain C that remain mapped on a host CPU (ARM or X86). It is probably valuable to test this, to create a comparison in performance and power with an (embedded) GPU. We will check for such an opportunity.

3.4 Functional partitioning in Xilinx ZYNQ SDSoC tool chain

Several partners in ALMARVI are working on mapping an application to Xilinx ZYNQ platform. Such mapping typically requires a restructuring of a reference algorithm implementation such that C/C++ functions become eligible units for mapping. That means that some of the encompassing code will remain on the (multi-)ARM host processors and that one or more C/C++ functions shall be converted into dedicated hardware accelerators, implemented in the on-chip FPGA logic.

Creating the communication infrastructure between the ARM(s) and the accelerator(s), and between accelerators, is a relatively complex task. There are usually multiple options available, with very different HW resource cost, different performance and different software-complexity and interfaces.

Creating data exchange between an application on Linux/ARM (operating with virtual address space) and a hardware accelerator through shared memory has been a significant challenge. The OpenCL-2.0 accelerator concept can help to establish a portable programming model helping to reduce this complexity if this complex data exchange HW/SW structures will be provided. Xilinx SDSoC tools help to design these hw/sw communication channels by providing

support to create the required hw/sw communication infrastructure. These Xilinx tools tend to create a streaming data infrastructure, which is conceptually different from the coarse-granularity memory I/O of OpenCL.

3.4.1 SDSoC project and Hardware Platform

The SDSoC tool is capable to take one or more C++ functions (running originally on ARM processor) and create a hardware accelerated equivalent, including the required hw/sw communication infrastructure.

Initially, the user is required to provide custom hardware platform as a starting point for the tool. It requires user to define basic configuration of ZYNQ Processing System (PS) and Programmable logic (PL). As a starting point for project can be also used one of platforms provided by Xilinx for ZC702 and Zed boards. The hardware platform typically contains following components:

- Description of PL design and PS configuration
- Exposed clock, reset and interrupts for implementation of SDSoC hardware blocks
- Optional libraries containing software drivers for IP cores present in PL
- Optional sample application for developers to start with
- Support for standalone, FreeRTOS and Linux targets (SDSoC uses it to generate boot files for SD card)
- Optional precompiled hardware bitstream – it speeds up software development cycles (debugging of applications written in C/C++ on arm cores before turning them to HW accelerators

The hardware architecture of such platform can be just ZYNQ PS core or it can be extended by additional I/O interfaces, dedicated IP cores or other IP core components.

3.4.2 Acceleration in PL fabric

SDSoC tool allows to setup C/C++ project and mark some functions to be accelerated by in FPGA fabric. Figure 1 shows the SDSoC GUI where the functions “ycrcb2y”, “sobel” and “y2gray” have been selected to be accelerated The SDSoC design environment compiles hardware blocks for each accelerated C/C++ function, connects them with each other by AXI stream busses and complements them with a “data mover”. The data mover reads input data from the DDR3 by dedicated DMA and connects them to the AXI streams chain. It also reads the results from the AXI stream and transfers data by DMA to the DDR3.

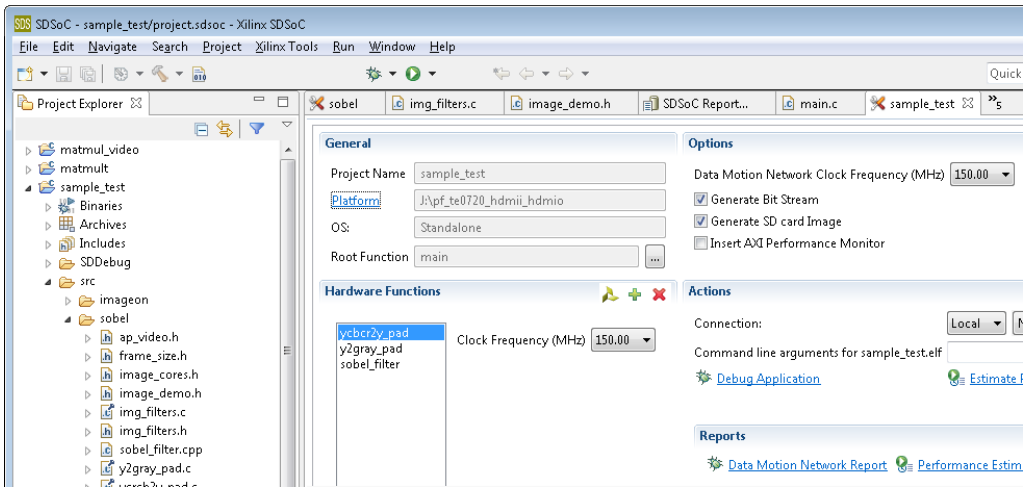

Figure 1: Functions to be HW accelerated, SDSoC GUI

The C/C++ function prepared to be accelerated must use a specific coding style and compiler directives (pragmas), as described in detail in the manual of the Vivado High Level Synthesis compiler (HLS).

4 Summary

OpenCL will be the key mechanism to allow portability of applications over a range of potential target architectures in the Almarvi project. Beyond the classic application partitioning between a host CPU and a GP-GPU extension card, Almarvi studies the use of recent OpenCL-2.0 extensions: shared-memory architectures, pipelining, and abstract accelerators, and utilization in embedded computer architectures. The project use-cases allow to explore these options and report on their merits for different application domains. This work is partially supported by developments in OpenCL itself, in the context of the *pocl* open-source implementation. Next to (and in combination with) OpenCL, application concurrency is deployed through OpenMP, which is a convenient and efficient parallelization method to exploit multi-core homogeneous host processors. As we focus on OpenCL, the project will not adopt recent OpenMP extensions from its version 4.0 that also target heterogeneous architectures. Finally, vectorization is a well-known mechanism to obtain throughput improvements in combination with power efficiency. It is deployed in a few applications, but most prominently visible in the wide-data-path transport triggered architecture (TTA), where the code portability issues are addressed by challenging compiler auto-vectorization.

For the programming of mixed hw/sw systems on the popular Xilinx Zync devices, the use of OpenCL-2 could be a valid option. However, the brand-new Xilinx SDSoC toolset has its own mechanism to create streaming communication infrastructures, and the project will explore this toolset for its merits.