www.vineyard-h2020.eu

VINEYARD

D4.1 Programming Language and Runtime System: Requirements

DOCUMENT ID	D4.1.
DUE DATE	01/08/2016
DELIVERY DATE	01/08/2016
CLASIFICATION	Confidential
AUTHOR/S	Hans Vandierendonck
DOCUMENT VERSION	0.6

CONTRACT START DATE	1st FEBRUARY 2016
CONTRACT DURATION	36 Months

PARTNERS
Institute of Communications and Computer Systems, Maxeler Technologies, Bull Systems, Queen's University of Belfast, Foundation for Research and Technology, The Hartree Centre / Science and Technologies Facilities Council, Neurasmus BV, Neurocom Luxembourg, Hellenic Exchanges SA, Holding, Clearing, Settlement and Registry, LeanXcale, Loba

European Commission
Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628

1 EXECUTIVE SUMMARY

The VINEYARD projects aims to achieve easy-to-use and transparent acceleration of data analytics. One of the components in the VINEYARD is the programming model and runtime system support, which is developed in Work Package 4. This document elaborates the requirements for the VINEYARD programming model and runtime system. We have summarized 12 requirements on the programming language, runtime system and acceleration library that are necessary to realize the goals and ambition of the VINEYARD project as it is shown in this table.

Component	Requirement
Programming model	Support acceleration of big data analytics frameworks (e.g., Spark, Storm and Heron) with accelerators including at least FPGAs and if possible also GPUs and Xeon Phi accelerators.
	Support concise description of equivalent implementations of the same algorithm and a uniform interface for invoking these implementations.
	If necessary, support annotation of equivalent implementations with additional information in order to enable the VINEYARD runtime system to autonomously select one version over another, or to efficiently load-balance work across accelerators.
	Balance programmer control versus transparency through the design of the programming model.
Runtime System	Develop techniques to efficiently share data between managed language runtimes and low-level (bare-metal) programming environments typically used on accelerators.
	Develop scheduling techniques for variable-rate data streams to optimize throughput, resource utilization and/or energy efficiency building on the concept of fair-share allocation.
	Develop scheduling techniques for hybrid scheduling across CPUs and accelerators.
	Design a memory management subsystem to manage data distribution across CPU nodes and accelerators.
	Optimize workload schedulers by taking into account existing data allocation and minimizing data movement.
	Design scheduling strategies and runtime system support for virtualized accelerators.
Acceleration library	Define library of reusable accelerator IP blocks
	Optimize configuration of the IP blocks given available hardware resources

PARTNERS
Institute of Communications and Computer Systems, Maxeler Technologies, Bull Systems, Queen's University of Belfast, Foundation for Research and Technology, The Hartree Centre / Science and Technologies Facilities Council, Neurasmus BV, Neurocom Luxembourg, Hellenic Exchanges SA, Holding, Clearing, Settlement and Registry, LeanXcale, Loba

European Commission | Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628

CONTRIBUTORS

Hans Vandierendonck	Queen’s University Belfast
Christoforos Kachris	ICCS
George Chatzikonstantis	ICCS
Dimitrios S. Nikolopoulos	Queen’s University Belfast
Tobias Becker	MAX

PEER REVIEWERS

Name	Organization
Mike Ashworth	STFC
Chrisotofors Kachris	ICCS

REVISION HISTORY

Version	Date	Author/Organisation	Modifications
0.1	22.06.2016	Hans Vandierendonck, QUB	Initial Version
0.2	3.7.2016	Christoforos Kachris, ICCS George Chatzikonstantis	FPGA programming model, Acceleration requirements Xeon Phi programming model
0.3	19.07.2016	Hans Vandierendonck and Dimitrios S. Nikolopoulos, QUB	Programming model and runtime system requirements
0.4	25.07.2016	Tobias Becker	Maxeler programming model
0.5	25.07.2016	Christoforos Kachris, ICCS	State-of-the-art of FPGA usage in data analytics
0.6	30.07.2016	Hans Vandierendonck, QUB	Internal review

European Commission
Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628

Table of Contents

1 EXECUTIVE SUMMARY 2

2 Introduction 8

2.1 Goal of Deliverable 8

2.2 Audience 8

2.3 Document Structure 8

3 Background on Accelerator Programming Models 9

3.1 Programming Models for FPGAs 9

3.1.1 Hardware Description Languages 9

3.1.2 High level Synthesis (HLS) 10

3.2 The Maxeler DFE Programming Model 14

3.2.1 Dataflow Engine Architecture 14

3.2.2 DFE Platform Programming 17

3.3 Programming Models for GPUs 22

3.3.1 C Language Interfaces 22

3.3.2 Java-C Bindings 24

3.3.3 Java-Language Integration 24

3.3.4 Performance Considerations 27

3.4 Programming Models for the Xeon Phi 28

3.4.1 MIC Hardware Architecture 28

3.4.2 Modes of Operation 30

3.4.3 Programming Frameworks....31

3.4.4 Performance Monitoring....32

4 Background on Data Analytics Platforms....32

4.1 Batch Processing....32

4.1.1 Acceleration of Batch Processing....33

4.2 Stream Processing....36

4.2.1 Acceleration of Stream Processing....37

5 Requirements....37

5.1 Programming Model....38

5.1.1 Programming Model Support for Accelerators....38

5.1.2 Runtime System and Scheduling....39

5.1.3 Data Distribution....40

5.1.4 Virtualization....41

5.2 Acceleration Libraries....41

5.3 Summary....43

6 Conclusion....44

Table of Figures

Figure 1. Current FPGAs can integrate several specialized components such as Application Processors, Real-time Processors, Graphics Processors, High speed transceivers, power management, DSP and memory blocks, Source: Xilinx, 2016....9

Figure 2. The Xilinx's SDAccel framework for the programming of FPGA based on OpenCL.....13

Figure 3: A conventional control-flow oriented processor (a) compared to a data-flow engine (b). 15

Figure 4: Structure of a Maxeler MAX4 data-flow engine. 15

Figure 5: A Maxeler MPC-X system with eight DFEs in a single node. Infiniband network is used to connect MPC-X with the CPUs. All DFEs can be allocated dynamically. 16

Figure 6: Parallel execution in data-flow system. 18

Figure 7: A simple data-flow graph (left). All arithmetic operations are carried out in parallel. A practical data-flow application (right) with 5000 arithmetic operators running concurrently in a data-flow pipeline. 20

Figure 8: Example of a DFE kernel performing a moving average computation. 21

Figure 9. Architecture of an Intel® Xeon Phi™ core. 29

Figure 10. Different execution models on a typical node with Xeon CPU and Xeon Phi accelerator. 30

Figure 11. High-level overview of the VINEYARD framework and the Acceleration library 42

List of Tables

Table 1 Summary of requirements of the VINEYARD programming model and runtime system 43

PARTNERS
Institute of Communications and Computer Systems, Maxeler Technologies, Bull Systems, Queen's University of Belfast, Foundation for Research and Technology, The Hartree Centre / Science and Technologies Facilities Council, Neurasmus BV, Neurocom Luxembourg, Hellenic Exchanges SA, Holding, Clearing, Settlement and Registry, LeanXcale, Loba

European Commission | Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628

2 Introduction

The aim of VINEYARD is to make accelerators easy and transparent to use such that infrastructure efficiency is improved and application-level Quality of Service (QoS) is enhanced. In order to help achieve this goal, Work Package 4 of the VINEYARD project aims to define a programming model and its accompanying runtime system that achieves the outlined goals. This programming model and runtime system will build on existing, leading big data analytics platforms and extend their capabilities with seamless and transparent acceleration.

This deliverable is a summary of our initial study of the requirements of such extensions to data analytics platforms. It identifies key challenges that need to be addressed by WP4.

2.1 Goal of Deliverable

The aim of Task 4.1 is to define, design and implement the VINEYARD programming model and integrate programmable accelerators into the programming model using a library interface. The starting point of this task will be the Spark and Storm programming models for processing stationary and streaming data, respectively. Deliverable D4.1 is the first checkpoint of this development.

2.2 Audience

VINEYARD partners involved with developing and evaluating the VINEYARD programming framework (WP4) and the runtimes (WP5). Also the application partners involved with applying the VINEYARD programming framework to the use cases on neuro-computing, financial applications and data management applications.

2.3 Document Structure

This document first reviews the state-of-the-art in programming models for accelerators. It is necessary to integrate accelerator programming models for the supported accelerators in the VINEYARD programming model. Next, the document reviews data analytics platforms and prior work on applying acceleration to these

European Commission | Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628

platforms. Finally, we deduce the requirements for the VINEYARD programming platform, which aims to address open issues.

3 Background on Accelerator Programming Models

3.1 Programming Models for FPGAs

Field Programmable Gate Arrays (FPGAs) are integrated circuits designed to be configured/programmed after manufacturing. FPGAs are usually based on Look-Up Tables that can be programmed to execute logical and arithmetic operations.

Although FPGAs were initially used as glue logic for digital design circuits, currently FPGAs are emerging as fully SoCs (system on chip) with many integrated devices such as memories, Digital Signal Processing Units (DSP), memory blocks (BRAM), and high-speed transceivers that can reach up to 28 Gbps.

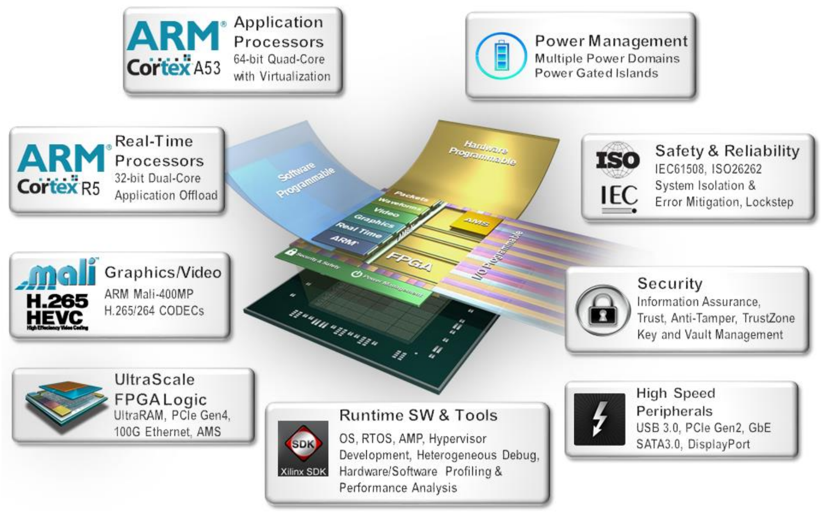


Figure 1. Current FPGAs can integrate several specialized components such as Application Processors, Real-time Processors, Graphics Processors, High speed transceivers, power management, DSP and memory blocks, Source: Xilinx, 2016

3.1.1 <u>Hardware Description Languages</u>

Despite the recent push toward high level synthesis (HLS), hardware description languages (HDLs) remain the most widely used programming model in field programmable gate array (FPGA) development. Specifically, two FPGA design languages have been used by most developers: VHDL and Verilog. Both of these "standard" HDLs emerged in the 1980s, initially intended only to describe and simulate the behavior of the circuit, not implement it. Most designs have been developed using one or the other of these languages.

HDLs allow the designer to describe in full detail the logic circuit that will be implemented on the FPGA. The logic circuits that are described in HDL are then mapper to the logic resources of the FPGAs (for example a digital logic function can be mapped to a Look-Up Table in the FPGA that will operate as a digital logic function).

3.1.2 High level Synthesis (HLS)

Although the most of the FPGAs are currently programmed using HDL, there are also other ways to program the FPGAs[1].

The C, C++ or System C option allows us to leverage the capabilities of the largest devices. The ability to use C-based languages for FPGA design is brought about by HLS (high level synthesis), which has been on the verge of a breakthrough now for many years with tools like Handle-C and so on. Recently it has become a reality with both major vendors, with Altera and Xilinx offering HLS within their toolsets Spectra-Q and Vivado HLx respectively.

However, HLS has limitations when using C-based approaches, just like with traditional HDL you have to work with a subset of the language. For instance, it is difficult to synthesize and implement system calls, and users have to make sure everything is bounded and of a fixed size.

Furthermore, dynamic memory allocation is not supported in HLS (malloc, free, etc.). Therefore, any legacy code that is written using dynamic memory management has to be modified accordingly.

[1] Adam Taylor, 10 Ways To Program Your FPGA, EETimes, online article, http://www.eetimes.com/document.asp?doc_id=1329857

PARTNERS
Institute of Communications and Computer Systems, Maxeler Technologies, Bull Systems, Queen's University of Belfast, Foundation for Research and Technology, The Hartree Centre / Science and Technologies Facilities Council, Neurasmus BV, Neurocom Luxembourg, Hellenic Exchanges SA, Holding, Clearing, Settlement and Registry, LeanXcale, Loba

European Commission | Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628

In general, C-based languages are not well suited for HLS. The major challenges are the lack of: 1) timing information in the code, 2) size-based data types (or variable bit length data types), 3) built-in concurrency model(s), 4) local memories separated from the abstraction of one large shared memory. While all these points are valid, the main attraction of C-based languages is familiarity. Most HLS tools using C-based languages provide workarounds for one or more of these obstacles[2].

One of the main benefits of HLS, however, is the ability to develop the algorithms in floating point and let the HLS tool address the floating- to fixed-point conversion.

A number of other C-based implementations are available, such as OpenCL which is designed for software engineers who want to achieve performance boosts by using a FPGA without a deep understanding of FPGA design. Open Computing Language (OpenCL) is a programming language originally proposed by Apple Inc. and maintained by the Khronos Group[3]. The OpenCL specification provides a framework for programming parallel applications on a wide variety of platforms including CPUs, GPUs, DSPs, and FPGAs[4]. Moreover, OpenCL is a royalty-free, cross-platform, cross-vendor standard that targets supercomputers, embedded systems, and mobile devices. OpenCL allows programmers to use a single programming language to target a combination of different parallel computing platforms. Parallel computation is achieved through both task-level and data-level parallelism. The OpenCL framework provides an extension of C (based on C99) with parallel computing capabilities and the OpenCL API, which is an open standard for different devices. In the OpenCL programming model, a host is connected to one or more accelerator devices running OpenCL kernels. Device vendors provide OpenCL compilers and runtime libraries necessary to run the kernels. The host program is written in standard C in order to query, select, and initialize compute devices. Communication between the host program and accelerators is established through a set of abstract OpenCL library routines. Each

[2] S. A. Edwards, "The challenges of synthesizing hardware from C-like languages," IEEE Design Test Comput., vol. 23, no. 5, pp. 375–386, 2006.
[3] Khronos. [Online]. Available: https://www.khronos.org/opencl
[4] N. Trevett, "OpenCL introduction," in SIGGRAPH Asia, 2013, https://www.khronos.org/assets/uploads/developers/library/2013-siggraph-asia/OpenCL%20Intro%20SIGGRAPH%20Asia%20Nov13.pdf

accelerator device is a collection of compute units with one or more processing elements. Each processing element executes code as SIMD or SPMD.

In the FPGA industry, both Altera and Xilinx have announced support for OpenCL HLS in their FPGA development tools. Altera released an OpenCL SDK in 2013 that supports a subset of the OpenCL 1.0 specifications. Xilinx introduced support for OpenCL in their Vivado HLS tool in April 2014.

Commercial frameworks

Xilinx Vivado HLS

Vivado High-Level Synthesis is a complete HLS environment from Xilinx. It has been in development for the last several years following Xilinx's acquisition of AutoESL. Vivado HLS is available as a component of Xilinx's larger Vivado Design Suite or as a standalone tool. Like most HLS tools, Vivado HLS is mostly oriented towards core generation over full system design. It is possible to create hybrid designs with portions of code running on a soft-core processor communicating with custom hardware accelerators. Depending on requirements, the hardware accelerator can be exported as one of several different Xilinx specific core formats for simple integration into other products, or just the HDL specification.

The Vivado HLS tool is built using LLVM compiler framework[5]. As such it has access to many software optimizations (e.g., loop-unrolling, loop-rotation, deadcode elimination, etc.). However, hardware and software programing paradigms are inherently different so we cannot expect all of LLVM's optimizations to work seamlessly for HLS. Several studies using Vivado HLS to generate FPGA accelerators have been demonstrated, including Dynamic Data Structures[6], and real-time embedded system vision[7].

Xilinx also offers an integrated framework for the deployment of FPGAs in data centers. SDAccel's architecturally optimizing compiler allows software developers to compile and

[5] LLVM. [Online]. Available: http://llvm.org/

[6] F. Winterstein, S. Bayliss, and G. A. Constantinides, "High-level synthesis of dynamic data structures: A case study using Vivado HLS," in Int. Conf. FPT, Dec. 2013, pp. 362–365.

[7] J. Hiraiwa and H. Amano, "An FPGA implementation of reconfigurable real-time vision architecture," in Adv. Inf. Netw. Appl. Workshops, Mar. 2013, pp. 150–155.

PARTNERS
Institute of Communications and Computer Systems, Maxeler Technologies, Bull Systems, Queen's University of Belfast, Foundation for Research and Technology, The Hartree Centre / Science and Technologies Facilities Council, Neurasmus BV, Neurocom Luxembourg, Hellenic Exchanges SA, Holding, Clearing, Settlement and Registry, LeanXcale, Loba

European Commission | Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628

optimize streaming, low-latency, and custom datapath applications. The SDAccel compiler targets high-performance Xilinx FPGAs and supports source code using any combination of OpenCL, C, C++, and kernels. According to Xilinx, the SDAccel compiler delivers as much as a 10X performance improvement over high-end CPUs with one tenth the power consumption of a GPU, while maintaining code compatibility and a traditional software programming model for easy application migration and cost savings.

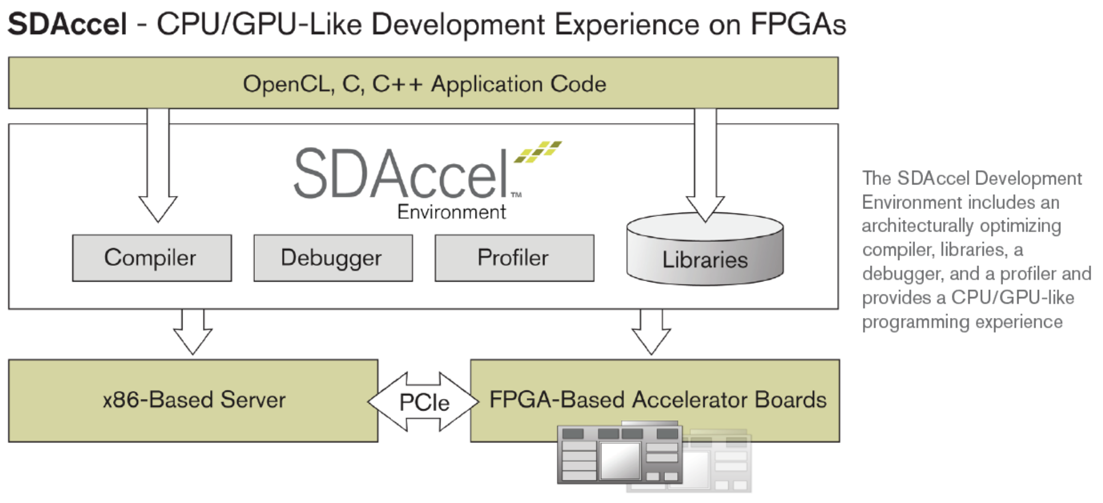

SDAccel - CPU/GPU-Like Development Experience on FPGAs
OpenCL, C, C++ Application Code
SDAccel™ Environment
Compiler
Debugger
Profiler
Libraries
x86-Based Server
PCIe
FPGA-Based Accelerator Boards
The SDAccel Development Environment includes an architecturally optimizing compiler, libraries, a debugger, and a profiler and provides a CPU/GPU-like programming experience

Figure 2. The Xilinx's SDAccel framework for the programming of FPGA based on OpenCL.

Altera

The Altera OpenCL SDK provides software programmers an environment based on a multi-core programming model that abstracts away the underlying hardware details while maintaining efficient use of FPGA resources. The Altera Offline Compiler (AOC) is an offline compiler that translates OpenCL to Verilog and runtime libraries for the host application API and hardware abstractions. Unlike the OpenCL compiler for CPUs and GPUs, where parallel threads are executed on different cores, AOC transforms kernel functions into deeply pipelined hardware circuits to achieve parallelism. AOC uses a CLANG front-end to parse OpenCL extensions and intrinsics to produce unoptimized LLVM IR (intermediate code). The middle-end performs optimization with about 150 compiler passes such as loop fusion, auto vectorization, and branch elimination. On the back-end, the compiler instantiates Verilog IP and manages control flow circuitry of loops, memory stalls, and branching. Finally, the generated kernel is loaded onto an Altera FPGA using an OpenCL compatible hardware image.

3.2 The Maxeler DFE Programming Model

3.2.1 Dataflow Engine Architecture

Maxeler commercialises a dataflow-oriented computing approach that fundamentally differs from conventional CPUs which are instruction and control-flow oriented. A CPU works by reading and decoding instructions, loading data, carrying out an operation on the data, and writing the result back to memory. This process is fundamentally sequential and requires complex control units to manage the operation of the processor. In comparison, the execution model of a Data-Flow Engine (DFE) is greatly simplified[8]. Data flows from memory into the chip where arithmetic operations are carried out by chains of functional units (data-flow cores) which are statically interconnected in a topology corresponding to the implemented functionality. This is illustrated in Figure 3. Data simply streams from one functional unit directly to the next one without the need for instructions; it arrives just in time when it is needed and the final results flow back into memory. Every single data-flow core performs only a simple arithmetic operation such as multiplication or addition. Therefore, thousands of arithmetic units can be put onto a chip and all of them can potentially perform useful calculations all of the time.

The dataflow pipeline is an application-specific compute structure which requires a reconfigurable chip substrate to create and customise the pipeline for a specific application. Maxeler realises Data-flow Engines by combining a large reconfigurable device with large amounts of DDR memory organised in multiple parallel channels. The structure of the current generation MAX4 DFE architecture is illustrated in Figure 4. It uses an Altera Stratix-V FPGA to provide the reconfigurable substrate for the data-flow computations.

[8] Tobias Becker, Oskar Mencer, Stephen Weston, Georgi Gaydadjiev. "Maxeler Data-Flow in Computational Finance" FPGA Based Accelerators for Financial Applications, pp 243-266, Springer, 2015.

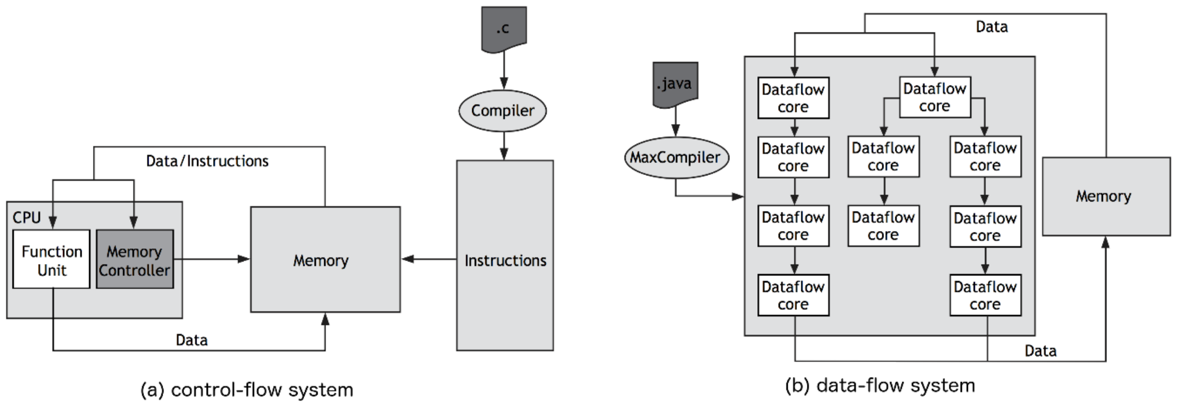


Figure 3: A conventional control-flow oriented processor (a) compared to a data-flow engine (b).

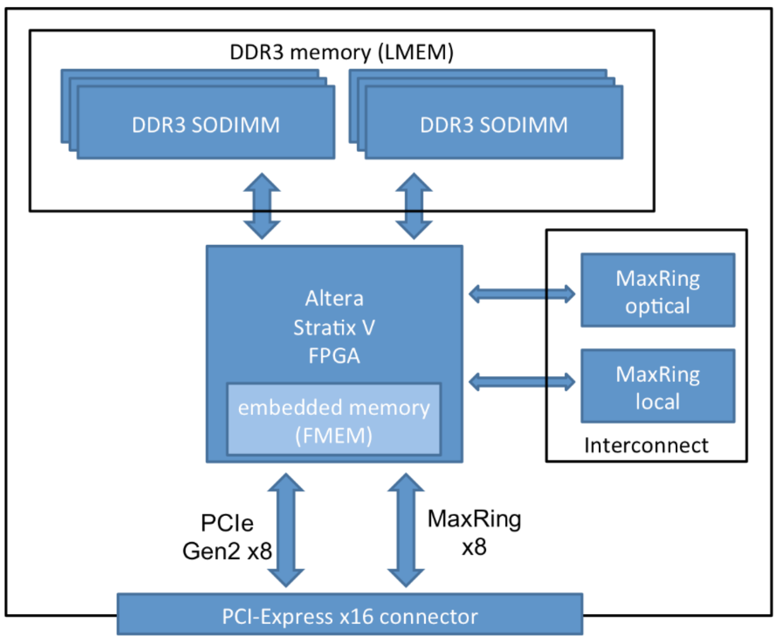


Figure 4: Structure of a Maxeler MAX4 data-flow engine.

Altera Stratix-V FPGAs contain programmable logic resources in form of general-purpose logic look-up tables, programmable interconnect, on-chip memory and programmable DSPs. These programmable resources are used to create the application-specific dataflow pipeline. Altera FPGAs also contain embedded memory blocks which are used in a DFE as so-called Fast Memory (FMEM). FMEM blocks are spread throughout the reconfigurable substrate and can be accessed at a total data-rate of several terabytes per second. This is useful for local low-latency buffering of data. The FPGA is surrounded by large amounts of DRAM. This memory is called Large

Memory (LMEM). LMEM is used for bulk storage and streaming of data. A MAX4 card uses an 8-lane PCIe interface to the CPU which provides a total bandwidth of up to 4GB/s. The card also provides several MaxRing connectors which create high-speed links directly between multiple DFE cards. Various electrical and optical MaxRing connector options are available. The next generation DFE to be developed in Vineyard will use a newer generation FPGA device but the overall concept of the DFE architecture will be maintained.

Maxeler's high-performance dataflow computing systems consist of multiple DFEs, CPUs, networking, and storage. Several system architectures are available and the overall component balance can be customised at system level to the requirements of the user. For example, Maxeler's MPC-X series systems are pure dataflow appliances that integrate eight MAX4 DFE cards into a dense 1U industry-standard chassis. This is illustrated in Figure 5.

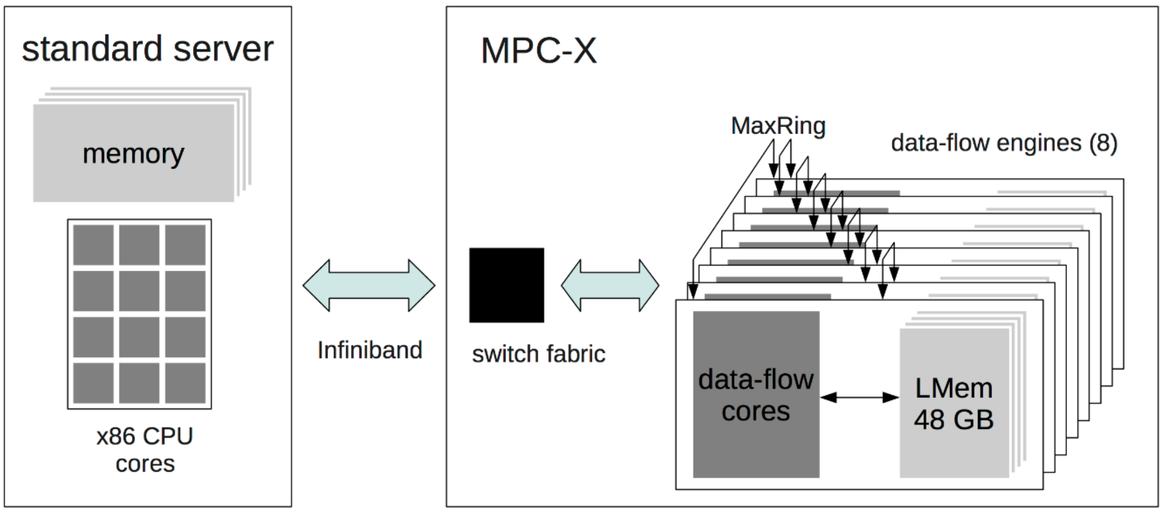


Figure 5: A Maxeler MPC-X system with eight DFEs in a single node. Infiniband network is used to connect MPC-X with the CPUs. All DFEs can be allocated dynamically.

The MPC-X system contains only DFE cards and no CPUs. Each DFE card contains 48 GB of DRAM as LMEM and DFEs are directly connected through MaxRing in a bidirectional 1D array topology. The MPC-X system is connected to industry standard CPU servers via an Infiniband network. The CPU server acts as an application host and compute intensive tasks are offloaded to DFEs. This architecture allows a flexible number of CPU servers and MPC-X nodes to be connected via an Infiniband network,

European Commission | Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628

and a various number of DFEs can be allocated dynamically to several host applications. Such scalability and flexibility is useful for applications with changing run-time behaviour, e.g., a computation that has several stages, which differ in their behaviour or complexity.

3.2.2 DFE Platform Programming

Programming a dataflow system requires the application to be described in a dataflow model. This involves splitting the application into its data plane and control plane. The data plane will be mapped onto the DFE and it will be highly efficient for carrying out large-scale computations with a static execution model. However, DFEs are not very efficient for computing small-scale problems with control-dominated dynamic behaviour. This part will be handled by a conventional CPU which acts as a host that sets up and controls the computation on the DFE and also and carries out the control-intensive tasks. The dataflow part of the application will be described in MaxJ, a Java-based meta-language while the control part is developed in C, C++ or other conventional programming approaches. Maxeler provides a programming environment and run-time system which comprises of several components:

- MaxCompiler, a programming environment to develop data-flow applications. The compute-intensive DFE parts are described in the MaxJ programming language. The compute kernels handling the data-intensive part of the application and the associated manager, which orchestrates data movement within the DFE, are written using this language. The CPU part of the application can be written in C, C++, etc;
- The SLiC (Simple Live CPU) interface, which is Maxeler's application programming interface for seamless CPU-DFE integration;
- MaxelerOS, a software layer and run time between the SLiC interface, the Linux operating system and the hardware, which manages DFE hardware and CPU-DFE interactions in a way transparent to the user;
- MaxIDE, a specialised Eclipse-based integrated development environment for MaxJ and DFE design, a fast DFE software simulator and comprehensive debug provisions used during development.

PARTNERS
Institute of Communications and Computer Systems, Maxeler Technologies, Bull Systems, Queen's University of Belfast, Foundation for Research and Technology, The Hartree Centre / Science and Technologies Facilities Council, Neurasmus BV, Neurocom Luxembourg, Hellenic Exchanges SA, Holding, Clearing, Settlement and Registry, LeanXcale, Loba

European Commission | Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628

To illustrate DFE programming in MaxJ we first consider the parallel execution model inside a DFE. All operations within a DFE are naturally parallel, and any operation specified in MaxJ code is parallel unless explicitly specified as sequential. The general model of a DFE is illustrated in Figure 6. Data streams from memory through a pipeline of data-flow cores with final results being streamed back to memory. Each data-flow core receives a continuous stream of data from either memory or from a previous data-flow core and the output data stream directly feeds into another data-flow core or back into memory. All data-flow cores operate concurrently and they are statically interconnected at design time. Hence, there is no control flow, synchronisation or routing necessary between data-flow cores. A CPU system is used to set up the computation on the DFE and to perform all the control-intensive tasks.

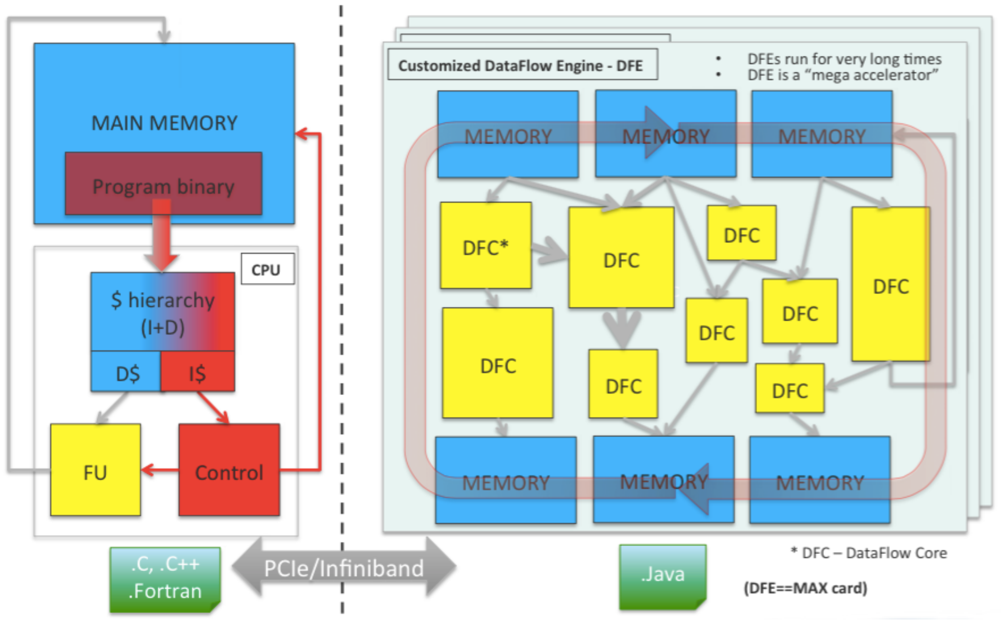


Figure 6: Parallel execution in data-flow system.

Inside the DFE, data-flow cores carry out the accelerated arithmetic and logic operations. Multiple data-flow cores form a compute kernel, and a so-called manager is responsible for managing the connections between the separate kernels, the connections to off-chip resources such as LMEM memory, and the various PCIe, Infiniband and MaxRing interconnects. Data-paths within kernels are deeply pipelined without any synchronisation concerns. During kernel development, a data-flow developer simply focuses on realising large degrees of parallelism and pipelining without having to worry about synchronisation or scheduling. MaxCompiler will perform the scheduling of operations and balancing the data paths inside a kernel automatically. A manager configuration (not shown in Figure 6) is used to create the connections between the compute kernels and LMEM memory, CPU host memory and other IO interfaces.

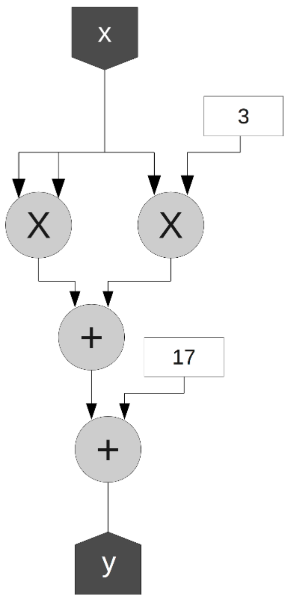
To program a data-flow engine, we create a completely parallel and fixed data-flow structure that can perform computations by simply streaming data through it. To illustrate this concept, we show how a simple loop computation can be transformed into a data-flow kernel. Let us consider an example where we want to calculate $y = x^2 + 3x + 17$ over a data set. A conventional C program requires a *for* loop to repeat the computation over a dataset even though there is nothing inherently sequential in this computation:

```
for (i = 0; i < numDataElements; i++)  {
    x = input[i];
    y = x*x + 3*x + 17;
    output[i] = y;
}
```

Figure 7 (left) shows a simple data-flow kernel representing the same computation. The operations that are located inside the body of the loop can be carried out by a fixed pipeline with two multipliers and two adders. The *for* loop is removed by using streaming inputs and outputs that are either connected to LMEM memory or CPU host memory. The arithmetic operations are also carried out concurrently rather than in sequence. A practical data-flow implementation can contain thousands of operators in a data-path all working concurrently (see Figure 7, right). The MaxJ kernel description that can generate this data-path is as follows:

```
class SimpleCalc extends Kernel {
    SimpleCalc() {
        DFEVar x = io.input("x", dfeFloat(8,24));
        DFEVar y = x * x + 3 * x + 17;
        io.output("y", y, dfeFloat(8,24));
    }
}
```

The MaxJ description begins by extending the kernel class. The *kernel* class is part of the Maxeler Java extensions and the user develops their own kernels by using inheritance. Next, we define a constructor for the class. It is important to point out that this MaxJ program will only run once to build the DFE configuration; the constructor will facilitate building the data-flow implementation. To create the streaming inputs and outputs for the kernel, the methods *io.input* and *io.output* are used. Streaming inputs and outputs replace the *for* loop in the original C code that iterates over data. The input method also allows to fully customise the input number format. In this case, we use a standard single precision floating point format (8-bit exponent and a 24-bit mantissa), but MaxJ also supports custom data types that can be defined by the user. This is useful when optimising the numerical behaviour and performance. The computation itself is expressed in a very similar way as in the original C code. A variable type *DFEVar* is used to handle all streaming data.



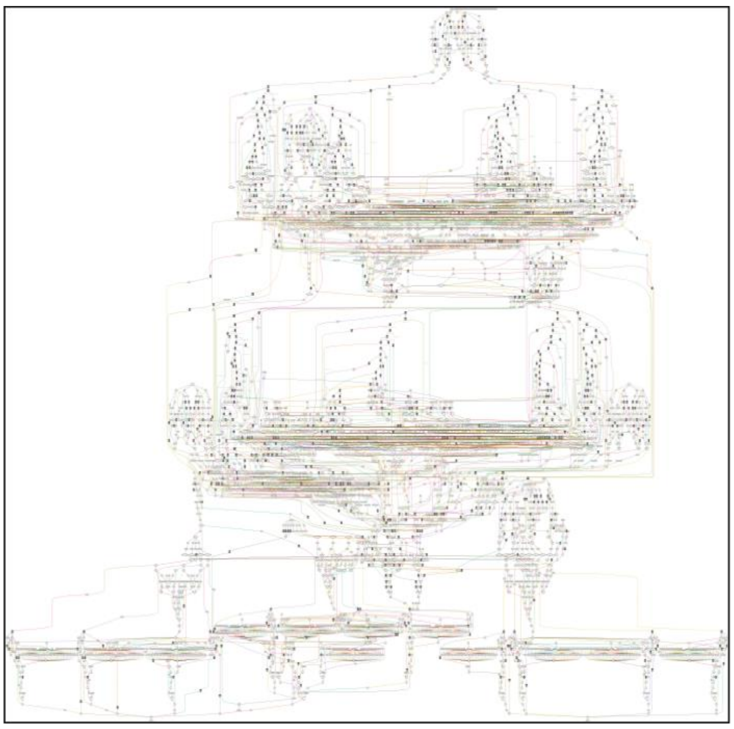


Figure 7: A simple data-flow graph (left). All arithmetic operations are carried out in parallel. A practical data-flow application (right) with 5000 arithmetic operators running concurrently in a data-flow pipeline.

Another example of a MaxJ dataflow description is show below. The code performs a moving average computation over three data elements. The resulting dataflow kernel that is generated by MaxCompiler is shown in Figure 8. As it can be seen, a counter and a ternary operator implement a highly customised control structure tightly coupled with the dataflow path. This is an example of handling control in a dataflow kernel and in this case it handles the boundary conditions when no valid data are present. Another

European Commission
Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628

important construct are stream offsets which allow access to data elements that are ahead or behind the current element in the stream.

```
class MovingAv extends Kernel {
    MovingAv() {
        DFEVar x = io.input("x", dfeFloat(8,24));
        DFEVar x_prev = stream.offset(x, -1);
        DFEVarx_next = stream.offset(x, +1);
        DFEVar cnt = control.count.simpleCounter(32, N);
        DFEVarvalid = (cnt > 0) & (cnt < (N-1));
        DFEVar y = valid ? (x_prev+x+x_next) / 3.0 : 0.0;
        io.output("y", y, dfeFloat(8,24));
    }
}
```

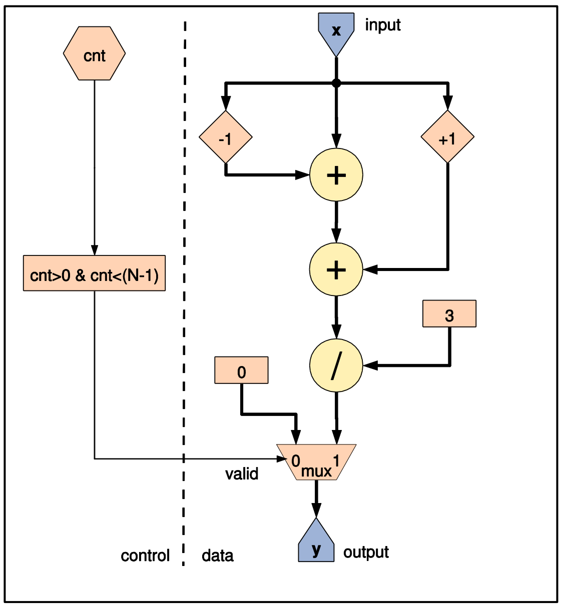


Figure 8: Example of a DFE kernel performing a moving average computation.

The result of compiling a MaxJ description using MaxCompiler is a binary file containing the FPGA configuration (referred as the .max file) that can be linked with the host application. Maxeler provides the SLiC API to invoke the max file from the host application code. SLiC provides various abstraction layers that let the programmer call the DFE with one simple function call or control it in more advanced ways. MaxCompiler automatically generates the necessary function prototypes. MaxelerOS is a run-time layer that handles interactions between host applications making SLiC calls and the DFEs. It coordinates the use of DFE resources at run time, and manages the scheduling and data movement within Maxeler systems. More information on MaxJ and SliC can be found in the compiler documentation[9].

Code development can be done in MaxIDE, an Eclipse-based integrated development environment that provides a unified view of that project that includes kernel and manager code in MaxJ and the host application. It also provides a simulation and debugging environment for the project.

A noteworthy characteristic of the DFE compute model is that the computation inside the DFE is entirely deterministic and predictable, making it easy to analyse and alleviate performance bottlenecks. This means the design can be analysed and optimised using simple spreadsheet calculations even before it has been compiled. The performance of a data-flow engine in terms of operations per second generally increases with the number of operations specified in a MaxJ data-flow design. Since all operations run in parallel, having more operations in the code automatically translates into more computations per fixed unit of time. The performance limit is reached when either the reconfigurable chip is completely filled up with arithmetic operators or the available memory bandwidth is fully consumed. Ideally, both should be utilised as close to 100% as possible.

3.3 Programming Models for GPUs

In this Section we give a brief overview of the prevalent approaches for heterogeneous programming in high-performance computing, in particular focussing on code acceleration with GPUs.

3.3.1 C Language Interfaces

As GPU programming is concerned with low-level machine details, it should come as no surprise that the main programming models for GPUs are based on the C language. The Compute Unified Device Architecture (CUDA)[10] is NVIDUA's proprietary

[9] Multiscale Dataflow Programming, Version 2015.2, Maxeler Technologies THIS NEEDS A WEB REFERENCE

[10] Parallel Programming and Computing Platform. http://www.nvidia.com/object/cuda_home_new.html Accessed 01 February 2016

programming model for GPUs. The term 'unified' implies that the same programming model applies across all programmable NVIDIA GPUs, even though these GPUs have very different architectures. As such, the code will be functionally correct on all supported GPUs, but the performance of a CUDA program can vary strongly between GPUs.

The architecture of GPUs differs between high-end and low-end GPUs and generally becomes more capable as device integration improves over time. As they have different numbers of compute units, internal registers, different amounts of local and global memory and different memory bandwidths, it may be necessary to restructure the code when moving from one type of GPU to another if one desires to maintain optimal performance. In other words, the way the code is structured optimally is intertwined with the specific GPU implementation.

OpenCL (Open Compute Language)[11] is an open standard for programming heterogeneous systems. It defines an abstraction of an accelerator that matches very well with a large class of accelerators, including GPUs, Field Programmable Gate Arrays (FPGAs), the Cell Broadband Engine[12] and mainstream CPUs. The programming model is in many aspects similar to CUDA and maps well onto it. OpenCL code is prone to the same limitations of performance portability as CUDA. However, OpenCL code is functionally portable across a larger number of accelerators from a range of vendors. As such, the performance portability problem can be considered to be more severe[13]. Both CUDA and OpenCL define a kernel as a section of code that is applied across an iteration range. The iteration range can be 1, 2 or 3-dimensional. The kernel code is written in a simplified version of the C99 standard, where additional keywords have been introduced to tag procedures that are kernels, as opposed to those that are auxiliary procedures for kernels.

[11] Khronos Group. The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/ Accessed 01 February 2016.
[12] H. P. Hofstee. Power efficient processor architecture and the cell processor. In Proceedings of the 11th International Symposium on High-Performance Computer Architecture, HPCA ’05, pages 258–262, Washington, DC, USA, 2005. IEEE Computer Society.
[13] S. Rul, H. Vandierendonck, D’Haene J., and K. De Bosschere. An experimental study on performance portability of OpenCL kernels. In Symposium on Application Accelerators in High Performance Computing, page 3, July 2010.

3.3.2 Java-C Bindings

The Java Native Interface (JNI)[14] is the basic mechanism for creating bindings between Java programs and low-level, machine-specific software. The JNI interfaces with the Java object system and garbage collector to export a safe description of Java data structures to external programs. It provides an efficient means to communicate between Java programs and the outside world; however, it breaks many useful properties of Java code such as out-of-bounds checking for arrays and pointer safety.

JCUDA[15] is a set of JNI bindings between Java programs and the C-level CUDA library. It provides an efficient means for Java programs to communicate with GPUs. However, it is programmed as if CUDA is used from a C program, which involves much boilerplate code and hardware-specific assumptions.

JOCL[16] is for OpenCL what JCUDA is for CUDA: an interface to the standard OpenCL library. Hereby, the OpenCL programming model is available to Java programs.

3.3.3 Java-Language Integration

While native interfaces are available and provide high performance, they break the programming abstractions built up and guaranteed by Java. As such, they are not a desirable set of extensions for GPU programming in Java. Several programming environments exist, however, that aim at providing a true Java-based GPU programming environment.

3.3.3.1 JaBEE

JaBEE[17] is a Java Binary Execution Environment that provides an abstract base class for GPU kernels. Deriving classes should implement a **run** method that corresponds to the

[14] Java SE 7 java native interface-related APIs and developer guides http://docs.oracle.com/javase/7/docs/technotes/guides/jni/ Accessed 01 February 2016
[15] Java bindings for CUDA http://www.jcuda.org/ Accessed 01 February 2016
[16] Java bindings for OpenCL http://www.jocl.org/ Accessed 01 February 2016
[17] W. Zaremba, Y. Lin, and V. Grover. JaBEE: Framework for object-oriented java bytecode compilation and execution on graphics processor units. In Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, GPGPU-5, pages 74–83, New York, NY, USA, 2012. ACM.

kernel method, specifying what operations to execute for a specific thread in a thread group. This method is intercepted by a compiler and automatically translated to GPU assembly code. It has many similar restrictions as Aparapi, discussed below. However, contrary to Aparapi, it does allow method calls on objects on the GPU side, including calls to virtual methods.

3.3.3.2 HSAIL

The Heterogeneous System Architecture (HSA)[18] is designed to efficiently support a wide range of data-parallel and task-parallel programming models and to support multiple instruction sets based on CPUs and accelerators, including GPUs. It bridges the diversity in programming models and hardware through HSAIL[19], a virtual instruction set architecture (ISA).

3.3.3.3 Sumatra

HSAIL is a good candidate for an intermediate target for code generation, among others, Java code translated to GPU instruction sets. The Sumatra[20] project aims to do exactly that. It aims to offload selected Java Stream API method calls. The stream API provides a functional way to operate on collections of data, offering operations such as **map**, **filter**, **reduce**, etc. Sumatra furthermore supports usage of Java objects, as well as lambda functions. However, restrictions apply in regards to data types. For example, stream operations must iterate over **IntRange**, **Array** or **ArrayList** in order to be ported to the GPU. The **reduce** operation can only work with primitive integers.

While it offers an interesting perspective, development on the Sumatra project is currently suspended as planned additions to Java 9 are crucial for its further development.

[18] HSA Foundation http://www.hsafoundation.com/standards/
[19] HSA Programmer Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer, and Object Format (BRIG), 1.0.1. edition http://www.slideshare.net/hsafoundation/hsa-programmers-reference-manual-hsail-virtual-isa-and-programming-model-compiler-writers-guide-and-object-format-brig-version-95 .
[20] Project Sumatra, OpenJDK website http://openjdk.java.net/projects/sumatra/

3.3.3.4 Aparapi

Aparapi[21] is a Java library that converts Java bytecode to OpenCL at runtime. The OpenCL code can then be executed by a GPU and, in principle, any OpenCL-compliant device. Aparapi provides an abstract class Kernel that contains a **run** method which is overridden to define the data parallel algorithm. The **run** method is constructed similarly as the a CUDA or OpenCL kernel as it is executed multiple times, each time applied to a specific index in the iteration range of the kernel. The iteration range is defined separately in the startup method of the kernel.

Aparapi hides most of the complexity of writing OpenCL code while maintaining common Java programming idioms and the use of the JVM to handle memory management. Aparapi, however, imposes limitations to the types of data and operations that can be used within a kernel. In particular, instances of Java objects cannot be used, Java Class Library and user defined objects alike.

As a result, no calls to **new** can be made from within the **run** method. This means that any data that is sent over to the GPU using Aparapi must be primitive and pre-allocated. As well as this, only one-dimensional primitive arrays can be used, as a multi-dimensional Java array is actually an array of array objects rather than a true multi-dimensional array. Other Java features such as exceptions, overloaded methods, for-each style loops and non-final static fields are not supported. Aparapi supports execution model on the CPU by making use of the Java Thread Pool (JTP). The JTP is used as a fallback if the kernel code is not compilable for the GPU, e.g. when compilation fails due to hardware constraints, or when the code violates the limitations set out above. This is useful also for heterogeneous clusters, where not all nodes are equipped with a GPU, or have incompatible GPUs.

An alternative branch of Aparapi (the ”lambda branch’”) interfaces Aparapi with HSAIL. As such, this branch is compatible only with AMD GPUs and Advanced Processing Units (APUs)[22]. This branch of Aparapi allows kernels to access Java objects and multi-dimensional arrays. Moreover, the shared address space model of HSA implies that the

[21] What is Aparapi? http://aparapi.github.io/ Accessed 01 February 2016
[22] P. Rogers. Heterogeneous system architecture overview, August 2013. Hot Chips Tutorial.

CPU and GPU access the same global address space and can share data without explicit transfers. There exists also an HSA emulator that can be used for functional testing and development in the absence of real hardware.

3.3.3.5 RootBeer

Rootbeer[23] allows programmers to express GPU kernels in Java code. The programmer defines a base class that kernels need to specialise. The kernel code is expressed in a way similar to CUDA and OpenCL, working under the assumption that the sequential kernel code is instantiated repeatedly for every coordinate in the iteration range of the kernel. Rootbeer uses Java byte-code inspection and translation and builds on the Soot library for byte-code introspection[24]. Soot allows an application to traverse the abstract representation of Java byte-codes and to generate corresponding GPU assembly code. This GPU code is subsequently compiled and execute don the GPU.

Rootbeer is more complete than Aparapi, as it supports full access to the Java language. It succeeds in this by serialising and de-serialising all data accessed by the kernel prior to sending the data over. It is however not clear how efficiently each Java feature can be executed by the GPU.

3.3.4 Performance Considerations

Few papers have analysed the performance differences between the cited acceleration programming models and systems. Docampo *et al*[25] compare Aparapi against JCUDA,

[23] Philip C. Pratt-Szeliga, James W. Fawcett, and Roy D. Welch. Rootbeer: Seamlessly using gpus from java. In Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems, HPCC '12, pages 375–380, Washington, DC, USA, 2012. IEEE Computer Society.
[24] R. Valle´e-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON '99, pages 13–. IBM Press, 1999.
[25] J. Docampo, S. Ramos, G.L. Taboada, R.R. Exposito, J. Tourino, and R. Doallo. Evaluation of java for general purpose gpu computing. In Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on, pages 1398–1404, March 2013.

native CUDA and CPU-only Java code using the SHOC benchmarks[26]. It shows that JCUDA results in up to 4 times slower execution than native CUDA. Aparapi can be between 2x faster and 8x slower than JCUDA, depending on the kernel.

3.4 Programming Models for the Xeon Phi

"Xeon Phi" is the brand-name of a series of processors based on Intel's Many Integrated Core (MIC) architecture that can act as accelerators. At the time of writing this document, the current generation of Xeon Phi is named "Knights Corner" (KNC). KNC Xeon Phi functions as an accelerator, like a GPU, attached via a PCI-Express link. The next generation of Xeon Phi, which is to be named as "Knights Landing" (KNL), will be released in late 2016 and will be shipped as two products. The initial release will act as a stand-alone MIC processor. Later versions will follow the KNC paradigm of being attached to a CPU, with the option of PCI-Express or the much faster OmniPath interconnect. The following discussion will be focused on the KNC Xeon Phi as an attached processor; however the user should find that most of it is applicable to its next iterations, even as a stand-alone processor.

3.4.1 MIC Hardware Architecture

The MIC architecture provides three important layers of assets which can be exploited for massive parallelism. Any potential developer should be aware of these assets and actively try to maximize their usage, in order to achieve high performance.

Firstly, as its name suggests, MIC processors offer a plethora of cores which can communicate by using the platform's shared DRAM. KNC processors feature up to 61 cores, whereas future generations will increase this number. Considering the emphasis of the architecture on high bandwidth (up to 320 GB/s for KNC), communication between the cores should not act as a bottleneck and hence, the developer is encouraged to use the entirety of available hardware cores in parallel fashion for his application.

[26] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU-3, pages 63–74, New York, NY, USA, 2010. ACM.

PARTNERS
Institute of Communications and Computer Systems, Maxeler Technologies, Bull Systems, Queen's University of Belfast, Foundation for Research and Technology, The Hartree Centre / Science and Technologies Facilities Council, Neurasmus BV, Neurocom Luxembourg, Hellenic Exchanges SA, Holding, Clearing, Settlement and Registry, LeanXcale, Loba

European Commission | Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628

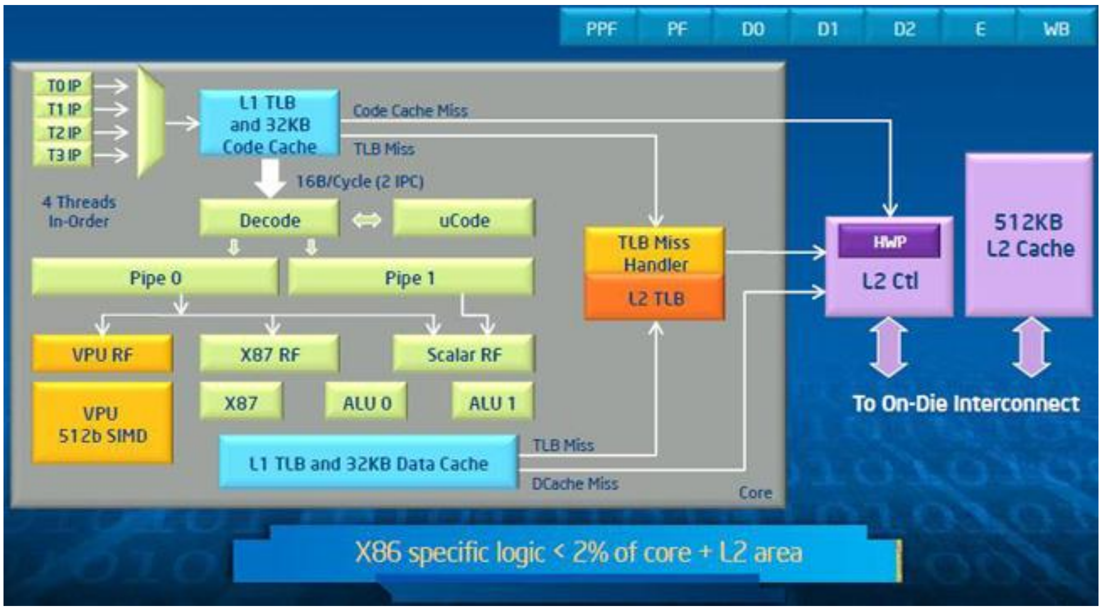


Figure 9. Architecture of an Intel® Xeon Phi™ core[27].

Secondly, each core features multi-threading technology, supporting up to 4 threads executing on a core at the same time. This essentially means for a developer that KNC processors can offer up to 244 threads operating simultaneously and their efficient usage is critical to the platform's performance.

Finally, each core uses 512-bit-wide Vector Processing Units (VPUs) which allow up to 16 single-precision, or 8 double-precision, operations per cycle. The VPUs are the primary challenge the developer faces when trying to extract good performance out of the MIC. Complicated code with many conditional jumps needs to be re-written in a more streamlined fashion so that it becomes more VPU-friendly, otherwise the card's performance will most likely be unimpressive. It should be noted that there is a variety of tools and well-written documentation on Vectorization and SIMD-instructions to aid any potential developer.

[27] George Chrysos, Intel Corporation, Intel® Xeon Phi™ X100 Family Coprocessor https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

PARTNERS
Institute of Communications and Computer Systems, Maxeler Technologies, Bull Systems, Queen's University of Belfast, Foundation for Research and Technology, The Hartree Centre / Science and Technologies Facilities Council, Neurasmus BV, Neurocom Luxembourg, Hellenic Exchanges SA, Holding, Clearing, Settlement and Registry, LeanXcale, Loba

European Commission | Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628

3.4.2 Modes of Operation

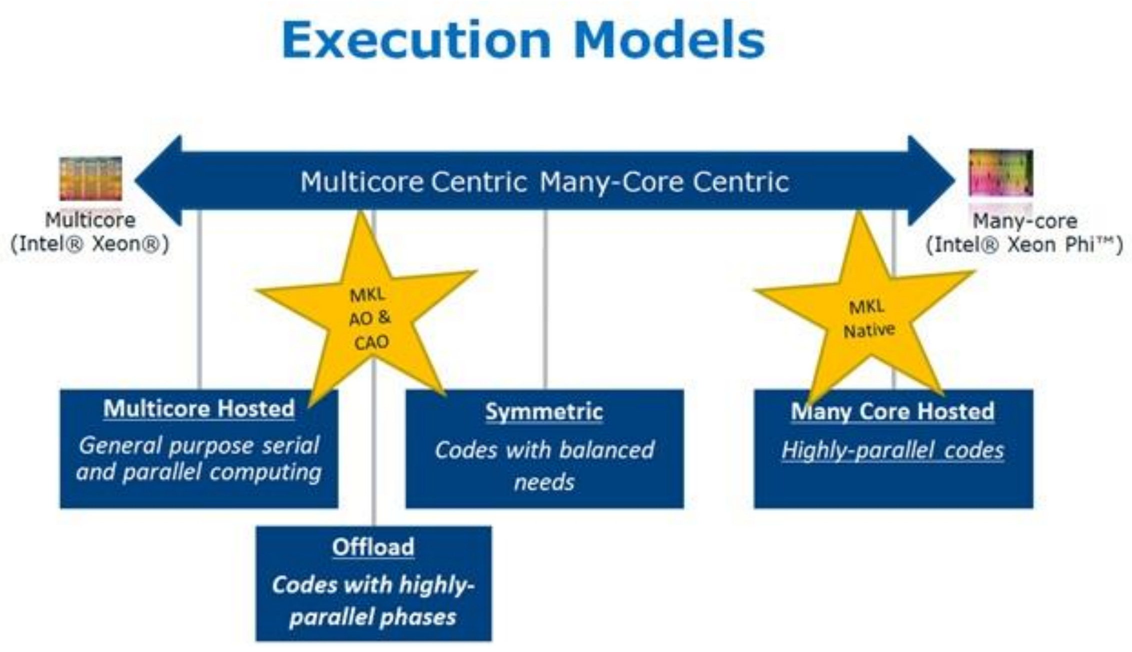


Figure 10. Different execution models on a typical node with Xeon CPU and Xeon Phi accelerator[28].

As a PCIe card, the Xeon Phi is accessible via Secure Shell (ssh) connection from the host; this is feasible due to the fact that a light Linux image is booted on the platform on the node's start-up. The user can treat the Xeon Phi as a stand-alone processor, executing any code natively, as long as it is compiled with the Xeon Phi as its target architecture. This is usually the optimum approach for highly parallel applications.

Alternatively, a code can run simultaneously on both the Xeon host and the Xeon Phi. This is achievable in two ways: two different instances of the code can run on the two machines, each compiled in accordance to the different underlying instruction set. The two instances can communicate and synchronize as needed by treating the two machines as different nodes in a virtual TCP/IP network, using widely-known methods such as message-passing libraries (e.g. MPI). Alternatively, an application can be developed with a host-and-accelerator paradigm in mind. Serial parts of the code are executed on the Xeon host, which then assigns highly parallel regions to the Xeon Phi

---

[28] Noah Clemons, Intel Corporation, Recommendations to choose the right MKL usage model for Xeon Phi https://software.intel.com/en-us/articles/recommendations-to-choose-the-right-mkl-usage-model-for-xeon-phi

via offload pragmas. This model of operation is strongly connected to the GPU-coding paradigm. As with programming for GPUs, account must be taken of the relatively slow communication between the host and the accelerator ensuring that pieces of work to be offloaded are large enough to deliver a performance advantage despite the data transfer costs.

Finally, heavily serial applications should not be run on the Xeon Phi processor at all; relying on the Xeon host's superior single-threaded performance.

3.4.3 Programming Frameworks

At its core, the MIC architecture is based on its well-known x86 counterpart. In essence, KNC cores are Pentium cores with heavy modifications to boost performance and support SIMD instructions. As such, tools used to write parallel code on any x86-based machine can be used for the Intel Xeon Phi. One of the most prominent tools for C-language applications is OpenMP, which has been noted as having excellent results on the platform, particularly for native execution. The MPI message passing library is another candidate, since it allows symmetric execution of code between the host and the Xeon Phi. It is also imperative to use MPI-like libraries for multi-node implementations of an application on any cluster. For multi-node clusters, combining the two tools is considered as a good practice. MPI functions allow for multiple machines to communicate, handling inter-node operations, whereas OpenMP shows superior performance when parallelizing intra-node tasks. This style of coding is often referred-to as "Hybrid MPI and OpenMP Parallel Programming".

Another notable tool for writing parallel applications on the machine is Intel Thread Building Blocks (TBB). This is a C++ tool which allows the breaking-up of a workload in tasks and assigns them to worker-threads. The tool exhibits similarities to the OpenMP paradigm. Finally, applications making heavy use of mathematical functions may be able to take advantage of Intel Math Kernel Library (MKL), which offers high-performance functions solving well-established mathematical problems, such as Linear Algebra, Fast Fourier Transforms (FFT) and statistical functions. The library supports Xeon Phi platforms since its 11.0 – Update 2 version and is heavily optimized for taking advantage of the platform's wider VPUs.

European Commission | Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628

3.4.4 Performance Monitoring

Intel has released a series of tools aiding the developer working on the MIC architecture in order to verify whether their application is working properly on the platform. Intel's VTune Amplifier is fairly straightforward tool that offers a wealth of information for any application running a variety of platforms. The tool supports the Xeon Phi architecture, offering useful insights into how efficiently its assets, particularly the VPUs and the caches, are used. It should be noted however, that due to its hardware-event sampling during the execution time, its time-overhead can be overbearing, particularly for large analyses and irregular memory-access patterns. Furthermore, some of its metrics, such as the estimated average usage of the VPUs (called Vectorization Intensity), are not always reliable for long and complex codes.

Another tool of note from the same suite is Intel Advisor, which offers tips on how to properly thread and vectorise an application. Finally, Intel ships the Xeon Phi with some tools built-in its operating system for power monitoring. The *micsmc* tool allows the host CPU to monitor and configure the Xeon Phi card's status, including device performance, driver info, temperatures, core usage, etc. However, little or no support is offered in regards to dynamically scaling the cores' levels of power consumption.

4 Background on Data Analytics Platforms

Data analytics platforms have been designed to simplify the job of the data analyst, namely to perform analytics on terabyte-size data sets. Data analytics platforms typically specialize on a specific set of workloads, e.g. batch processing, stream processing or graph analytics. Data analytics platforms are typically designed to execute on scale-out clusters of commodity processors.

4.1 Batch Processing

Batch processing involves the processing of a large quantity of data. No specific time bounds are set on the processing although it is typically hoped that the processing does not take too much time.

European Commission | Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628

We can distinguish two generations of batch processing platforms. The first generation, with Hadoop as its main example, is a programming framework centred on the map-reduce parallel pattern. This parallel pattern allows the representation of many computations in a way that allows parallel execution, but does not require the programmer to coordinate the execution of tasks. Coordination, as well as data partitioning and data distribution, is handled by Hadoop. Inputs and outputs of map and reduce tasks are streamed from and to disk. As such, the performance of Hadoop is strongly dominated by disk access times.

The second generation of batch processing platforms attempts to hold the intermediate data sets in memory, thus significantly improving performance. The key example of such systems is Spark.

Both Hadoop and Spark are organized as master/slave systems. Support for redundancy is built-in and varies between masters and slaves. Slaves are requested by the master to execute well-defined tasks. In case a slave fails, or is simply slow to respond, the same task can be executed or re-executed by another slave. Failed slaves are automatically restarted when they stop sending heartbeat messages at the appropriate rate. Masters are protected against failures through a redundancy scheme.

4.1.1 Acceleration of Batch Processing

Various techniques to accelerate Hadoop and Spark have been proposed in the literature. These techniques employ GPU and/or FPGA acceleration programming frameworks and aim to integrate the accelerated code as neatly into the analytics framework as possible. Typically, however, analytics platforms are programmed using high-level languages executed on managed runtimes, which stands in stark contrast with the low-level programming approaches used for accelerators. Much attention has been paid to this issue, which is typically resolved through using systems such as JOCL, JNI and Aparapi to make the link between managed runtime and accelerator[29,30,31]. Other issues that have been researched are the buffer management.

[29] S. Okur, C. Radoi, and Y. Lin, "Hadoop+aparapi: Making heterogenous mapreduce programming easier," 2012, http://www.semihokur.com/docs/okur2012-hadoop - aparapi.pdf.

European Commission | Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628

GPU buffers are pre-allocated with fixed size. It is however typically a priori unknown how much data will be produced by analytics codes. This can be addressed by making two passes over the data set: once to compute the required buffer size and one to produce the data.[32]

There are several research efforts towards the acceleration of data analytics applications based on distributed programming frameworks such as Hadoop and Spark.

One of the first attempts to accelerator cloud computing application using FPGA was presented by Microsoft and Tsinghua University[33]. In this work a MapReduce framework on FPGA, which provides programming abstraction, hardware architecture, and basic building blocks to developers is presented. The performance evaluation of the proposed system has been performed using the RankBoost application[34] that is used for page ranking. The most time consuming procedure of RankBoost is WeakLearn, which consumes more than 95% execution time and it is the one that is ported to the FPGA[35]. Both the mapper and the reduce tasks of the WeakLearn algorithm have been mapped to the FPGA. To test the performance of the RankBoost acceleration on FPMR, a real world dataset for a commercial search engine is used. This time-consuming procedure achieves up to 16.74× speedup in the FPMR framework while the overall system speedup is 14.44×.

[30] R. Nitu, E. Apostol, and V. Cristea, "An improved gpu mapreduce framework for data intensive applications," in Intelligent Computer Communication and Process- ing (ICCP), 2014 IEEE International Conference on, Sept 2014, pp. 355–362.
[31] M. Grossman, M. Breternitz, and V. Sarkar, "HadoopCL: Mapreduce on distributed heterogeneous platforms through seamless integration of Hadoop and OpenCL," in Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International, May 2013, pp. 1918–1927.
[32] B. He, W. Fang, N. K. Govindaraju, Q. Luo, and T. Wang, "Mars: A mapreduce framework on graphics processors," in IEEE Conference on Parallel Architectures and Compilation Techniques. Oct 2008, pp. 260–269.
[33] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "Fpmr: Mapreduce framework on fpga," in Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays, ser. FPGA '10. New York, NY, USA: ACM, 2010, pp. 93–102
[34] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer, "An efficient boosting algorithm for combining preferences," J. Mach. Learn. Res., vol. 4, pp. 933–969, Dec. 2003
[35] N.-Y. Xu, X.-F. Cai, R. Gao, L. Zhang, and F.-H. Hsu, "Fpga acceleration of rankboost in web search engines," ACM Trans. Reconfigurable Technol. Syst., vol. 1, no. 4, pp. 19:1–19:19, Jan. 2009

An architecture for the FPGA acceleration of MapReduce applications has been presented also by D. Yin et al.[36] . A cluster of worker nodes is designed for the MapReduce framework, and each worker node consists of both a CPU-based worker and an FPGA-based worker. The CPU-based worker runs the major communications with other worker node and tasks, while the FPGA-based worker operates extended MapReduce tasks to speed up the computation processes. The proposed framework has been implemented by modifying the open-source Hadoop project and mapped to the NetFPGA boards. The CPU worker runs the modified Hadoop MapReduce program which processes file system requests and transmits data to FPGA workers. The proposed framework has been evaluated using two typical applications of the MapReduce framework: matrix multiplication and page ranking. For the case of matrix multiplication using one FPGA board, the proposed system can achieve almost 15× speedup compared to CPU. In the case of the page ranking, the proposed system can achieve approximately 4× faster execution time compared to the software execution.

An integrated FPGA architecture is proposed for the efficient implementation of the MapReduce framework by NTUA and DUTH[37,38]. The proposed architecture implements the Phoenix MapReduce framework that is a C-based version of MapReduce. In one case, a HW-SW co-design is presented where the Map tasks are executed in the processors and a specialized hardware accelerator is implemented for the efficient processing of the Reduce tasks. In the second architecture (an integrated framework is proposed where the whole application is mapped to the FPGA. The Map computational kernels, that are usually application-specific, are created using High Level Synthesis (HLS) tools and the Reduce tasks, which are common to most of the applications, are executed using the common Reduce hardware accelerator. The presented system proposes the complete decoupling of MapReduce tasks' data-paths to distinct busses,

[36] D. Yin, G. Li, and K.-d. Huang, "Scalable mapreduce framework on fpga accelerated commodity hardware," in Internet of Things, Smart Spaces, and Next Generation Networking, ser. Lecture Notes in Computer Science, S. Andreev, S. Balandin, and Y. Koucheryavy, Eds., vol. 7469. Springer Berlin Heidelberg, 2012, pp. 280–294

[37] C. Kachris, D. Diamantopoulos, G. C. Sirakoulis, and D. Soudris, "An fpga-based integrated mapreduce accelerator platform," Journal of Signal Processing Systems, pp. 1–13, 2016

[38] C. Kachris, G. C. Sirakoulis, and D. Soudris, "A reconfigurable mapreduce accelerator for multi-core all-programmable socs," in System-on-Chip (SoC), 2014 International Symposium on, Oct 2014, pp. 1–6.

accessed from individual processing engines. Such a dataflow approach implies a holistic C/C++ to RTL domain-level MapReduce transition. The performance evaluation shows that the proposed scheme can achieve up to 4.3× overall speedup (system speedup) in MapReduce applications while offering significant lower power and energy consumption compared to a high-end multi-core processor. Specifically, it can provide up to 25× lower power consumption and up to 33× better energy efficiency compared to the software-only solution in the low-power cores.

The University of Hong Kong has presented the design and implementation of the k-means clustering algorithm on an FPGA-accelerated computer cluster[39]. The implementation followed the MapReduce programming model, with both the map and reduce functions executing autonomously on the CPU and on multiple FPGAs. A hardware/software framework was developed to manage the execution on multiple FPGAs across the cluster. The experiment was run on three compute nodes, each containing a KC705 FPGA board from Xilinx. Each KC705 board contains a Kintex-7 FPGA connected to the CPU through a PCIe x3 interface. When compared to a similar software implementation executing over the Hadoop MapReduce framework, from 15.5× to 20.6× performance improvement has been achieved across a range of input data sets.

4.2 Stream Processing

In stream processing streams of messages arrive in the system and need to be processed. Typically, messages are handled in multiple stages with each stage emitting messages to the next. Stages can be interlinked in arbitrary complex ways. The key example in this area is the Storm framework. Kafka is a related system providing a publish-subscribe service. As with Hadoop and Spark, these systems are designed to execute on scale-out systems and provide inherent fault-tolerance.

[39] K. Neshatpour, M. Malik, M. A. Ghodrat, A. Sasan, and H. Homayoun, "Energy-efficient acceleration of big data analytics applications using fpgas," in Big Data (Big Data), 2015 IEEE International Conference on, Oct 2015, pp. 115–123

4.2.1 Acceleration of Stream Processing

We are aware of a few prior works in the area of accelerating stream processing. Pinnecke *et al*[40] have proposed GPU acceleration of stream processing in a row-oriented database management system. They transform the row-oriented tables into individual columns in order to make the data transfer to the GPU more efficient. Nakagawa *et al*[41] focus on the overlap of computation with communication in stream processing on GPUs. Georgakoudis *et al*[42] investigate how to evaluate heterogeneous servers with different energy and performance, although not accelerated, for streaming workloads. Their work can form a basis to evaluate the accelerated servers as well.

5 Requirements

WP5 of the VINEYARD project investigates and develops the VINEYARD programming model, which consists of software support for heterogeneous computing, and the VINEYARD acceleration library.

The VINEYARD programming model defines language extensions, library support and a runtime system for expressing and executing programs on heterogeneous distributed systems. The VINEYARD programming model must be designed in such a way that it can achieve the required non-functional properties of the programs, in particular those properties and metrics related to execution time and energy or power consumption.

The VINEYARD acceleration library is a repository of pre-defined kernels that capture commonly occurring computations in the area of data analytics. These kernels are optimized for FPGAs. Making these implementations available in a library facilitates the adoption of the VINEYARD system.

[40] M. Pinnecke, D. Broneske, G. Saake. Toward GPU Accelerated Data Stream Processing. In; 27th GI-Workshop on Foundations of Databases. 2015.
[41] S. Nakagawa, F. Ino and K. Hagihara. A middleware for efficient stream processing in CUDA. Computer Science – Research and Development. Springer. April 2010.
[42] G. Georgakoudis, C. Gillan, A. Sayed, I. Spence, R. Faloon, D. S. Nikolopoulos. Iso-Quality of Service: Fairly Ranking Servers for Real-Time Data Analytics. Parallel Processing Letters Vol. 25 No. 3. 2015.

The following sections elaborate the requirements for the VINEYARD programming model and acceleration library.

5.1 Programming Model

5.1.1 Programming Model Support for Accelerators

The programming model and runtime system should ideally support a wide range of accelerators. As discussed extensively in section 3, accelerators are accompanied by a variety of different programming models. The discrepancy between programming FPGAs and GPUs is significant as they employ fundamentally different abstractions to express parallelism. Maxeler DataFlow Engines represent programs as a fine-grain data flow graph and extract parallelism from the repeated application of the data flow graph to a stream of data. GPUs, on the other hand, use massively parallel programming models which may result in streaming data from the GPU global memory into the processor, but the resulting codes may also reuse smaller working sets in local memory. Unifying these programming models behind a single interface is beyond the scope of this project.

An attractive short-term solution is one where the programming model is extended to allow definition of a number of alternative implementations of the same code, e.g. one version for executing on the CPU, typically written in a high-level interpreted language such as Scala, Java or Python, one version compiled to a bitstream for execution on FPGAs, and one massively parallel version for execution on GPUs. The key problems to be addressed by the programming model relate to how to represent these alternative versions in a tractable way and how to understand the relative efficiency of various implementations.

The design of the programming model needs to be a careful trade-off between the tensions of programmer control, allowing expert programmers to control the runtime system through the programming model in order to optimize non-functional metrics such as performance and energy consumption, versus transparency, which implies that the optimization of non-functional properties of the program execution is transparent to the programmer or user. In general we desire to have transparency as it enhances

PARTNERS
Institute of Communications and Computer Systems, Maxeler Technologies, Bull Systems, Queen's University of Belfast, Foundation for Research and Technology, The Hartree Centre / Science and Technologies Facilities Council, Neurasmus BV, Neurocom Luxembourg, Hellenic Exchanges SA, Holding, Clearing, Settlement and Registry, LeanXcale, Loba

European Commission | Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628

productivity and time-to-market. Nonetheless, we need to retain the option of manual control and optimization of the program execution.

Summary of requirements:

- Support acceleration of big data analytics frameworks (e.g., Spark, Storm and Heron) with accelerators including at least FPGAs and if possible also GPUs and Xeon Phi accelerators.
- Support concise description of equivalent implementations of the same algorithm and a uniform interface for invoking these implementations.
- If necessary, support annotation of equivalent implementations with additional information in order to enable the VINEYARD runtime system to autonomously select one version over another, or to efficiently load-balance work across accelerators.
- Balance programmer control versus transparency through the design of the programming model.

5.1.2 Runtime System and Scheduling

The runtime system should be able to efficiently share data between the managed language runtime, such as the Java Virtual Machine or the Python VM, and the memory manager of accelerators. The latter are typically programmed at a low level of abstraction, implying that memory management and data layout is under full control of the programmer. The former, in contrast, uses automatic memory management supported by garbage collection. Data layout and memory allocation cannot easily be manipulated by programmers. It is however crucial that data is shared between both environments with little or no overhead. The conventional solution of serialization, translating a data set to a platform-independent bitstream, will therefore not lead to acceptable non-functional properties. Different mechanisms need to be designed to support data sharing between managed runtimes and accelerators. These mechanisms should have minimal impact on the managed language runtime and the programming API in order to facilitate adoption.

In the context of streaming data, there is a need for better scheduling policies. Data streams have varying message rates and data volumes and may have different computational complexity per message. Scheduling such streams is often done on a per message basis. It is possible to achieve non-functional properties of the system in a better way if we utilize fair-share allocation of data streams, i.e. scheduling streams rather than scheduling individual messages.

Scheduling in heterogeneous systems requires careful balancing of a number of parameters including job size, efficiency of executing a job on a particular resource, availability of resources, communication efficiency of the accelerator, etc. Many of these parameters are moreover dependent on the characteristics of the job, e.g. job size for batch processing, and stream arrival rate and message workload complexity for streaming systems. The VINEYARD runtime system should utilize these characteristics to schedule jobs pro-actively and judiciously based on run-time information.

Summary of requirements:

- Develop techniques to efficiently share data between managed language runtimes and low-level (bare-metal) programming environments typically used on accelerators.
- Develop scheduling techniques for variable-rate data streams to optimize throughput, resource utilization and/or energy efficiency building on the concept of fair-share allocation.
- Develop scheduling techniques for hybrid scheduling across CPUs and accelerators.

5.1.3 Data Distribution

Distributed systems for big data processing employ some form of data distribution to balance the workload across the distributed system. Adding heterogeneity into the mix adds a new dimension along which data needs to be distributed: the accelerators' private memories. The VINEYARD runtime system must be able to distribute data between CPU memory and accelerator memory. The accelerator memory can be used to store data for longer periods of time, thereby avoiding repeated data movement. As an optimization, data required by the accelerator but not used by the CPU need never be loaded in CPU memory. This requires moving directly from the master or I/O device to accelerators and by-passing CPU memory.

European Commission
Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628

Summary of requirements:

- Design a memory management subsystem to manage data distribution across CPU nodes and accelerators.
- Optimize workload schedulers by taking into account existing data allocation and minimizing data movement.

5.1.4 Virtualization

Data analytics are often performed on cloud infrastructures where the hardware is virtualized, e.g. through the VineTalk protocol developed in WP5 of the VINEYARD project. Here, accelerators will be virtualized as well. The VINEYARD runtime system needs to be tuned to execution scenarios using virtualized accelerators. This has an impact on memory management and scheduling. Memory management techniques must be aware that the memory space of a virtualized accelerator is shared between resources. Scheduling techniques must take into sharing of virtualized accelerators, which is possible, e.g. in the context of streaming workloads. In these cases, fair-share scheduling and co-scheduling of CPU threads with threads communicating with the accelerators is a necessity.[43]

Summary of requirements:

- Design scheduling strategies and runtime system support for virtualized accelerators.

5.2 Acceleration Libraries

The hardware accelerators that will be developed in VINEYARD will be based on FPGAs and dataflow engines than can be reconfigured to host several types of accelerators such as compression, encryption, and machine learning kernels. The kernels will be

[43] On the virtualization of CUDA based GPU remoting on ARM and X86 machines in the GVirtuS framework. Raffaele Montella · Giulio Giunta · Giuliano Laccetti · Marco Lapegna · Carlo Palmieri · Carmine Ferraro · Valentina Pelliccia · Cheol-Ho Hong · Ivor Spence · Dimitrios S. Nikolopoulos. International Journal of Parallel Programming, to appear.

mapped in the accelerators in the form of IP blocks as library components. These library components of IP blocks will be hosted in repositories (i.e. in the github repository), and the user will have the option to select and import the required accelerator based on the application requirements.

The following figure describes the overview of the acceleration libraries in the context of VINEYARD. The applications that are written in high-level languages will define the applications libraries that need to be hosted in the accelerators. The VINEYARD framework, based on the required accelerators will automatically import the accelerators in the form of IP blocks from the central repository (Accelerator IP Repository) and will feed the scheduler. The VINEYARD scheduler, based on the application requirements and the hardware resources (number of accelerators) will find the optimum configuration and partitioning of the resources. The hardware controller will then be used for the configuration and the programming on the hardware resources based on the partitioning that has been performed.

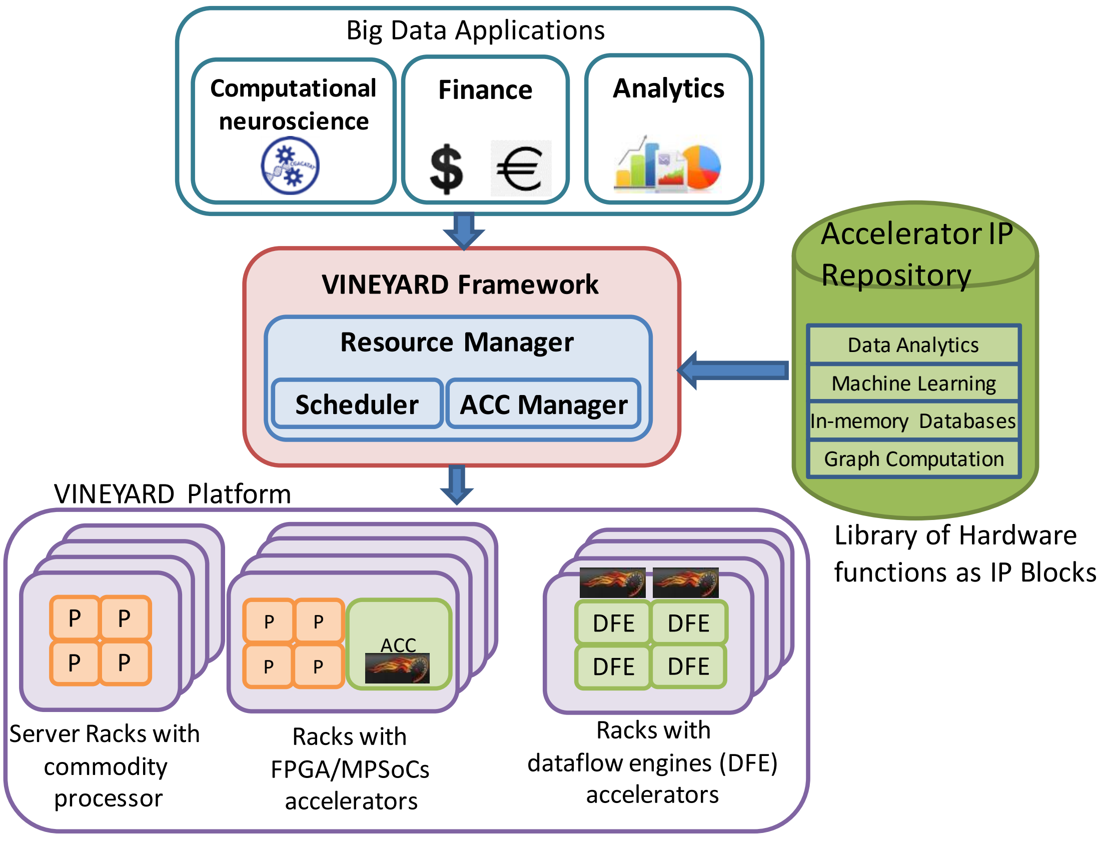

Big Data Applications
Computational neuroscience
Finance
$ €
Analytics
Accelerator IP Repository
VINEYARD Framework
Resource Manager
Scheduler
ACC Manager
Data Analytics
Machine Learning
In-memory Databases
Graph Computation
VINEYARD Platform
Library of Hardware functions as IP Blocks
P P P P
Server Racks with commodity processor
P P P P ACC
Racks with FPGA/MPSoCs accelerators
DFE DFE DFE DFE
Racks with dataflow engines (DFE) accelerators

Figure 11. High-level overview of the VINEYARD framework and the Acceleration library

PARTNERS
Institute of Communications and Computer Systems, Maxeler Technologies, Bull Systems, Queen's University of Belfast, Foundation for Research and Technology, The Hartree Centre / Science and Technologies Facilities Council, Neurasmus BV, Neurocom Luxembourg, Hellenic Exchanges SA, Holding, Clearing, Settlement and Registry, LeanXcale, Loba

European Commission | Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628

The programming model that will be developed in VINEYARD will deliver to the programmers a seamless and transparent utilization of the hardware accelerators by simply describing in the high level language the accelerators that will be instantiated. This can be achieved for example by replacing a specific function that is used for a task (e.g. compression) with a new function that calls the hardware accelerators for the specific task.

5.3 Summary

Table 1 gives a summary of all requirements.

Table 1 Summary of requirements of the VINEYARD programming model and runtime system

Component	Requirement
Programming model	Support acceleration of big data analytics frameworks (e.g., Spark, Storm and Heron) with accelerators including at least FPGAs and if possible also GPUs and Xeon Phi accelerators.
	Support concise description of equivalent implementations of the same algorithm and a uniform interface for invoking these implementations.
	If necessary, support annotation of equivalent implementations with additional information in order to enable the VINEYARD runtime system to autonomously select one version over another, or to efficiently load-balance work across accelerators.
	Balance programmer control versus transparency through the design of the programming model.
Runtime System	Develop techniques to efficiently share data between managed language runtimes and low-level (bare-metal)

European Commission | Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628

	programming environments typically used on accelerators.
	Develop scheduling techniques for variable-rate data streams to optimize throughput, resource utilization and/or energy efficiency building on the concept of fair-share allocation.
	Develop scheduling techniques for hybrid scheduling across CPUs and accelerators.
	Design a memory management subsystem to manage data distribution across CPU nodes and accelerators.
	Optimize workload schedulers by taking into account existing data allocation and minimizing data movement.
	Design scheduling strategies and runtime system support for virtualized accelerators.
Acceleration library	Define library of reusable accelerator IP blocks
	Optimize configuration of the IP blocks given available hardware resources

6 Conclusion

The VINEYARD aims of easy and transparent acceleration of data analytics using a choice of accelerators has been translated in a set of requirements. These requirements aim to address open issues in the research landscape around programming models and runtime system support for acceleration. Addressing these issues will result in a major step forward in the achievement of the VINEYARD goals.

European Commission
Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 687628