Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelor Thesis

Raphael Hiesgen

libcppa on SIMD machines - GPGPU computing using transparent C++11 actors and OpenCL

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Raphael Hiesgen

libcppa on SIMD machines - GPGPU computing using transparent C++11 actors and OpenCL

im Studiengang Technische Informatik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Thomas C. Schmidt
Zweitgutachter: Prof. Dr. rer. nat. Friedrich Esser

Abgegeben am October 24, 2013

Raphael Hiesgen

Thema der Bachelorarbeit
libcppa on SIMD machines - GPGPU computing using transparent C++11 actors and OpenCL

Stichworte
C++, Aktormodell, Nebenläufigkeit, GPGPU, OpenCL, heterogene Hardware, SIMD

Kurzzusammenfassung

In den letzten Jahren haben sich Grafikprozessoren (GPU) von dedizierten Pipelines zu Gruppen von datenparallelen programmierbaren Kernen entwickelt. Ihre Rechenleitung ist nicht mehr nur für graphische Instruktionen reserviert, sondern lässt sich für universelle Berechnungen nutzen. Eine hochparallele Architektur führt dazu, dass GPUs in manchen Bereichen eine deutlich höhere Leistung erzielen als CPUs. Spezielle Frameworks wie CUDA oder OpenCL haben sich als Programmierschnittstelle für diese Systeme etabliert. Das Ziel dieser Arbeit ist es, GPUs von einer höhere Abstraktionsebene aus programmierbar zu machen, indem wir GPGPU mit dem Actor Model vereinen. Das Actor Model schafft eine Abstraktion über parallele Systeme und bietet Möglichkeiten verteilte, fehlertolerante Systeme zu bauen. Im Speziellen beschäftigen wir uns mit OpenCL und der `C++11` Actor Library `libcppa`. Wir präsentieren den OpenCL Actor, der eine einfache Möglichkeit bietet, Code auf einer GPU auszuführen. Er ist in die Laufzeitumgebung von `libcppa` integrieret und ermöglicht transparentes Message Passing in verteilten Systemen auf heterogener Hardware. Actoren für die GPU werden durch die Funktion spawn_cl instanziiert, die das Setup für OpenCL im Hintergrund durchführt. Empfangene Nachrichten werden vom OpenCL Actor als Argumente für ein auf der GPU ausgeführtes Program genutzt. Die Ergebnisse aus der Berechnung werden anschließend an der Actor gesandt, der die Berechnung initiiert hat. Eine empirische Untersuchung der Mehraufwände zeigt deutlich, dass der induzierte Mehraufwand unserer Implementierung gegenüber den nativen OpenCL API vernachlässigbar ist.

Raphael Hiesgen

Title of Bachelor Thesis

libcppa on SIMD machines - GPGPU computing using transparent C++11 actors and OpenCL

Keywords

C++, actor model, concurrency, GPGPU, OpenCL, heterogeneous hardware, SIMD

Abstract

In recent years, graphics processing units (GPU) developed from single-purpose pipelines to clusters of data-parallel programmable units. Their performance is no longer reserved for graphics calculations, but can be used for general-purpose computing. A highly parallel architecture enables GPUs to outperform CPUs by several orders of magnitude for various use cases. Frameworks like OpenCL or CUDA give access to their capabilities. The objective of this work is to raise the abstraction level of GPGPU programming by combining it with the actor model. The actor model provides a high level approach to concurrency and offers mechanisms to built fault tolerant distributed systems. Our work focuses specifically on OpenCL and `libcppa`, an actor library written in `C++11`. We present the concept of the OpenCL-enabled actor, which offers an easy way to deploy code to GPUs. It is integrated into the runtime environment of `libcppa` and gives rise to transparent message passing in distributed systems and on heterogeneous hardware. Actors on the GPU are instantiated by the function spawn_cl, which handles the OpenCL setup process in the background. Messages sent to the OpenCL actor are used as arguments to trigger a kernel execution. Once the kernel finishes, the actor sent the results to the actor that requested the calculation. We examined our implementation for overhead over the native OpenCL API and find that the induced overhead is negligible.

Contents

1 Introduction

While modern CPUs moved towards multiple cores, GPUs developed from single purpose pipelines to clusters of data-parallel programmable units. This highly parallel hardware is no longer reserved for graphics, but available for any general purpose calculation that can be mapped to this architecture. Furthermore, it is available on a wide range of systems from servers to mobile devices. Regarding performance, GPUs outperform their CPU equivalents by an order of magnitude for several use cases. The best performance can be achieved with algorithms that can be split into independent tasks. However, most programs require a management for the performed tasks. Hence GPU computing is often combined with CPU computing to benefit from the advantages of both architectures.

GPGPU programming libraries offer a low-level, hardware related approach for programmers. A higher abstraction level transfers responsibilities from the programmer to the runtime environment in form of a library or framework. OpenCL is a standard for cross-platform, parallel programming developed by the Khronos Group. It enables the development of heterogeneous application for a wide range of platforms.

In the context of multi-core CPUs, the classic approach uses threads to parallelize the execution of a single program and locks to avoid race conditions. Since the developer is responsible for avoiding race conditions and deadlocks, this programming model is inherently error prone. A higher level of abstraction to handle multiple cores is offered by the actor system. It describes isolated entities called actors that communicate via message passing. An important part of the actor system is an error handling model, that guarantees either collective failure or life of all components in a distributed system. Known implementations of the actor system include Erlang, Scala and libcppa. Although Erlang was not designed as an actor system, it implements de-facto actors in form of processes, which are part of the language. `libcppa` on the other hand, is an actor library for `C++`.

The objective of this work is the integration of GPGPU programming into the actor library `libcppa`. We introduce OpenCL-enabled actors, which offer an easy way to deploy code to GPUs from a high abstraction layer. We do not only want to keep the benefits provided by `libcppa`, such as network-transparency, but offer new options, such as hardware transparency. This allows transparent message passing between actors, independent of their hardware platform.

This work is organized as follows. Section 2 gives an introduction to the context of the actor model and GPU computing. This includes the actor model and its implementation in Erlang and `libcppa`, as well as GPU computing in general and OpenCL specifically.
Subsequently, Section 3 discusses three different approaches for programming single in-

struction multiple data (SIMD) architectures.
Section 4 presents the design goals for the OpenCL-enabled actor and discusses the OpenCL API as well as the interfaces of classes surrounding the OpenCL-enabled actor. It concludes with the discussion of open questions.
Section 5 presents the implementation of the `actor_facade` and the function `spawn_cl`. Furthermore, this section explains our implementation of the key concepts previously discussed in Section 4.
To evaluate our implementation, we tested the OpenCL-enabled actor implementation against the native library. Section 6 presents the results and shows that our implementations induces only a small amount of overhead.
Finally, Section 7 draws conclusions and gives an outlook for future work.

2 Actors and GPU Computing

This section discusses the general concepts of actors and GPU computing. In the first section, we introduce Erlang and `libcppa`. Later we discusses GPU architecture and OpenCL, a framework for GPU computing.

2.1 The Actor Model

In a historic perspective, extending message passing by adding error handling capabilities for distributed systems lead to the actor model. It was first specified by Hewitt et al. in 1973 (Hewitt et al. (1973)). 13 years later Agha continued to work out the theoretical aspects of the actor model in his dissertation (Agha (1986)). At the same time, Armstrong took a more practical approach by developing Erlang (Armstrong (2007). Although Erlang does not mention the actor model, its processes are a de-facto actor implementation.

Actors are concurrent, isolated entities that interact via message passing (Agha (1986)). They can address each other using network transparent, unique identifiers. In response to a received message, an actor can send messages to other actors, spawn new actors, or change its behavior by exchanging the message handler for processing the next incoming message.

The actor model offers several benefits. Since actors can only interact via message passing, they neither share nor can corrupt each others state. Hence, race conditions are avoided by design. Actor creation is a lightweight operation and is used to distribute work. To handle errors in distributed systems, actors can monitor each other. If an actors dies unexpectedly, the runtime environment sends a message to each actor monitoring it.

2.1.1 Erlang

Erlang is a concurrent, dynamically typed programming language developed for programming large-scale, fault-tolerant systems (Armstrong (2003)). Though Erlang was not build with the actor model in mind, it satisfies its characteristics. Each process in Erlang is in fact an actor with the characteristics described in Section 2.1. New processes are created by a function called `spawn`. Their communication is based on asynchronous message passing. Processes use pattern matching to identify incoming messages.

Processes implement error handling in form of monitors. When a monitored actor terminates, the runtime environment sends a “down” message containing the reason to all actors monitoring it. A stronger coupling is created by a link. A link is a bidirectional monitor, that causes the runtime to send “exit” messages instead. The default reaction to an abnormal exit message causes the receiving actor to terminate with the same reason. This can be overridden to receive and handle exit messages like other messages.

These characteristics originate from the goal to build fault-tolerant systems. If a process experiences an error, it should not corrupt other processes in the same system. This goal is achieved by isolating processes. Distributed systems can use redundancy to protect themselves against machine failure and allow other machines to continue work and correct errors. Erlang implements non-local error handling to achieve this goal.

2.1.2 `libcppa`

`libcppa`[1] is an actor library written in `C++11` (Charousset and Schmidt (2013)). It addresses concurrency and distribution by providing a message-oriented programming model. The API is designed in a style familiar to `C++` developers and provides a domain-specific language (DSL) for actor programming. To achieve good scalability on distributed systems, the creation and destruction of actors in `libcppa` is a lightweight operation. Instead of assigning a thread to each actor, which would rely on system calls and kernel resources, `libcppa` schedules actors in a cooperatively managed pool.

Actors are created using the function `spawn`. It takes a function or class as first argument and returns a handle to the created actor. The handle can be used to address it and provides a unique address in a distributed system. Actors can communicate via asynchronous or synchronous message passing, using the `send` or `sync_send` functions. Network transparency hides whether an actors runs on the same core, same system, or another machine in the network.

Messages are buffered at the receiver in order of arrival before they are processed. `libcppa` implements two types of actors. The default actor implementation is scheduled cooperatively. Hence, the use of blocking functions may starve other actors. A second type of actors, thread-mapped actors, can be used to avoid this conflict.

The behavior of an actor specifies its response to messages it receives. `libcppa` uses partial functions as message handlers, which are implemented using pattern matching. Messages that cannot be matched stay in the buffer until they are discarded manually or handled by another behavior. The behavior can be changed dynamically during message processing.

[1] `http://libcppa.org`

`libcppa` includes monitors and links (Charousset (2012)) to build fault tolerant distributed systems with the same semantics as Erlang.

The work of Charousset et al. (2013) compared `libcppa` to the actor implementations in Erlang and Scala. It measures (1) actor creation overhead, (2) sending and processing time of message passing implementations and (3) memory consumption for several use cases. The results show that `libcppa` performs best in the presented use cases.

2.2 GPU Computing

Graphic Processing Units (GPUs) are traditionally used to calculate high resolution graphic effects in real-time (Nickolls and Dally (2010)). To achieve high frame rates, GPUs are massively parallel. A routine written to calculate one pixel can be executed concurrently to calculate multiple pixels at once. Frameworks like OpenCL (Open Computing Language, Scarpino (2011)) or CUDA (Compute Unified Device Architecture, Kirk and Hwu (2013)) offer an API to use the available hardware for non-graphical applications that benefit from the amount of parallelism offered by the GPU. This approach is called GPGPU (general purpose GPU) computing.

Only algorithms that can be split into multiple independent tasks benefit from a high amount of parallelism. To make optimal use of GPGPU computing, it is often combined with calculations done on the CPU, a concept called heterogenous computing.

2.2.1 The Evolution of GPUs

The first graphic cards were build around a pipeline, where each stage offered a different fixed operation with configurable parameters (Lindholm et al. (2001)). Soon, the capabilities supported by the pipeline were neither complex nor general enough to keep up with the developing capabilities of shading and lighting effects. To adapt to the challenges, each pipeline stage evolved to allow individual programability and include an enhanced instruction set (Blythe (2006)). Although this was a major step towards the architecture in use today, the design still lacked mechanisms for load balancing. If one stage required more time than others, the other stages were left idle. Furthermore, the capacities of a stage were fixed and could not be shifted depending on the algorithm. To provided an overall better workload, the pipeline was replaced by data-parallel programable units (Owens et al. (2008)). All units share a memory area for synchronization, while in addition each unit has a local memory area only accessible by its own processing elements. A single unit only supports data parallelism, but a cluster of them can process task parallel algorithms as well.

2.2.2 A Simplified GPU Architecture

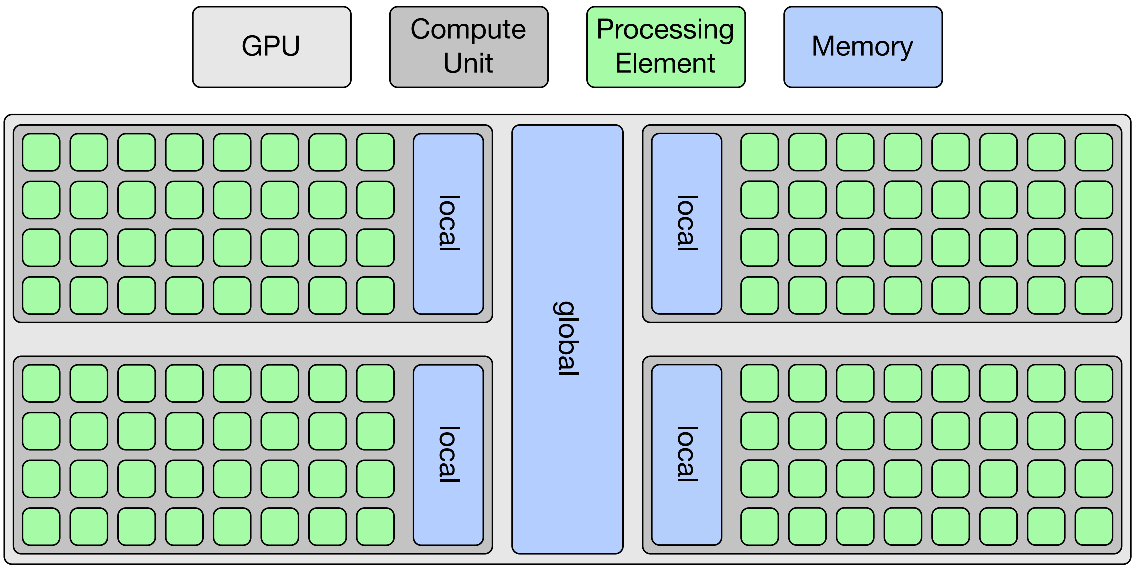


Figure 1: Simplified GPU architecture, showing a GPU with 4 compute units with 32 processing elements each.

In this section we briefly discuss GPU architecture and examine concurrency on a hardware level. Figure 1 shows a simplified architectural scheme for GPUs. Each GPU provides compute units (CU), its numbers ranging from two in laptops or desktop GPUs to more than ten in server GPUs. Each CU follows a data parallel model and consists of several processing elements (PE) and a local memory area. The PEs inside a CU follow an single instruction multiple data (SIMD) approach. They share one program counter and the local memory, but also include private memory as cache. At the same time, all CUs have read and write access to the global memory area. Other device, such as the CPU, use this area for data exchange.

It is worth noting that architecture presents the perspective of OpenCL and is a simplified view. A real GPU is much more complex (Hennessy and Patterson (2012)). For example, PEs are bundled into smaller groups within a CU. In addition, some components are omitted such as a scheduler that manages the executions done on the CUs. Instead, the figure aims at giving a basic insight into the different memory areas and the relation between processing elements and computing units.

2.3 OpenCL

There are two major frameworks for GPU computing. CUDA (Compute Unified Device Architecture, Kirk and Hwu (2013)) is a vendor-specific API developed by Nvidia. It grants access to all GPU-dependent features. OpenCL (Open Computing Language, Scarpino (2011)) is developed by the OpenCL Working Group, a subgroup of the Khronos Group. The Khronos Group is a non-profit organization that creates “open standards for the authoring and acceleration of parallel computing, graphics, dynamic media, computer vision and sensor processing on a wide variety of platforms and devices” (khr (2013)). Among others, it is known for the standardization of OpenGL and OpenML. OpenCL is used for parallel general purpose computing on a broad range of hardware. Universality is the key feature of OpenCL, but has the downside that it is not possible to exploit all hardware-dependent feature. The OpenCL framework includes an API and a cross-platform programming language called “`OpenCL C`” (Munshi (2012)). Since its first release in December 2008, it reached the stable version OpenCL 1.2. A provisional specification for version 2.0 is available as of June 2013.

A study by Fang et al. (2011) examines the performance differences between OpenCL and CUDA. Their benchmarks are divided into two categories. The first category consists of synthetic benchmarks, which measure peak performance and show similar results for OpenCL and CUDA. The second category includes real-world applications and shows a better overall performance for CUDA. However, the authors examine sources of the overhead and present an example that results in similar performance.

In this work, we chose OpenCL over CUDA, because it (1) provides the capability to program a variety of hardware, (2) is supported by a lot of vendors, including AMD and Nvidia and (3) is built from an open standard. The following sections will explain the programming model of OpenCL.

2.3.1 Program Structure

Each OpenCL program consists of two parts. One part runs on the host, normally a CPU, and is called host program. The other part is called “kernel” and runs on an OpenCL device such as a GPU. An OpenCL device is further divided into compute units (CU) and processing elements (PE), as shown in Figure 1. The kernel is written in `OpenCL C`, a subset of the `C` programming language with GPU specific extensions. Kernels are compiled at runtime by the host program.

The host program initializes data on the device, compiles kernels, and manages their execution. The OpenCL context is the central abstraction primitive and holds resources for devices, kernels and memory objects. A context is associated with a platform, representing the used device driver or platform, and the devices it is responsible for. The host program also manages a command queue to interact with devices. The number of command queues per context or device is not limited, though a command queue is associated with a single device.

OpenCL offers events to organize operations in command queues. They allow the definition of dependencies between operations from the same context and the use of callback functions. This enables an asynchronous workflow.

2.3.2 The Kernel Execution

Each kernel is executed in an N-dimensional index-space called “NDRange”. Derived from three dimensional graphic calculations, N can be either one, two or three. Each dimension has at least one item, even if it is not considered for a calculation. Figure 2 shows a sample index space. Each tuple (n_x, n_y, n_z) in the index space identifies a single kernel execution, called work-item. These tuples are called global IDs. They allow each work-item to be identified during the kernel execution and can be used to make them operate on different data. The total number of kernel executions can be calculated by multiplying the maximum range of each dimension $(N_x * N_y * N_z)$.

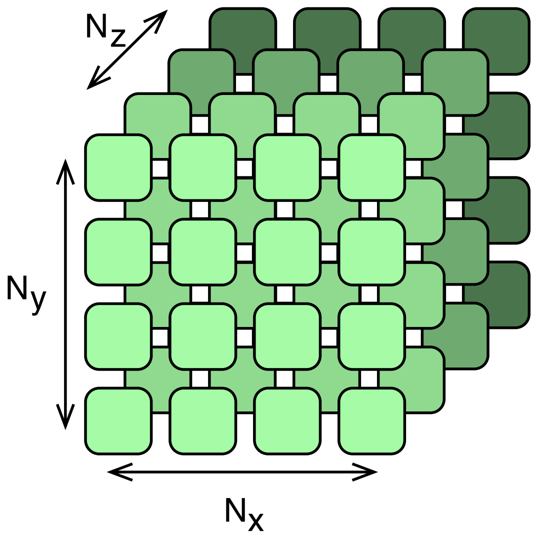


Figure 2: An index-space with sample size 4.

Work-items can be organized in so-called work-groups. Each work-group has an ID and each work-item has a secondary ID in its work-group, called local ID. This allows each work-item to be identified in one of two ways: either by its global ID or its local ID combined with the work-group ID. The number of work-items per work-group cannot exceed the number of processing elements in a compute unit. Similar to the global index space, the local ID can be distributed in up to three dimensions. All items in a work-group run in parallel on a single CU. Depending on the available CUs, work-groups may run sequentially or in parallel.

A work-group with a maximum of 1024 work-items, for example, could be divided into a work-group with 1024 work-items in the first dimension and one in both other dimensions (1024, 1, 1), for a total of 1024 (1024 * 1 * 1) work items. Alternatively, the work-items could be spread into two dimensions, 32 in the first dimension, 32 in the second and 1 in the third (32, 32, 1). This, again, sums up to 1024 work-items. A work-group is allowed to have fewer work-items than the maximum.

2.3.3 The Memory Model

A GPU has four distinct areas of memory, which differ in location, speed and accessibility. `OpenCL C` uses keywords called “address space modifiers” to assign an area to a variable.

Global memory is addressed by the `__global` keyword and usable by all work-items, but has a slow access speed. It can also be initialized by the host program.

Constant memory is addressed by the `__constant` keyword and a region of the global memory with identical properties, except that it only offers read access to the work-items. The host program has read as well as write access.

Local memory is addressed by the `__local` keyword and owned by a work-group and offers read and write access to all corresponding work-items. Access is about a hundred times faster compared to the global memory (Scarpino (2011)). Additionally the host program has no access to this memory region.

Private memory is addressed by the `__private` keyword and the default memory region. It is a small memory area owned by a single work-item and does not permit read or write access by any other entity, including the host program. Unlike the local memory it has not to be shared with other work-items. Hence access has not to be synchronized.

Figure 1 shows the global and local memory areas in blue. The private memory belongs to a PE. The constant memory is part of the global memory.

2.3.4 The OpenCL Workflow

The following steps describe the workflow for a simple program. More complex programs can be built by using callbacks or multiple command queues.

1. Find a platform (`clGetPlatformIDs`). OpenCL devices are bound to a platform. On "OS X" this is Apple, while a Linux system with Nvidia hardware would show "Nvidia CUDA".
2. Find a device (`clGetDeviceIDs`). This can be done independently for each device type, such as GPU or CPU.
3. Create a context (`clCreateContext`). A context is created to manage a bundle of devices and its associated resources.
4. Create a command queue (`clCreateCommandQueue`). Each command queue is linked to one device, but more than one queue can be created for a device. It is required to interact with the device. Operations can be synchronized between multiple queues from the same context by using events.
5. Create a program object (`clCreateProgramWithSource`). A program object contains source code for kernels and functions in use. It is associated with a context.
6. Compile the programs (`clBuildProgram`). This call compiles the code for a list of devices passed as an argument or all devices in the context of the program if called without a list. It fails if the code contains errors. A build log can then be obtained with the `clGetProgramBuildInfo` function.
7. Extract a kernel (`clCreateKernel`). Kernels must be encapsulated in a `cl_kernel` when passed to the `command_queue`. The name of the kernel function is used to select it from a program object.
8. Create memory objects (`clCreateBuffer`). Each memory object belongs to a context. When it is created, access modifiers for the work-items can be specified. Additionally three flags are available to decide where the memory is allocated. Its size must also be set when it is created.
9. Initialize memory objects (`clEnqueueWriteBuffer`). When a memory object is not initialized when it is created, this function can be used to do so. It can be executed asynchronously with an other command or a callback function linked to it via events.

10. Set memory objects as arguments (`clSetKernelArg`). Memory objects must be set as kernel arguments. The argument used for return values does also requires a memory object.

11. Enqueue kernel (`clEnqueueNDRangeKernel`). This function is used to trigger a kernel execution and to specify (1) the created work-items, (2) an offset for the global IDs and (3) the work-group sizes. The offset and the work-group sizes can be set to `nullptr`. While an unspecified offset defaults to zero, the work-group size will be calculated by OpenCL. This function always works asynchronously. Either `clFinish` can be used to wait for it, or a callback is set, or another command it linked to it via events.

12. Read results (`clEnqueueReadBuffer`). Used to read data from memory objects into arrays. Similar to `clEnqueueWriteBuffer` this can be done asynchronously.

3 SIMD Programming Concepts

Computer organizations can be classified in four categories, examining the amount of interaction between data and instruction stream (Flynn (1972)). In this context, a stream describes a sequence of operations.

Single Instruction, Single Data (SISD) This is the most conventional class. It includes single core machines, such as the traditional Von Neumann architecture.

Single Instruction, Multiple Data (SIMD) This category includes array processors, vector registers and GPUs. These architectures have multiple processing elements, that execute the same instruction on different data. Some CPU architectures include hardware extensions to perform simple SIMD instructions.

Multiple Instruction, Single Data (MISD) Different instructions are performed on the same data. In a broad view, pipeline architectures may be included into this category.

Multiple Instruction, Multiple Data (MIMD) This architecture describes multiple independent units consisting of instruction and processing elements. This classification does not further specify the organization of these units. Among others, multicore processors are included in this category.

3.1 The Connection Machine - A Historic SIMD Computer

The idea that led to the development of the Connection Machine (CM) was the desire to build a “thinking machine”. The concept was published by William Daniel Hillis (Hillis (1985)). The computers build at that time were single core computers with the Von Neumann architecture. Hillis started his analysis with a picture and the inability of a machine to process and capture its contents in the same way humans do. He argued that the problem is not the available speed but the inefficient use of hardware. Of all available transistors only a small percentage was busy, while the rest remained idle. To achieve a better utilization, Hillis suggested to build the Connection Machine, that consisted of a large number of parallel processing elements.

The requirements to the CM were motivated by use cases for image processing and path-length algorithms. Instead of processing the pixels in an image or the nodes in a graph sequentially, the machine should distribute the problem to a large number of processing units and run the algorithm concurrently. In addition, processing units should be able to build connections with other units to enable a fast data exchange. Furthermore, the connections should not be restricted to neighbors, but enabled among non-adjacent processing units. This would allow to represent images by connection neighbors or complex graphs with edges across the complete network. The connectivity itself should be transparent to the user on a software level.

The architecture of the CM consisted of a host processor and a cluster of parallel processing units. The prototype (CM-1) and the second version (CM-2) were SIMD machines. This approach was favored over a MIMD architecture because the distribution of separate instructions would be too expensive regarding memory bandwidth. Their successor, the CM-5, was built following a MIMD architecture. An unresolved problem was to distribute the instructions as quickly as the processing units could execute them. A microcontroller was used to translate high level instructions into a sequence of simpler ones for the processing units.

Hillis developed a language extension for Common Lisp, called Connection Machine Lisp (CmLisp), to make use of the architecture. The most interesting extensions are:

xector The core data structure in CmLisp is called xector (pronounced zek'tor). A xector is distributed on the processing units of the CM. Operations performed on a xector are performed concurrently on all elements. Each element consists of an index-value pair. In mathematical terms, a xector is a function from Lisp values to Lisp values.

Alpha Notation (α) CmLisp includes a new notation to describe “all-at-once” parallelism. The letter α can precede any value or expression to convert it into a constant xector. A xector of functions is evaluated by applying its function to all arguments concurrently. Indices that do not occur in all arguments are ignored. The operator “•” excludes values from the alpha notation.

Beta Reduction (β) A function expecting two arguments preceded by a β can be used to reduce a xector to a single value. This is performed in parallel and requires logarithmic time, with respect to the length of the xector. The indices of the elements are ignored.

3.2 MMX - A SIMD Instruction Set for CPUs

MMX was designed by Intel in 1996 (Peleg et al. (1997)). It was introduced to enhance the multimedia capabilities of CPUs by adding machine-level SIMD operations. The MMX instruction set includes operations for arithmetic, comparison, conversion, logical, shift, and data transfer tasks. In addition to new instructions, MMX introduces four new packed data types. These types have 64 bits length and bundle either eight bytes, four words, two doublewords or a single item. The wrapped values are interpreted as fixed point integers. MMX operations process all values of one packed type in parallel. Furthermore, the values have to be located in eight special registers that are mapped to registers of the floating point unit (FPU). Since over- or underflows are often undesired in multimedia applications, the instructions include a saturation arithmetic. On the occurrence of an overflow (underflow), the largest (smallest) value in range is selected.

3.3 Programming a GPU with OpenCL

While Sections 3.1 and 3.2 presented different approaches to general SIMD programming, this section is related specifically to GPU programming. Its first two parts discuss `OpenCL C`, the `C` dialect used to program OpenCL kernels, which offers SIMD operations to each work-item. The third part presents an program that uses the OpenCL to square a matrix.

3.3.1 Language Extension of `OpenCL C`

`OpenCL C` is based on the `C` programming language. More precisely, it is a subset of `C` with extensions for GPGPU programming. Among others, function pointers, variable-length arrays and variadic functions are not available.

The primitive types (`char`, `int`, `long` and `float`) and their unsigned variants are available as vector data types. These are defined by appending a literal n to the primitive type, where n is restricted to 2, 3, 4, 8 or 16. Vector types can be initialized using vector literals and passing elements of the same type equal to the size of the created vector. The following code shows 3 example vectors:

```
float4 a = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
uint4  b = (uint4)(1); // creates (1, 1, 1, 1)
long4  c = (long4)((long2)(1L, 2L), 3L, 4L);
```

Elements in a vector are accessible in two ways. In all vectors with up to four elements, the elements can be addressed via the letters `x`, `y`, `z` and `w`. The first element is addressed by `x`, the second by `y`, and so on. This can be used to access the elements for both read and write access. An element can only be written once per expression, but read multiple times.

```
float4 d;
d.y = 2.0f;
d.xzw = (float3)(1.0f, 3.0f, 4.0f);
// creates (1.0, 2.0, 3.0, 4.0)
```

```
float8 e = (float8)(d.xx, d.yy, d.xy, d,wz);
// creates (1.0, 1.0, 2.0, 2.0, 1.0, 2.0, 4.0, 3.0)
```

The second way to address the elements is via their indices, using `s` plus indices from `0` to `f` (hex for 15). For an `int4 i`, the notation `i.s0` would access the first element and `i.s3` the last one. Thought the notations cannot be mixed, more than one index can be appended to the `s`.

```
int4 f, g;
f = (int4)(1,2,3,4);
g.s0123 = f.wzyx;
// creates (4,3,2,1)
```

Vectors provide a syntax to access their upper or lower half, `.hi` and `.lo`. Even or odd indices can be accessed by using `.even` or `.odd`. It is not allowed to take the address of an vector element.

Arithmetic operands can be either two scalars, two vector types or a scalar and a vector type. If both types are scalars, the operation will return a scalar as well. If both types are vectors, the result will be a vector of same size and the operation is applied once per element pair. If only one is a scalar, it will be widened to a vector of equal size before the operation is applied. Operations with other operands work in the same way. If the arguments are not of the same type, a conversion may be possible. OpenCL has specific rules for type casts and offers explicit conversion functions for some cases. A complete list of conversions rules is included in the OpenCL specification, Munshi (2012).

```
int4 h, i;
int  k;

h = i + k;
// is equivalent to
h.x = i.x + k;
h.y = i.y + k;
h.z = i.z + k;
h.w = i.w + k;
```

Derived from graphics calculations, OpenCL supports image types. These come in 1, 2 or 3 dimensional variants. Currently, our work does not support these types as kernel arguments.

Address space qualifiers determine in which memory area a variable is placed. The memory areas and qualifiers are discussed in Section 2.3.3. Variables are placed in private memory if no qualifier is used.

Two function qualifiers can be used to mark functions for the compiler. One is the `__kernel` or `kernel` qualifier, required to execute the function on an OpenCL device. A kernel can be called by other kernels as a normal function. A kernel function always returns `void` and uses call by reference to return values. The other function qualifier is `__attribute__`. It gives hints to the compiler for better optimizations, such as work-item distribution.

3.3.2 Built-in Functions of `OpenCL C`

Each work-item has to be identifiable at runtime to make use of the work-item distribution. Each ID consists of a tuple with one value per dimension. The values range from 0 to the total number of work-items in the dimension. If a dimension is not used, the global ID for that dimension is 0 for all elements. The function `get_work_dim` returns the number of dimensions in use.

OpenCL offers functions to identify work-items by their global ID. They require an argument to select a dimension from the tuple. `get_global_size` returns the current maximum number of work-items in a dimension. `get_global_id` return the ID itself. It must be called once for each dimension to get the complete ID. The offset for the global IDs can be obtained by calling `get_global_offset`.

The functions to manage work-groups are similar and return local and group ID. The function `get_local_size` returns the maximum values for the local IDs and `get_num_groups` the maximums for the group ids. The IDs are obtained by `get_local_id` and `get_group_id`, respectively.

Furthermore, functions for math, geometrics, integer and float operations are available. Most of them work on scalars as well as vectors.

OpenCL offers barriers to synchronize work-items. A `barrier` function must be passed by all work-items in a work-group prior to continuation. It ensures a correct ordering of memory operations, and is called memory fence. A flag specifies if the memory operations synchronize for global or local memory. A memory fence can be used without the barrier to ensure either loads (`read_mem_fence`), stores (`write_mem_fence`) or both (`mem_fence`) are ordered.

3.3.3 An OpenCL Use Case Scenario

This section shows the multiplication of quadratic matrices in OpenCL. A straightforward algorithm calculates each index by multiplying and adding the corresponding line and column. OpenCL speeds up the process by calculating many indices concurrently. For an optimal distribution, we want to create a number of work-items equal to the number of indices in the matrix. For an easy mapping, our index space will have two dimensions to make each matrix element identifiable by a tuple (n_x, n_y) in the index space. For better readability, error handling is omitted in the presented code.

The following code shows the implementation of the kernel. It is written in `OpenCL C` and stored in the host program as a string. The kernel itself is a function marked with the `__kernel` keyword. Input and output matrices are arguments placed in the global memory. They are represented as one-dimensional arrays, because OpenCL does not allow multidimensional arrays as arguments.

```
 1  constexpr const char* kernel_source = R"__(
 2      __kernel void matrix_mult(__global float* matrix1,
 3                                __global float* matrix2,
 4                                __global float* output) {
 5          size_t size = get_global_size(0); // Nx == Ny
 6          size_t x = get_global_id(0);      // nx
 7          size_t y = get_global_id(1);      // ny
 8          float result = 0;
 9          for (size_t idx = 0; idx < size; ++idx) {
10              result += matrix1[idx + y * size]
11                      * matrix2[x + idx * size];
12          }
13          output[x+y*size] = result;
14      }
15  )__";
16  constexpr size_t kernel_size = strlen(kernel_source);
17  constexpr const char* kernel_name = "matrix_mult";
18  constexpr size_t      matrix_size = 1000;
```

In line 5, the function `get_global_size` returns the number of items in the first dimension. Since we use quadratic matrices, the number of items in both dimension are equal and correspond to the number of rows or columns. The function `get_global_id` is used to get the global id of the specific work-item. This ID is a tuple (n_x, n_y) and used to determine the index each work-item calculates in the matrix. The calculation is performed in a for-loop. In addition to the kernel, we also define the length of the kernel source, the kernel function name and the size of the matrices. These information are required at runtime, when the kernel is compiled and enqueued for execution. They do not have a fixed size.

The code shown below is placed in the main function of the host program. The fist step in an OpenCL program is to find a platform and its OpenCL devices. The platform depends on the OS and device drivers. On a MacBook, we found “Apple” as platform, while our Linux setup returned “Nvidia CUDA”. The `clGetPlatformIDs` and `clGetDeviceIDs` follow a similar pattern. If no container is passed to the functions, they return the number of available IDs, which can be used to allocate a container to hold all IDs. This example always chooses the first ID, i.e., the first device.

```
cl_uint number_of_platforms = 0;
clGetPlatformIDs(0, nullptr, &number_of_platforms);
vector<cl_platform_id> platform_ids(number_of_platforms);
clGetPlatformIDs(number_of_platforms, platform_ids.data(),
                 nullptr);

cl_uint number_of_gpus = 0;
auto device_type = CL_DEVICE_TYPE_GPU;
clGetDeviceIDs(platform_ids[0], device_type, 0,
               nullptr, &number_of_gpus);
vector<cl_device_id> gpu_ids(number_of_gpus);
clGetDeviceIDs(platform_ids[0], device_type, number_of_gpus,
               gpu_ids.data(), nullptr);

cl_device_id device = gpu_ids[0];
```

In the next step, we create a context to manage devices and command queues. The context requires a list of all devices it manages, while the command queue expects a context and the device it is responsible for.

```
cl_context context = clCreateContext(0, device.size(), &device,
                                     nullptr, nullptr, nullptr);

cl_command_queue cmd_queue = clCreateCommandQueue(context,
                                                  device,
                                                  0, nullptr);
```

To compile the source code for the device and create a kernel, we have to call `clCreateProgramWithSource`. The function creates an OpenCL program object containing all sources. `clBuildProgam` then compiles the code for all devices associated with the context. In case it returns an error, the log can be obtained by using clGetProgramBuildInfo. Finally, a kernel has to be chosen from the program object by calling clCreateKernel, which takes a program object and a kernel name as arguments.

```
cl_program program = clCreateProgramWithSource(context, 1,
                                               &kernel_source,
                                               &kernel_size,
                                               nullptr);
clBuildProgram(program, 0, nullptr, nullptr, nullptr, nullptr);

cl_kernel kernel = clCreateKernel(program, kernel_name, nullptr);
```

Arguments are passed to a kernel as memory objects. Consistent with the arguments passed to the kernel presented above, we use a vector to represent the matrix. The iota function provided by the C++ standard template library (STL) fills the matrix with ascending values, starting at 0. In line 7, the size for our memory objects is calculated using the number of indices and data type. The clCreateBuffer function requires a context, access modifiers for the work-items, and the size of the create memory object. All created memory objects are of equal size, but the input should be read only and the output write only. Since the output values do not need to be initialized, only the input values are copied to the device using the clEnqueueWriteBuffer function. Setting the third argument to CL_TRUE causes the function to block. Alternatively, OpenCL offers a callback-based approach, that avoids blocking.

```
1  vector<float> matrix1(matrix_size * matrix_size);
2  vector<float> matrix2(matrix_size * matrix_size);
3
4  iota(begin(matrix1), end(matrix1), 0);
5  iota(begin(matrix2), end(matrix2), 0);
6
7  auto data_size = matrix_size * matrix_size * sizeof(float);
8  cl_mem buf_in1 = clCreateBuffer(context, CL_MEM_READ_ONLY,
9                                  data_size, nullptr, nullptr);
10 cl_mem buf_in2 = clCreateBuffer(context, CL_MEM_READ_ONLY,
11                                 data_size, nullptr, nullptr);
12 cl_mem buf_out = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
13                                 data_size, nullptr, nullptr);
14
15 clEnqueueWriteBuffer(cmd_queue, buf_in1, CL_TRUE,
16                      0, data_size, matrix1.data(),
17                      0, nullptr, nullptr);
18 clEnqueueWriteBuffer(cmd_queue, buf_in2, CL_TRUE,
19                      0, data_size, matrix2.data(),
20                      0, nullptr, nullptr);
```

The last step before enqueuing the kernel for execution is setting the memory objects as arguments. This is done by using the `clSetKernelArg` function and providing kernel, argument index, and buffer as arguments. The function `clEnqueueNDRangeKernel` is used to enqueue the kernel and specify the number of work-items created for its execution. The first argument is a command queue and identifies the device in use. It is followed by the kernel we want to execute. The next arguments indicate the number of work-items and their partitioning into work-groups. The third argument indicates the number of dimensions. The fourth argument is the offset of the global IDs, which we do not need in this example. The sixth argument are the work-items in each dimension. It is followed by the size of the work-groups, which can be set to `nullptr` to make OpenCL determine the size itself. The last three arguments can be used for event handling. In this example, we wait for all commands in the queue to finish with the `clFinish` function.

```
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*) &buf_in1);
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*) &buf_in2);
clSetKernelArg(kernel, 2, sizeof(cl_mem), (void*) &buf_out);

vector<size_t> index_space{matrix_size, matrix_size};
clEnqueueNDRangeKernel(cmd_queue, kernel,
                       index_space.size(),  // work dimensions
                       nullptr,             // global offset
                       index_space.data(),  // global work size
                       nullptr,             // local work size
                       0, nullptr, nullptr);// event handling
clFinish(cmd_queue);
```

The final step is to read the calculated matrix back from the GPU. This is done by `clEnqueueReadBuffer`, which behaves similar to `clEnqueueWriteBuffer` used before.

```
vector<float> output_data(matrix_size * matrix_size);
clEnqueueReadBuffer(cmd_queue, buf_out, CL_TRUE, 0,
                    data_size, output_data.data(),
                    0, nullptr, nullptr);
```

4 The Design of OpenCL-enabled Actors

Before taking a closer look at the newly introduced OpenCL actors, we want to summarize the general design goals and review a few problems that emerged during the integration of OpenCL into `C++` and specifically `libcppa`.

4.1 Design Goals

This work adds OpenCL-enabled actors to `libcppa`. We aim at offering a high level of abstraction when using actors combined with OpenCL. This extension should be integrated into the existing runtime environment and shall benefit from the features provided by libcppa.

High-level Abstraction The OpenCL API is a low-level API, whereas `libcppa` offers a high level of abstraction. To ease the combination, this work aims to embed GPU programming concepts into an actor system and help the programmer remain at a high abstraction level.

The setup required to run a program in OpenCL includes several steps. Although the integration into `C++` has been improved by an official wrapper, many of the required steps have to be repeated in every program. This can be improved by transferring the setup process to the background and only leave the relevant decisions to the user such as the distribution of the kernel.

Seamless Integration into `libcppa` The OpenCL actors should integrate into the runtime environment of `libcppa` and benefit from existing properties, such as integration into the message passing system and network transparency. However, we want to keep a clean separation to the core library. The features presented here have to be enabled by a flag at compile-time. Furthermore, OpenCL-specific code is kept in a separate namespace, called `cppa::opencl`.

Access Transparency We want to examine this goal from three perspectives: (1) message passing, (2) actor creation and (3) error handling.

An OpenCL-enabled actor should be addressable transparently in a distributed system as well as on heterogeneous hardware. Network transparency is a feature provided by `libcppa`. It makes remote actors addressable in the same way as local actors and works for OpenCL-enabled actors as well. The abstraction over heterogeneous hardware allows transparent message passing to actors, regardless whether they are running on the CPU or GPU.

Actors are created by a `spawn` function. A transparent `spawn` function allows the creation of OpenCL-enabled actors with the same syntax used to create other actors.

`libcppa` implements error handling in form of monitors and links. We aim to integrate OpenCL-enabled actors into the error handling model and thus enable the transparent creation of links and monitors.

Location Transparency Creating an OpenCL actor should work in the same way regardless of the available hardware. A created actor should run on the CPU if no GPU is available or even be created on a remote systems that provides suitable hardware.

4.2 Working with the OpenCL API

The OpenCL API provided by the Khronos Group is written in `C`. In this work, we use the `C` API directly. A `C++` wrapper is available, but shows inconsistencies, as some API calls throw exceptions, while others still return error codes.

4.2.1 Datatypes from OpenCL

The OpenCL API uses self-defined types. This includes a number of type aliases that rename primitive types from the `stdint.h` header such as `cl_uint` or `cl_long`.

New data types introduced by OpenCL include `cl_command_queue` and `cl_kernel`. These types are created by API calls and require memory management. All OpenCL types use a reference count, which has to be updated manually. To avoid memory leaks, we manage these types using Resource Acquisition Is Initialization (RAII) containers called `smart_ptr`.

Our smart pointer class requires three template arguments: (1) the type it handles, (2) one function to increment the reference count, and (3) one to decrement it. Typedefs allow to use this class for all reference counted types OpenCL offers.

```
template<typename T, cl_int (*ref)(T), cl_int (*deref)(T)>
class smart_ptr { ... };

typedef smart_ptr<cl_kernel,
                  clRetainKernel,
                  clReleaseKernel> kernel_ptr;
```

4.2.2 Error Handling

`libcppa` uses logging functions of several levels to enable debug output on demand. Errors are thrown as exceptions with meaningful messages to help users debugging. On the other hand, the OpenCL API uses `C` style error values, which have to be checked manually after each function. In addition, OpenCL does not offer a function to convert an error value to an expressive string representation. For better compliance, all error handling in this work uses the existing logging functions and exceptions from `libcppa`. For a better workflow, we created a function to convert error values to strings.

4.2.3 The OpenCL Setup

Most steps in the OpenCL setup are repeated in a similar way in every program. This includes all steps from platform setup to the creation of command queues, as shown in Section 2.3.4. We use a singleton, called `opencl_metainfo`, to perform the discovery of platforms and devices and command queue creation. Furthermore, it obtains basic information about each device such as the maximum work-group size. Other operations access these information through this class.

4.3 Combining Actors with OpenCL

We introduce OpenCL-enabled actors to incorporate GPU computing into `libcppa`. An OpenCL actor can only interact via message passing. The class `actor_facade` is derived from `actor` and implements a message-passing facade that encapsulates an OpenCL kernel. It checks each received message against the signature of the kernel and, if compliant, executes a kernel using the message data as arguments. Afterwards, it sends the result back to the actor that sent the request. By inheriting from `actor`, the `actor_facade` is integrated into the `libcppa` runtime and benefits from features such as network transparency and error handling via monitors and links. Due to the limitations of OpenCL it is not possible to change the behavior of an actor that runs on an OpenCL device. Likewise it is not possible to spawn new actors from a kernel.

Figure 3 shows the components involved in a kernel execution. When an OpenCL-enabled actor receives a message delivered by the runtime environment, it extracts the data from the message and uses it to create memory objects for the arguments and a buffer of the results. It has access to the command queue of an associated device through the `opencl_metainfo`.

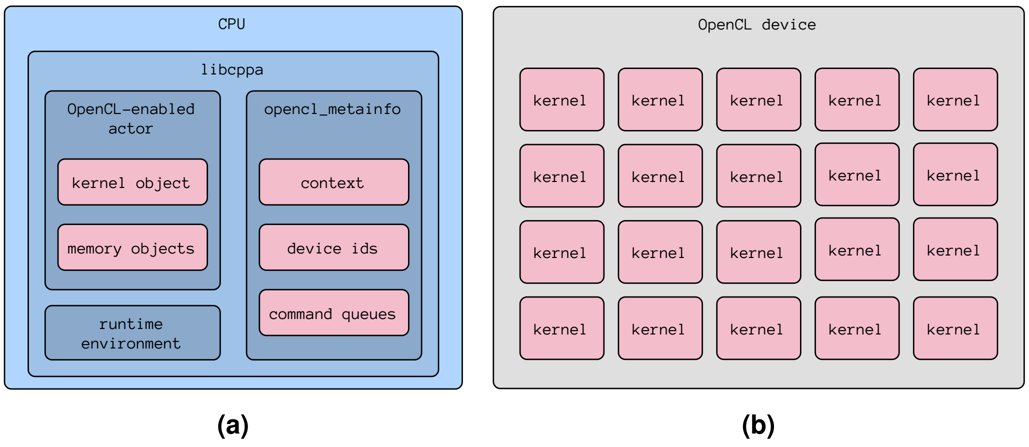


Figure 3: (a) Ownership of OpenCL resources in `libcppa`. The actor stores the kernel and memory and the metainfo stores context, devices and command queues. (b) Each work-item on the OpenCL device runs an own instance of the kernel.

Via the command queue, the kernel is sent to the GPU and the execution is triggered. The kernel is executed concurrently on the OpenCL device. Once the calculation is done, the data is transferred back to the device through the command queue. The actor proceeds to send the results back to the actor that request the calculation.

OpenCL allows to use GPUs as well as CPUs for computation. This suggests to run OpenCL actors on the CPU, if no GPU is available. Conceptually this is possible, but not all device drivers support OpenCL, and even if drivers for both device types are available, both libraries would have to be linked to the program. To manage multiple OpenCL implementations, the Khronos group offers a loader, called Installable Client Driver (ICD). At the current state of work, we did not take a closer look at this feature. Another problem to consider is the workload caused by an OpenCL actor running on the CPU. It is not scheduled with other actors, but uses the same resources.

4.4 Creating OpenCL-enabled Actors

The `spawn_cl` function creates OpenCL-enabled actors. The first two arguments are the code and function name of the kernel, both represented as strings. The following three arguments specify the distribution of the work-items. The third argument is the size of the global dimensions. Followed by the offset for the global IDs, which can be omitted and defaults to zero. The last argument is the size for the local dimensions. It will be calculated by OpenCL, if it is omitted. To match incoming messages to the arguments, the actor requires the signature of the kernel as a temple argument. By convention, the last kernel argument is an output argument, because a kernel function always has the return type `void` and thus cannot return its result.

```
template<typename Signature, typename... Ts>
inline actor_ptr spawn_cl(const char* source,
                          const char* fname,
                          const opencl::dim_vec& dims,
                          const opencl::dim_vec& offset = {},
                          const opencl::dim_vec& local_dims = {});
```

Kernels used to create OpenCL-enabled actors expect their arguments to be arrays placed in the global memory. Hence, messages send to OpenCL actors have to use data types such as `std::vector` that grant access to the underlying memory region. To allow user-defined types, we offer a second `spawn_cl` with two additional arguments: a function to map a custom message type to the arguments passed to the kernel and a second function to map the result type from the kernel back to a custom type sent back in the result message. If this `spawn_cl` variant is used, no template argument needs to be provided, since the kernel signature can be deduced from the function signatures.

```
template<typename MapArgs, typename MapResult>
inline actor_ptr spawn_cl(const char* source,
                          const char* fname,
                          MapArgs map_args,
                          MapResult map_result,
                          const opencl::dim_vec& dims,
                          const opencl::dim_vec& offset = {},
                          const opencl::dim_vec& local_dims = {});
```

4.4.1 A Wrapper For OpenCL Programs

The kernel source code can contain multiple kernels. Kernels in the same source code can share functions and data. Furthermore, they are compiled together. We offer the class `program` to wrap multiple kernels. Both `spawn_cl` functions are overloaded to accept a `program` instead of a string representation of the kernel. A `program` can be passed to multiple actors but compiles the code only once.

```
template<typename Signature, typename... Ts>
inline actor_ptr spawn_cl(const opencl::program& prog,
                          const char* fname,
                          const opencl::dim_vec& dims,
                          const opencl::dim_vec& offset = {},
                          const opencl::dim_vec& local_dims = {});
```

4.5 A Use Case with an OpenCL-enabled Actor

Based on the OpenCL example in Section 3.3.3, we shows the multiplication of two quadratic matrices using an OpenCL-enabled actor. All code snippets use the `std` and `cppa` namespace. The kernel code is did not change.

```
 1 void multiplier() {
 2     vector<float> m1(matrix_size * matrix_size);
 3     vector<float> m2(matrix_size * matrix_size);
 4     iota(m1.begin(), m1.end(), 0);
 5     iota(m2.begin(), m2.end(), 0);
 6
 7     auto worker =
 8         spawn_cl<float*(float*,float*)>(kernel_source,
 9                                         kernel_name,
10                                         {matrix_size,
11                                          matrix_size});
12
13     send(worker, move(m1), move(m2));
14     become(
15         on_arg_match >> [=](const vector<float>& result) {
16             cout << "done" << endl;
17             print_as_matrix(result);
18             self->quit();
19         }
20     );
21 }
```

The code above shows the function `multiplier` that is spawned as an actor in the main function. It creates two input matrices, starts an OpenCL-enabled actor, triggers the calculation and prints the result. The first four lines initialize two matrices with ascending values starting with zero. Lines 7 to 10 spawn the actor. The template argument represents the kernels signature: it requires two `float*` arguments and returns `float*` as a result. This example passes the kernel source directly to the `spawn_cl` and does not provide mapping functions. The arguments for the `spawn_cl` function are the kernel source, its name and the global dimension sizes. To trigger the kernel execution the matrices are send to the actor using the `send` function. Afterwards, the actor changes its behavior to await and print the result matrix.

```
1 int main() {
2     announce<vector<float>>();
3     spawn(multiplier);
4     await_all_others_done();
```

```
    shutdown();
}
```

The following code spawns an actor with the `multiplier` function and waits for it to finish. The `announce` call in line 2 is required by `libcppa` to enable serialization of `vectors` containing `float` values.

4.6 Discussion about our Design

The design goals list access and location transparency. However, we did not implement all their aspects. We divide access transparency into three parts. Transparent message passing and error handling were achieved in our implementation. The OpenCL-enabled actor implements the interface `actor` provided by `libcppa` and thus is integrated into the runtime environment of `libcppa`. It can be addressed by the `send` function in the same way as other actors and can be linked and monitored by other actors.
The third part of access transparency concerns actor creation and we did not achieve it. The OpenCL-enabled actors requires OpenCL specific parameters to be created, such as a kernel written in `OpenCL C` and information how to distribute the kernel on a device. Hence, we named the function to create OpenCL-enabled actors `spawn_cl`.
Actor creation includes the aspect of location transparency. OpenCL is available for GPUs as well as CPUs. Hence, on a local machine it is be possible to create OpenCL-enabled actors for both architectures. However, OpenCL depends on vendor specific libraries. Most often CPU and GPU do not share a vendor. Thus, they are addressed by different libraries. The Khronos Group provides an extension called Installable Client Driver (ICD) that allows the coexistence of multiple OpenCL libraries. Though the evaluation of ICD is left for future work, is suggests that location transparency on local machines can be achieved for actor creation.
Location transparency also includes actor creation on remote hosts. Since the OpenCL kernel is available as a sting it is possible to send it to a remote system and use it to create an OpenCL-enabled actor. However, the integration into `libcppa` is left for future work.

The context change to an OpenCL device comes with limitations. A key feature of actors is the ability to spawn new actors. A kernel can neither spawn new actors nor trigger new kernel executions. It can only call other kernels as a function. Furthermore, it is not possible to send a message from within a kernel or trigger message passing in the runtime environment of `libcppa`. The communication between the context of OpenCL and `libcppa` is limited to the interface provided by OpenCL.

5 Implementation

This section presents our implementation of the OpenCL-enabled actor. The first Sections briefly discuss concepts and patterns we have used throughout this work. This includes error handling (5.1), factories (5.2), smart pointers (5.3) and singletons (5.4). Section 5.5 then presents the classes we have use to manage the initialization and data required to run OpenCL. Finally, Section 5.6 discusses the implementation of the class `actor_facade` and the function `spawn_cl`.

5.1 Logging & Exceptions

`libcppa` offers logging machanisms through the `logging.hpp` header. There are five different levels: error, warning, info, debug and trace. The level is chosen at compile time. Using the logging functions from `libcppa` ensures, that all messages are kept in one place and not spread over multiple logs.

Besides logging, an exception is thrown whenever an error occurs. We use exceptions from the Standard Template Library (STL). When an essential process fails, a `std::logic_error` is thrown. This includes the platform discovery. For non-essential errors, we use the `std::runtime_error`.

5.2 The Factory Pattern

Some of our classes include an error-prone initialization. We uses a factory function to check invariants before the invocation of the constructor. To prevent the use of the constructor, it is declared private. Instead, a static `create` function is available. It takes the arguments required to call the constructor and invokes it only if the passed arguments fulfill the requirements for a successful object creation. If the input is not valid, an exception is thrown and the object is never created. Otherwise, the constructor can be called safely.

5.3 Smart Pointers with Reference Count

This section presents two smart pointer implementations, both depending on internal reference counting. The first part discusses the `intrusive_ptr` implemented in `libcppa`. The second part introduces the class `smart_ptr`, used to manage OpenCL data types.

5.3.1 The `intrusive_ptr` from `libcppa`

In `libcppa` garbage-collected types inherit `ref_counted`, which provides internal reference counting. The counter is thread safe and implemented as an `std::atomic<size_t>`. Besides functions to increment (`ref`) and decrement (`deref`) the reference count, the class provides functions to acquire the number of reference and to test whether the object has exactly one reference.

In addition to the base class, `libcppa` implements the class `intrusive_ptr`. It provides RAII containers for classes derived from `ref_counted`.

5.3.2 A General Apporach: `smart_ptr`

OpenCL types are reference counted. However, the process is not automated and has to be managed manually. Furthermore, they do neither meet the requirements to be managed by the `intrusive_ptr` of `libcppa`, nor can we modify the implementation of OpenCL. Hence, we implemented a new smart pointer class to automate management of the reference counted OpenCL types, in the same way the `intrusive_ptr` provides garbage-collection in `libcppa`.

The class `smart_ptr` can be used for all types that include an internal reference count and provide functions to manage it. The class expects three template arguments. The first argument defines the type. The second argument is a function to increment the reference count. The last argument is a function to decrement it.

```
template<typename T, cl_int (*ref)(T), cl_int (*deref)(T)>
class smart_ptr { /* implementation */ };
```

In addition to a default, copy and move constructor, we implements an `adopt` function. The function `adopt` transfers ownership, but does not increment the reference count. OpenCL objects are created with a reference count of one, which should not be incremented a second time, when the pointer is passed to a `smart_ptr`.

```
inline void adopt(pointer ptr) {
    reset(); // discards previous pointer
    m_ptr = ptr;
}
```

To improve readability and usability of the `smart_ptr`, we provide typedefs for all OpenCL data types. The following code shows an example with the `cl_kernel` type. The function `clRetainKernel` increments the reference count, while `clReleaseKernel` decrements it.

```
typedef smart_ptr<cl_kernel,
                  clRetainKernel,
                  clReleaseKernel> kernel_ptr;
```

5.4 The Singleton Interface of `libcppa`

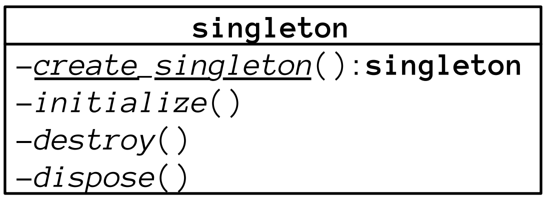


Figure 4: This diagram shows the singleton interface of `libcppa`.

A singleton is a class that will be instantiated only once. `libcppa` provides a class called `singleton_manager`, that aggregates access to all singletons in `libcppa` and handles their creation. To be manageable, a singleton has to implement the interface shown in Figure 4.

`create_singleton` This static function creates a new object of the class and calls the constructor. It should be a lightweight operation, since it may be called multiple time, explained below.

`initialize` It is called after `create_singleton`. If another thread created a valid singleton by now, `initialize` is not called again. It may be executed multiple times.

`dispose` Is used to delete uninitialized instances.

`destroy` Is used to delete initialized instances.

Whenever a singleton is accessed through the `singleton_manager`, the function `lazy_get` is called. It ensures that each concurrent call returns the same instance. If the singleton was not created previously, it is created and initialized implicitly. Although multiple temporary instances of the singleton may be created when `lazy_get` is called concurrently, only one is kept and all others are destroyed. This can only happen the first time the singleton is accessed. Once the pointer is set, access to singleton is a lightweight operation. The pointer stored in the manager is checked twice to reduce the `initialize` calls to a minimum.

5.5 Handling OpenCL metadata

We handle to OpenCL setup in the background and store information and resources in a singleton. It is used by all actors to access OpenCL resources, such as command queues. The first part of this section discusses the singleton class `opencl_metainfo` and the second part presents the class `device_info` that is used to store device specific information.

5.5.1 The `opencl_metainfo` Singleton

The class `opencl_metainfo` is responsible for the OpenCL setup. It is a singleton and available via `get_opencl_metainfo` through the singleton manager of `libcppa`, discussed in section 5.4. If `libcppa` is complied without OpenCL support, accessing the singleton throws an exception.

```
opencl::opencl_metainfo* singleton_manager::get_opencl_metainfo() {
#   ifdef CPPA_OPENCL // defined if libcppa is compiled with OpenCL
    return lazy_get(s_opencl_metainfo);
#   else
    CPPA_LOGF_ERROR("libcppa was compiled"
                    "without OpenCL support");
    throw std::logic_error("libcppa was compiled"
                           "without OpenCL support");
#   endif
}
```

The `initialize` function, required by the `singleton_manager`, performs the setup for the OpenCL platform and devices. An OpenCL platform is a vendor specific OpenCL driver. Since we have not examined the use of OpenCL with multiple platforms, per default the first detected platform is selected. Because this work aims at GPU computing, the setup prefers GPU over CPU devices. The code below shows the device discovery. Line 3 sets the device type to GPUs. The following function `clGetDeviceIDs` is used to get the number of available devices for the chosen type. If no devices are found, the flag is changed to CPUs in line 13. If no CPUs are found either, an exception is thrown in line 23. Once the number of available device is known, `clGetDeviceIDs` can be used to acquire their IDs, line 28.

```
 1  vector<cl_platform_id> ids = ... // available platforms ids
 2  int pid{0}; // index of the selected platform
 3  cl_uint num_devices{0}; // number of available devices
 4  cl_device_type dev_type{CL_DEVICE_TYPE_GPU};
 5
 6  // get number of available GPUs
 7  auto err = clGetDeviceIDs(ids[pid], dev_type,
 8                            0, nullptr, &num_devices);
 9
10  // if no devices are found, look for CPUs
11  if (err == CL_DEVICE_NOT_FOUND) {
12      CPPA_LOG_TRACE("No GPU devices found."
13                     "Looking for CPU devices.");
14      dev_type = CL_DEVICE_TYPE_CPU;
15      err = clGetDeviceIDs(ids[pid], dev_type,
16                           0, nullptr, &num_devices);
17  }
18
19  // if no devices are found, throw exception
20  if (err != CL_SUCCESS) {
21      ostringstream oss;
22      oss << "clGetDeviceIDs: " << get_opencl_error(err);
23      CPPA_LOG_ERROR(oss.str());
24      throw runtime_error(oss.str());
25  }
26
27  // get device ids
28  vector<cl_device_id> devices(num_devices);
29  err = clGetDeviceIDs(ids[pid], dev_type, num_devices,
30                       devices.data(), nullptr);
```

A single context is created to manage all devices. This enables the synchronization of operations over command queues. Once the context is created, a loop initializes a command queue for each device and stores it in a `device_info`. OpenCL offers the function `clGetDeviceInfo` to get device information, which we use acquire additional information for the `device_info`, explained in Section 5.5.2.

5.5.2 A Container for OpenCL Devices

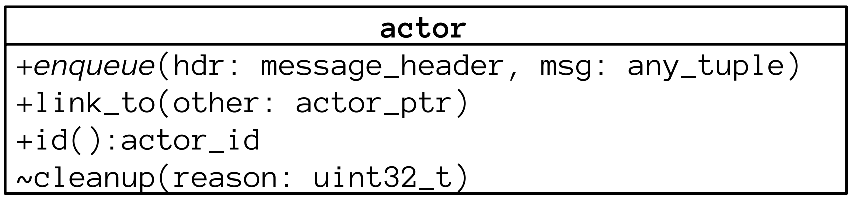
The class `device_info` stores information about an OpenCL device. The class is used by `opencl_metainfo` to manage available devices on a platform. A list with all devices is available to the user. Currently, the class `device_info` stores information about the maximum work-group size, the maximum number of dimensions and the maximum work-item size. Furthermore, device ID and a command queue managing the corresponding device are stored.

All members are declared private, because the OpenCL device id and command queue should no be publicly available. Getters offer access to the information dedicated to users, such as the work-group size.

5.6 OpenCL-enabled Actors

This Section describes the implementation of OpenCL-enabled actors. The first part presents the `actor` interface. It is the base class for all actor implementations in `libcppa`. Subsequently, we discuss a wrapper for OpenCL kernels, the class `program`. It can be passed to the function `spawn_cl` to create OpenCL-enabled actors. Furthermore, we will present our approach by highlighting important implementation details.

5.6.1 The `actor` Interface from `libcppa`



```
actor
+enqueue(hdr: message_header, msg: any_tuple)
+link_to(other: actor_ptr)
+id():actor_id
~cleanup(reason: uint32_t)
```

Figure 5: The actor interface of `libcppa`.

Figure 5 shows the interface of the class `actor`. It is the base class for all actor implementations in `libcppa`.

The member function `enqueue` is used to deliver a message to an actor. It has no default implementation and has to be implemented for each actor type individually. The first argument for enqueue is a message_header, which stores sender and receiver as well as a priority and a unique message ID. The second argument is the message data, i.e., a tuple.

The function link_to links the calling actor to the callee. When a linked actor exits, it sends a message with the exit reason to each linked actor. It has a default implementation for local, i.e., non-proxy, actors.

Each actor has a process-unique ID. It can be obtained by the function id. In combination with the process ID and node ID, it is a unique identifier in a distributed system.

Before an actor object gets destroyed, the member function cleanup is called. The default implementations sets an exit reason. It can be overridden to include additional clean up code.

5.6.2 A Wrapper for OpenCL Kernels

The OpenCL program object stores kernel code and is used to compile it for specified OpenCL devices. All kernels in a program are compile collectively and may share functions and data. Furthermore, a program object is required to create kernel objects, which can be passed to a command queue for execution.

We implemented the class program to wrap kernel source code and an OpenCL program object. A program instance can be passed to the function spawn_cl instead of a plain string representation of the code. The code in a program is compiled for a specified device. Multiple OpenCL actors can be created with the same program, but may use different kernels stored in it.

The constructor of the class program expects a context, a command queue and an OpenCL program object. The kernel is compiled at runtime when the program is created. The static member function create constructs new instances, it expects a kernel and a device ID as arguments. This is not identical to the device_id used by OpenCL, but a consecutive numbering of all available devices. The ID defaults to zero if it is not specified.

```
static program create(const char* kernel_source,
                      uint32_t device_id = 0);
```

The function create accesses the list of the available devices from the opencl_metainfo singleton and checks whether a matching device exists. In the next step, it creates an OpenCL program object and compiles the code for the selected device. The build log is printed to std::cerr in case of an error. After collecting the mandatory information, the constructor of program is invoked.

5.6.3 Creating OpenCL-enabled Actors

The function spawn_cl creates an actor_facade and returns a reference to it. The function has five arguments, including two optional ones. The required arguments are the source code of a kernel, the name of a kernel function in the code and a vector specifying the distribution among work-items. The optional arguments specify an offset for global IDs and the work-group size. The type dim_vec restricts the number of dimensions to the maximum allowed by OpenCL.

In addition to the arguments, this function expects the signature of the kernel as a template parameter. The actor_facade uses the signature to identify invalid messages and to map message elements to kernel arguments.

```
template<typename Signature, typename... Ts>
inline actor_ptr spawn_cl(const char* source,
                          const char* fname,
                          const opencl::dim_vec& dims,
                          const opencl::dim_vec& offset = {},
                          const opencl::dim_vec& local_dims = {});
```

The arguments to the kernel are passed in containers of the type std::vector. OpenCL expects the arguments as C arrays, which can be obtained from the vector. The function spawn_cl is overloaded to allow user-defined types rather than vectors. It is shown in the following code and expects the two additional arguments map_args and map_result. The first functor converts an incoming message to types that match the signature of the kernel. The second functor converts the result from the kernel execution to a user-defined type. This overload deduces the template parameters from the signatures of the functors.

```
template<typename MapArgs, typename MapResult>
inline actor_ptr spawn_cl(const char* source,
                          const char* fname,
                          MapArgs map_args,
                          MapResult map_result,
                          const opencl::dim_vec& dims,
                          const opencl::dim_vec& offset = {},
                          const opencl::dim_vec& local_dims = {});
```

Two additional overloads accept a kernel wrapped in a `program` instead of a string as first argument. The class `program` is explained in Section 5.6.2.

All `spawn_cl` functions pass their arguments to the `cl_spawn_helper`. It is a metaprogramming utility that assembles the template parameters for the `actor_facade` and creates default mapping functions. The following code shows the utility class `carr_to_vec` and the `cl_spawn_helper` with two specializations.

```
template<typename T>
struct carr_to_vec { typedef T type; };

template<typename T>
struct carr_to_vec<T*> { typedef std::vector<T> type; };

template<typename Signature, typename SecondSignature = void>
struct cl_spawn_helper;

template<typename R, typename... Ts>
struct cl_spawn_helper<R (Ts...), void> {

    using result_type = typename carr_to_vec<R>::type;

    using impl = opencl::actor_facade<
                    result_type(typename carr_to_vec<
                                    typename carr_to_vec<
                                        Ts
                                    >::type
                                >::type...)
                >;
    using map_arg_fun = typename impl::arg_mapping;
    using map_res_fun = typename impl::result_mapping;

    template<typename... Us>
    actor_ptr operator()(map_arg_fun f0,
                         map_res_fun f1,
                         const opencl::program& p,
                         const char* fname,
                         Us&&... args) const {
        using std::move;
        using std::forward;
        return impl::create(p, fname,
                             move(f0), move(f1),
                             forward<Us>(args)...);
    }

    template<typename... Us>
    actor_ptr operator()(const opencl::program& p,
                         const char* fname,
                         Us&&... args) const {
        using std::move;
        using std::forward;
        map_arg_fun f0 = [] (any_tuple msg) {
            return tuple_cast<
                         typename util::rm_const_and_ref<
                             typename carr_to_vec<Ts>::type
                         >::type...
                     >(msg);
        };
        map_res_fun f1 = [] (result_type& result) {
            return make_any_tuple(move(result));
        };
        return impl::create(p, fname,
                            move(f0), move(f1),
                            forward<Us>(args)...);
    }

};

template<typename R, typename... Ts>
struct cl_spawn_helper<
    std::function<optional<cow_tuple<Ts...>> (any_tuple)>,
    std::function<any_tuple (R&)>
>
: cl_spawn_helper<R (Ts...)> { };
```

The struct `carr_to_vec` is used to convert `C` style arrays in the signature of kernels to `C++` vectors. Unless a pointer is passed as a template parameter, `type` is defined as the parameter `T`. If a pointer is passed, the template specialization converts the pointer to a vector, assuming that the pointer is in fact a `C` array.

The first specialization of the `cl_spawn_helper` has the template parameter `R` and a variadic template parameter pack `Ts`. `R` represents the return type of a function and `Ts` the types of the arguments. This specialization expects only a function signature `R (Ts...)` and no second parameter. The information about result and parameter type are used to specify the template parameters for the `actor_facade` type. All pointer arguments are removed from the “new” kernel signature and replaced with vector types. The second template specialization allows to use custom mapping functions which retrieve the necessary type information from the signature of the mapping functor, as shown in Section 5.6.4. The function `spawn_cl` uses the apply operator of the `cl_spawn_helper` to pass all arguments required to create the `actor_facade`. The arguments are similar to the arguments from the `spawn_cl` function. When called without mapping functions, the `cl_spawn_helper` creates default versions and uses them instead.

5.6.4 The `actor_facade`

The class `actor_facade` encapsulates an OpenCL kernel and is created by the function `spawn_cl` described in section 5.6.3. It matches incoming messages against the signature of the kernel, pushes the kernel arguments to OpenCL, and reads and forwards the results from the GPU. The class is derived from the class `actor`.

```cpp
template<typename Signature>
class actor_facade;

template<typename Ret, typename... Args>
class actor_facade<Ret(Args...)> : public actor {

    using args_tuple =
        cow_tuple<typename util::rm_const_and_ref<Args>::type...>;
    using arg_mapping = function<optional<args_tuple>(any_tuple)>;
    using result_mapping = function<any_tuple(Ret&)>;

    /* implementation */
}
```

The code above shows the template parameters of the class `actor_facade`. They represent the signature of the encapsulated kernel. It has a return type `Ret` and expects a variadic number of arguments. The template parameters are used to create the function types of the mapping functions. A function of the type `arg_mapping` reads the arguments from an `any_tuple` and has an optional return type. Once the kernel execution is done, a function of the type `result_mapping` takes an argument of the type `Ret` and returns an `any_tuple`, that will be used as a response.

```cpp
actor_facade::actor_facade(const program& prog,
                           kernel_ptr kernel,
                           const dim_vec& global_dimensions,
                           const dim_vec& global_offsets,
                           const dim_vec& local_dimensions,
                           arg_mapping map_args,
                           result_mapping map_result);
```

The constructor of `actor_facade` requires seven arguments. The first arguments is a `program`, which holds information about the device and command queue used for the kernel. The second argument is the kernel. The next three arguments are vectors containing information about the kernel distribution among work-items and work-groups. The last two arguments are mapping functions with the types `arg_mapping` and `result_mapping`. Either they are user-defined and passed via `spawn_cl`, or the `cl_spawn_helper` creates default implementations that match the signature of the kernel.

```
static intrusive_ptr<actor_facade> actor_facade::create(
                                     const program& prog,
                                     const char* kernel_name,
                                     arg_mapping map_args,
                                     result_mapping map_result,
                                     const dim_vec& global_dims,
                                     const dim_vec& offsets,
                                     const dim_vec& local_dims
                                 );
```

To avoid errors in the constructor, all input values are checked before an instance of the `actor_facade` is created. Hence, it offers a factory function as explained in Section 5.2 for its creation and declares the constructor private. The arguments expected by the factory differ minimally from the constructor. Instead of a kernel, it expects a kernel name, which is used to extract a kernel from a program (Section 5.6.2). Furthermore, it checks if the vectors containing the parameters for the kernel distribution are consistent and contain meaningful data.

```
1 void actor_facade::enqueue(const message_header& hdr,
2                            any_tuple msg) override {
3     typename
4         util::il_indices<util::type_list<Args...>>::type indices;
5     enqueue_impl(hdr.sender, std::move(msg), hdr.id, indices);
6 }
```

A message is delivered to an actor through the function `enqueue`. Line 3 and 4 in the code above create a type from the `Args` template parameter pack of the class, which contains the indices for all elements in the `msg` tuple. This information is required to extract all elements from the input type in the next function. Along with the received msg and the variable of the newly created type, the sender of the message and a message ID are passed to `enqueue_impl`.

```
1  template<long... Is>
2  void actor_facade::enqueue_impl(const actor_ptr& sender,
3                                  message_id id,
4                                  any_tuple msg,
5                                  util::int_list<Is...>) {
6      auto opt = m_map_args(move(msg));
7      if (opt) {
8          response_handle handle{this, sender,
9                                 id.response_id()};
10         size_t ret_size = accumulate(m_global_dimensions.begin(),
11                                      m_global_dimensions.end(),
12                                      1, multiplies<size_t>{});
13         vector<mem_ptr> arguments;
14         add_arguments_to_kernel<Ret>(arguments,
15                                      ret_size,
16                                      get_ref<Is>(*opt)...);
17         auto cmd = make_counted<
18                        command<actor_facade, Ret>
19                    >(handle, this, move(arguments));
20         cmd->enqueue();
21     }
22     else {
23         CPPA_LOGMF(CPPA_ERROR, this,
24                    "actor_facade::enqueue() tuple_cast failed.");
25     }
26 }
```

The code above shows the implementation of `enqueue_impl`. The arguments `sender` and `id` are used to create a response handle in line 8. After the calculation is done, the handle identifies the original request unambigiously and is used to send a response message. The third argument contains the arguments for the kernel execution. The type of the fourth argument contains indices for all elements in the `msg` tuple. These indices are used to access individual tuple elements.

In line 6, the argument mapping function converts the message data to the arguments expected by the kernel. If the conversion fails, we log an error and the kernel is not executed. `ret_size` determines the size to the result buffer for the kernel execution. We assume, that the result size is equal to the number of work-items. This allows every work-item to calculate one result. In future releases, the result size will be configureable. To calculate the size, we multiply the work-items for each dimension. The function `add_arguments_to_kernel` creates and initializes memory object for each argument. It is discussed in Section 5.6.5. The kernel execution is wrapped in the class `command`. It runs asynchronously and is triggered via `cmd->enqueue()`. Section 5.6.6 explains its implementation.

5.6.5 Using Variadic Templates to Create Memory Objects from Message Data

The function `add_arguments_to_kernel` creates a memory object (see Section 2.3.4) for each message element and the result buffer. Furthermore, it sets the memory objects as arguments for the kernel execution. Recursion is used to iterate over the template parameters, which specify the types of the elements.

```
template<typename R, typename... Ts>
void add_arguments_to_kernel(args_vec& arguments,
                             size_t ret_size,
                             Ts&&... args);
```

The function itself expects a buffer for the memory objects, the size of the result buffer and a variadic number of message elements. The template parameters state the type of values stored in the result buffer and a variadic number of types, which specify the type of each message element. This function creates the result buffer and starts the recursion.

```
template<typename T0, typename... Ts>
void add_arguments_to_kernel_rec(args_vec& arguments,
                                 T0& arg0, Ts&... args);
void add_arguments_to_kernel_rec(args_vec& arguments);
```

The recursion is implemented using the overload function `add_arguments_to_kernel_rec`. Both implementations expect a vector for the created memory objects as first argument. The first function expects the head of the elements `arg0` and the tail `args`. It is called until all elements are processed and creates a memory object for each element, which is then placed in the arguments vector. The second function ends the recursion and thus has no template arguments. It iterates over the vector and sets the objects as kernel arguments.

5.6.6 Wrapping the Kernel Execution

The class command wraps a kernel execution and provides a callback function to OpenCL. Its template parameters specify the type `T` of the `actor_facade` instance that created the command and the result type `R` of the kernel. The `command` is derived from `ref_counted`, which enables memory management via reference count, as explained in section 5.3.1.

```
template<typename T, typename R>
class command : public ref_counted {

 public:
    command(response_handle handle,
            intrusive_ptr<T> actor_facade,
            std::vector<mem_ptr> arguments);

    /* more code */
};
```

The constructor of the class `command` requires three arguments. The first argument is a `response_handle`, which specifies the recipient of the results. The second argument is the `actor_facade` instance that created the `command`. It is required to access the kernel and related information. The last argument is a vector containing the kernel arguments. Once the `command` is created, the `actor_facade` calls its `enqueue` function.

```
void command::enqueue () {
    this->ref();
    auto event = m_kernel_event.get();
    auto data_or_nullptr = [](const dim_vec& vec) {
        return vec.empty() ? nullptr : vec.data();
    };
    clEnqueueNDRangeKernel(
        m_queue.get(),
        m_actor_facade->m_kernel.get(),
        m_actor_facade->m_global_dimensions.size(),
        data_or_nullptr(m_actor_facade->m_global_offsets),
        data_or_nullptr(m_actor_facade->m_global_dimensions),
        data_or_nullptr(m_actor_facade->m_local_dimensions),
        0,
        nullptr,
        event
    );
    clSetEventCallback(
        event,
        CL_COMPLETE,
        [](cl_event, cl_int, void* data) {
            auto cmd =
                reinterpret_cast<command*>(data);
            cmd->handle_results();
            cmd->deref();
        },
        this
    );
    clFlush(m_queue.get());
}
```

Since the `command_queue` needs the `command` for its callback, it increments its own reference count in line 2. This represents the reference held by OpenCL and prevents the deletion of the command. Afterwards, it acquires an event, which is required to link a callback to the kernel execution.

The function `clEnqueueNDRangeKernel` is an OpenCL API call to enqueue a kernel to a command queue. It is explained in Section 4.5. In addition, the function `data_or_nullptr` returns either a pointer to an array inside a vector or a `nullptr`, if the vector is empty. It is used to pass a `nullptr` to OpenCL if the user did not specify a value for the dimensions or offset. Furthermore, the last argument is the event mentioned earlier.

In the next step, a callback is set with the function `clSetEventCallback`. The first parameter is the event we have passed to the function `clEnqueueNDRangeKernel`. This links the callback to the kernel execution. The next parameter specifies a trigger for the callback invocation. A callback can only be registered with the `CL_COMPLETE` flag that causes OpenCL to invoke the callback. The third argument is the callback function. It has to match the signature `void (...) (cl_event, cl_int, void *)`. The `void` pointer allows users to pass arbitrary data to `clSetEventCallback`. We pass a pointer to the command itself. Once the callback is set, `clFlush` sends all operations in a command queue to the device.

Once the kernel finished execution, the callback dereferences the pointer to the `command` and calls its member function `handle_results`. Finally, the reference count is decremented.

```
void command::handle_results () {
    R result(m_number_of_values);
    clEnqueueReadBuffer(m_queue.get(),
                        m_arguments[0].get(),
                        CL_TRUE,
                        0,
                        sizeof(typename R::value_type)
                            * m_number_of_values,
                        result.data(),
                        0,
                        nullptr,
                        nullptr);
    reply_tuple_to(m_handle, m_actor_facade->m_map_result(result));
}
```

The function handle_result sends the results to the client. The result type R is a template parameter of the class command. The function clEnqueueReadBuffer is explained in Section 3.3.3. Before the results are sent, they are converted using the function m_map_result from the actor_facade. The client is identified via the response handle m_handle.

6 Evaluation

In this section, we compare our work to established implementations, specifically the native OpenCL API and event-based actors from `libcppa` actors. The first benchmark compares the runtime of our implementation to the native API of OpenCL and measure the overhead we induced. The second test compares the creation time of OpenCL-enabled actors with original event-based actors. The third benchmark is published in Charousset et al. (2013) and examines the scaling behavior in a distributed, heterogeneous system. Finally we compare the performance of several GPUs.

Our first two benchmarks are run on a Tesla C2075 workstation GPU on a 12 core server running Ubuntu 13.04 with the proprietary driver provided by Nvidia. The setup for the third benchmark uses a distributed setup that consists of eight nodes. The last benchmark makes use of different GPUs available in our lab, the details of which are presented as part of the experimental setting.

6.1 Runtime Overhead of Actors Over Native OpenCL Programming

Our first benchmark program measures the overhead induced by our actor approach compared to the native API of OpenCL. Though the OpenCL-enabled actor uses the OpenCL API internally, it performs additional steps such as the setup of the runtime environment and the actor creation. This benchmark quantifies the overhead. We have implemented one program with the native API and one with `libcppa`. Since both programs use the same kernel, the runtime should be independent of the pure numerical calculations on the GPU and differences should only depend on the different processing overheads. The kernel calculates the product of two matrices and is similar to the kernel we used in Sections 3.3.3. Both programs use the same configuration for the kernel execution and work asynchronously using a callback.

```
void matrix_mult_native(iterations) {
    init_opencl();
    for (0 to iterations) {
        enqueue_kernel(...);
        set_callback();
        wait_for_callback();
    }
}
```

The pseudo code above shows the native benchmark variant. The function `init_opencl` represents the setup process for OpenCL and the creation of matrices for the kernel execution. Each iteration of the loop enqueues the kernel, sets a callback and waits for the callback to finish.

```
void matrix_mult_cppa(iterations) {
    actor = spawn_cl( ... );
    for (0 to iterations) {
        send(actor, matrix1, matrix2);
        await_result_message();
    }
}
```

The benchmark variant with `libcppa` creates an OpenCL-enabled actor, which performs the calculation on the GPU. In each iteration, a message with the matrices is sent to the actor and the result are awaited. We measured the wall-clock time required to perform an increasing number of matrix multiplications.

Figure 6 displays the wall-clock time required by the programs as a function of the number of iterations. We performed a high number of iterations to enlarge the runtime difference between both programs. Adding complexity to the calculations on the GPU would only increase the runtime of both programs, but not their difference. We have plotted the average from 50 measurements as well as a 95% confidence interval. Both implementations exhibit linear growth with a similar slope. Since we use the OpenCL API inside `libcppa` it is not possible to achieve a better performance than OpenCL itself, which would result in a smaller slope. However, a larger slope for our implementation would indicate that we produce an overhead. Overhead could ,e.g., originate in unnecessary copy operations. Since both implementations show a similar slope, the induced overhead seems to be negligible.

The difference between the two curves is plotted in Figure 7. For some values the confidence interval is too large to observe a detailed behavior. However, the values for 1000, 6000 and 8000 have a small confidence interval and a line of best fit would exhibit a linear growth of very low slope. This is expected. Since the overhead does not depend on the calculation time it is smaller in relation to the absolute wall-time, if the calculation takes longer.

6.2 Spawn time

Our second benchmark focuses on the time to instantiate OpenCL-enabled actors. It compares the creation time of OpenCL-enabled actors to that of Event-based actors from `libcppa`. We expect the creation time for OpenCL actors to be longer, but want to show

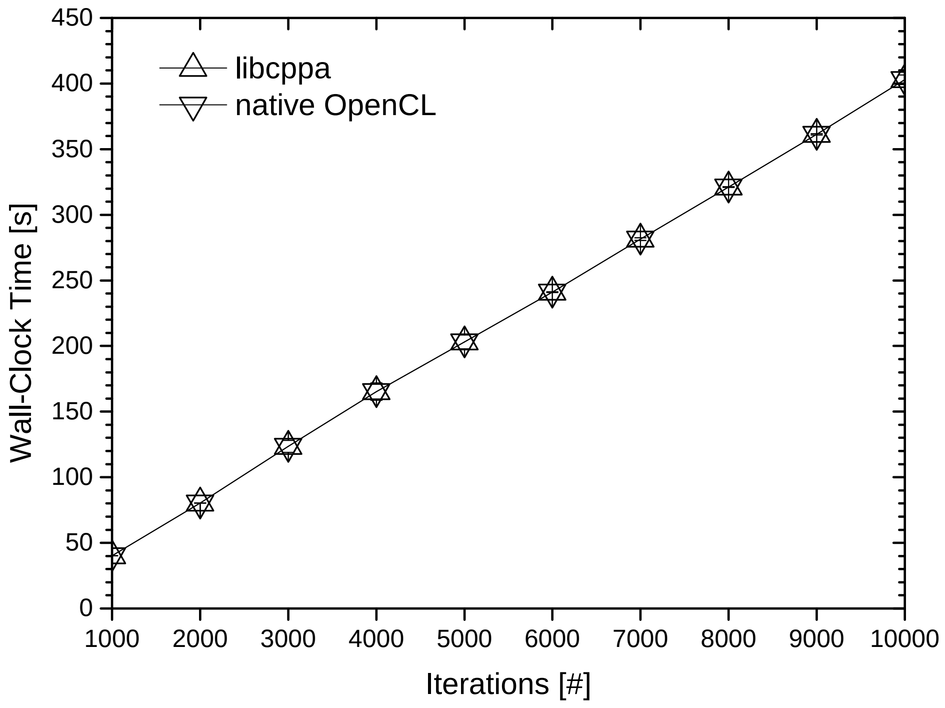


Figure 6: Comparison of the wall-clock time required for a matrix multiplication implemented in OpenCL, native vs. `libcppa`.

that its creation is a lightweight operation and can be used for short calculations as well as for longer ones.

```
void spawn_event_based() {
    for (0 to iterations) {
        spawn( ... );
    }
    wait_for_all_actors();
}
```

```
void spawn_opencl_enabled() {
    prog = create_program();
    for (0 to iterations) {
        spawn_cl(prog, ... )
    }
    wait_for_all_actors();
}
```

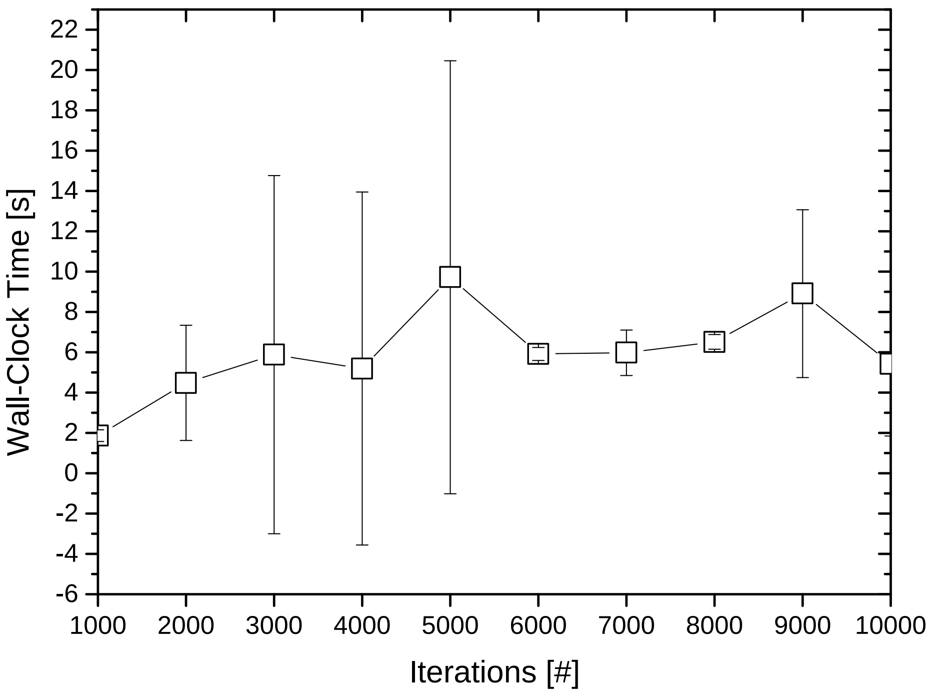


Figure 7: Absolute runtime difference between the implementations, native vs. `libcppa`.

Both benchmarks consist of a loop that spawns one actor per iteration. Afterwards we ensure that all actors are active by sending a message to the last created actor and waiting for its answer. We measure the wall-time required to spawn an increasing number of actors.

Figure 8 visualizes wall-clock runtime as a function of spawned actors. Both implementations show a linear growth. Event-based actors take less time then OpenCL-enabled actors and exhibit a smaller slope.

Compared to the time required for a simple calculation, such as the matrix multiplication in the first benchmark, the creation time is reasonably small. It is worth mentioning that OpenCL-enabled actors are parallelized internally by OpenCL. They are not created as frequently as event-based actors. Hence, creation time is less import for the OpenCL-enabled actor.

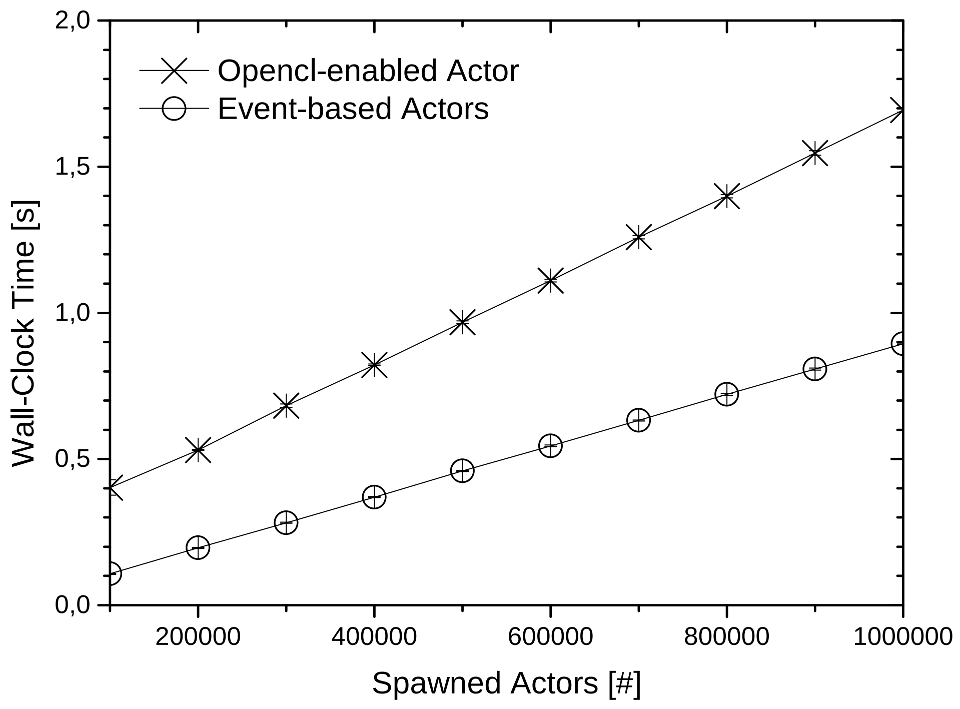


Figure 8: Comparison of the wall-clock time required to spawn OpenCL-enabled and event-based actors.

6.3 Scaling Behavior in a Heterogeneous Setup

The work of Charousset et al. (2013) presents several benchmarks regarding `libcppa`, including a heterogenous setup that examines a distributed system with eight worker nodes. The system calculates a fixed number of images of the Mandelbrot set.

Figure 9 shows the wall-clock runtime as a function of the deployed processing units. In the initial setup, each node uses one CPU for the calculation. Each node is then strengthened by either an additional CPU or an additional GPU.
Adding an additional CPUs scales linearly and reduces the runtime by a factor of ≈ 0.9. However, adding only one GPU reduces the runtime to a third. Doubling the number of GPUs thereafter reduces the runtime by a factor of ≈ 0.6. For the eight additional nodes, the heterogenous setup exhibits one eighth of the runtime the homogeneous system exhibits.
The benchmark provides an example of the performance gain available through heterogeneous systems.

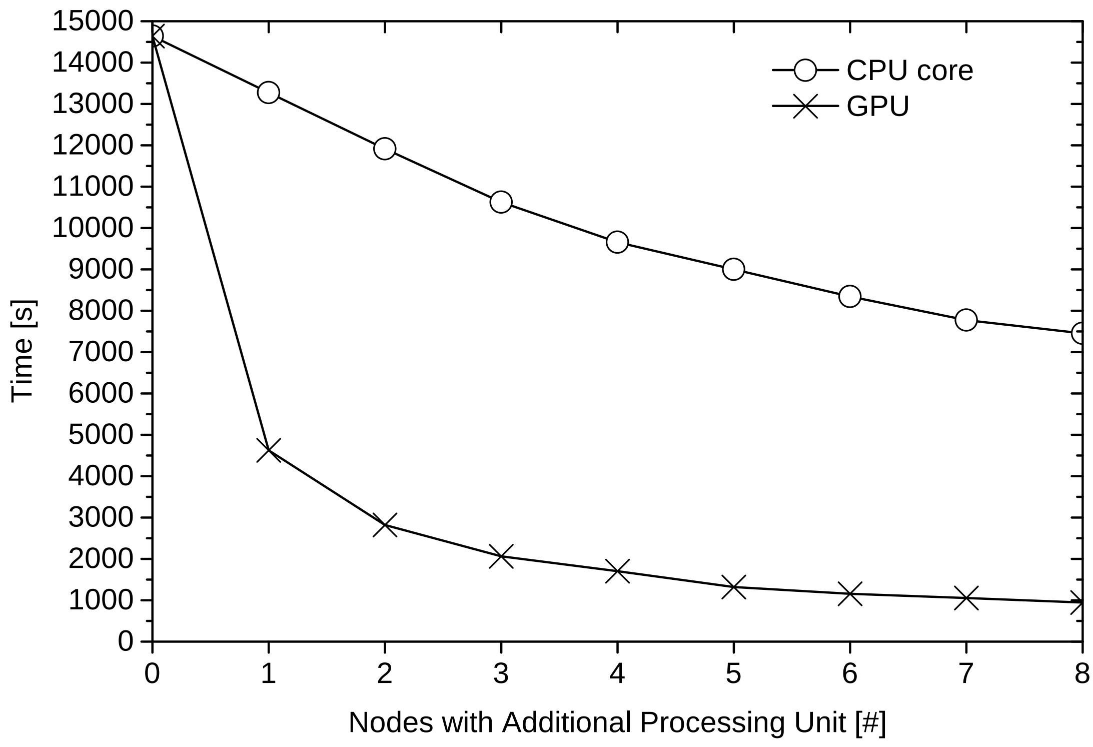


Figure 9: Sending and processing time in a distributed computation using heterogeneous hardware ressources, from Charousset et al. (2013).

6.4 Comparing Graphics Cards

Our final benchmark examines different GPUs with regard to their memory access capabilities. Aside from its internal hardware characteristics, such as the number of cores or memory size, their performance is characterized by the surrounding hardware, especially the available memory bandwidth. The kernel we use for the benchmark does not include complex calculations, but copies data from an input to an output array. Hence, the main runtime will be consumed by copy operations between GPU and CPU. The program creates an OpenCL-enabled actor before sending a message to the actor and waiting for the results in a loop.

```
__kernel void copy(input, output) {
    output[myID] = input[myID];
}

void comapre_gpus(iterations) {
    actor = spawn_cl(copy, ... );
    for (0 to iterations) {
        send(actor, ... );
        await_result_message();
    }
}
```

Figure 10 shows the characteristics of the GPUs we have used to conduct our benchmarks. The columns show (from left to right): (1) the number of compute cores, (2) the clock frequency in MHz, (3) the maximum number of work-items in a work-group executing a kernel using the data parallel execution model, (4) the maximum memory bandwidth in GB/s, (5) global memory size and (6) the support of the flag `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` that allows a command queue to optimize kernel execution. The Tesla is a dedicated GPGPU co-processor and provides a large amount of memory and cores. The Quadro is also a workstation GPU, but has fewer cores, fewer memory and is built for power-efficiency. The other two GPUs are mobile versions, as indicated by the “M” in their names. The GeForce is build into a Thinkpad SL500 and Radeon GPU is build into an iMac. All PCs but the iMac run Ubuntu and use drivers provided by Nvidia. The iMac runs OS X and uses the driver shipped with the OS.

Figure 11 shows the runtime as a function of the performed iterations. We plotted the average of 50 measurements and a 95% confidence interval. All graphic cards exhibit linear growth. The Radeon outperforms the others, while the Quadro shows the worst performance.

Compared to the memory bandwidth from Figure 11, it is expected that the Radeon performs best, but the difference to the Tesla should not be as big. Furthermore, the Quadro has better specifications than the GeForce, but performs worse.

Overall, the benchmark seems to favor both mobile GPUs. However, further investigation is necessary to explain the deviation from the characteristics and draw reasoned conclusion.

Name	Cores	Clock [MHz]	Parallel Work-Items	Bandwidth [GBs/s]	Memory [MB]	Out of Order Execution
Tesla C2075	14	1147	1024	144	5636	Yes
Quadro 600	2	1280	1024	25	1073	Yes
GeForce 9300M GS	1	1450	512	7	267	No
Radeon HD 6970M	12	680	1024	176	1073	No

Figure 10: Specifications of the used GPUs.

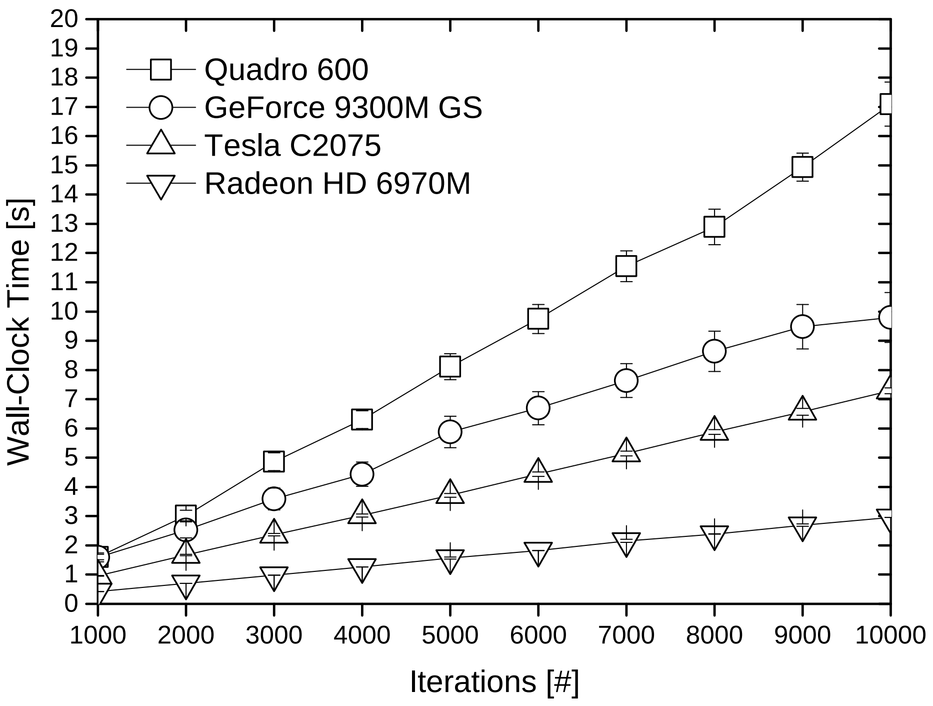


Figure 11: Comparison of the wall-clock time required to run the benchmark on different GPUs.

7 Conclusion & Outlook

In this work, we have extended the `C++` actor library `libcppa` with capabilities for GPU computing. We introduced the new actor class `actor_facade`, which is created by the function `spawn_cl`. It is integrated into the runtime environment of `libcppa` and profits from access transparency for message passing and error handling. However, we did neither achieve access transparency nor location transparency for actor creation. To provide an easy user interface, we automated repetitive task of the OpenCL setup process, such as platform and device discovery.

Our benchmark results indicate that the OpenCL-enabled actor does not induce noticeable overhead compared to the native OpenCL API. Since our implementation does not affect the runtime performance of the GPU, the relative overhead gets smaller the longer the calculation on the GPU takes. Furthermore, we measured the creation time for OpenCL actors. It takes longer to create an OpenCL actor than to create an event-based actor, but the creation time is small compared to the the absolute runtime.

While writing this thesis, it became clear, that some use cases require more control over OpenCL specific parameters. For example, the research of Fang et al. (2011) shows specific compiler options are important for performance tuning. In our future work we would like address this and offer an interface to configure our actors accordingly. This can be achieved by extending our kernel wrapper, which currently links a kernel to an OpenCL device. Alternatively, a new class could package configuration parameters and be passed to the function `spawn_cl` as an additional argument.

In the context of location transparency we mentioned the creation of remote OpenCL actors. The implementation is left for future work, but the information necessary to create an OpenCL actor are available at runtime. A future implementation could allow the runtime environment on the remote note to create an OpenCL actor with the data.
A completely transparent approach could switch to remote actor if its own platform does not provide the appropriate hardware. However, OpenCL kernels can be tuned to exploit hardware dependent features, which are possibly not available on a remote system. Additionally, the explicit creation of remote OpenCL actors could be enabled.

The performance could be further improved by allowing direct messages between OpenCL-enabled actors that remain on the device. This would prevent copy operations between GPU and CPU and thus improve performance. Although each OpenCL-enabled actor already holds a reference to memory buffers on the GPU, it reads the values back from the GPU before sending them to another actor. Instead, OpenCL actors would pass the reference directly rather than copying the values while communicating with another OpenCL-enabled

actor.
The copy operations between GPU and CPU can be executed using different flags and functions. We did not compare the different approaches yet, but determining which method suits best for which use-case could on the overall lead to an improved performance. In addition, we want to give users the possibility to configure the flags according to their use case.

A further direction for future work is load balancing. Currently, the programmer is responsible for ensuring that kernels are distributed among devices and to manage the device memory. Ideally, this task should be transferred to the runtime environment, since it has information about ongoing processes and can ensure that access from multiple sources is handled appropriately. OpenCL does not offer load balancing for multiple GPUs and it remains an open question whether enough runtime information are available to implement proper load balancing.

References

(2013). The Khronos Group. http://www.khronos.org/.

Agha, G. (1986). Actors: A Model Of Concurrent Computation In Distributed Systems. Technical Report 844, MIT, Cambridge, MA, USA.

Armstrong, J. (1997). The Development of Erlang. In *ICFP '97 - Proceedings of the second ACM SIGPLAN International Conference on Functional Programming*, pages 196–203, New York, NY, USA. ACM.

Armstrong, J. (2003). *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Department of Microelectronics and Information Technology, KTH, Sweden.

Armstrong, J. (2007). A History of Erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III)*, pages 6–1–6–26, New York, NY, USA. ACM.

Blythe, D. (2006). The Direct3D 10 System. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 724–734, New York, NY, USA. ACM.

Charousset, D. (2012). libcppa – An actor library for C++ with transparent and extensible group semantic. Master's thesis, Hamburg University of Applied Sciences.

Charousset, D. and Schmidt, T. C. (2013). libcppa - Designing an Actor Semantic for C++11. In *Proc. of C++Now*.

Charousset, D., Schmidt, T. C., Hiesgen, R., and Wählisch, M. (2013). Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments. In *Proc. of the 4rd Annual Conference on Systems, Programming, and Applications (SPLASH '13), Workshop AGERE!*, New York, NY, USA. ACM.

Fang, J., Varbanescu, A. L., and Sips, H. (2011). A Comprehensive Performance Comparison of CUDA and OpenCL. In *Parallel Processing (ICPP)*, pages 216–225.

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960.

Hennessy, J. L. and Patterson, D. A. (2012). *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, fifth edition.

Hewitt, C., Bishop, P., and Steiger, R. (1973). A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd IJCAI*, pages 235–245, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Hillis, W. D. (1985). *The Connection Machine*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Intel (2013). *Intel® Architecture Instruction Set Extensions Programming Reference*. Intel, 319433-015 edition.

Kirk, D. B. and Hwu, W.-m. W. (2013). *Programming Massively Parallel Processors, A Hands-on Approach*. Morgan Kaufmann, second edition.

Lindholm, E., Kilgard, M. J., and Moreton, H. (2001). A User-Programmable Vertex Engine. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 149–158, New York, NY, USA. ACM.

Munshi, A. (2012). *The OpenCL Specification*. Khronos OpenCL Working Group, Khronos Group. Version 1.2, Revision 19.

Nickolls, J. and Dally, W. J. (2010). The GPU Computing Era. *IEEE Micro*, 30(2):56–69.

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008). GPU Computing. *Proceedings of the IEEE*, 96(5):879–899.

Peleg, A., Wilkie, S., and Weiser, U. (1997). Intel MMX for multimedia PCs. *Commun. ACM*, 40(1):24–38.

Scarpino, M. (2011). *OpenCL in Action: How to Accelerate Graphics and Computation*. Manning Publications Company, Manning Publication Co., 20 Baldwin Road, Shelter Island, NY 11964.

List of Figures

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, October 24, 2013 Raphael Hiesgen