Manyfold Actors: Extending the C++ Actor Framework to Heterogeneous Many-Core Machines using OpenCL

Raphael Hiesgen Dominik Charousset Thomas C. Schmidt
Dept. Computer Science, Hamburg University of Applied Sciences, Germany
{raphael.hiesgen,dominik.charousset,t.schmidt}@haw-hamburg.de

Abstract

The processing power of modern many core hardware such as graphics processing units (GPUs) or coprocessors is increasingly available for general-purpose computation. The seamless way of actor systems to addresses concurrent and distributed programming makes it an attractive approach to integrate these novel architectures. In this work, we introduce OpenCL-enabled actors to the C++ Actor Framework (CAF). This offers a high level interface for accessing any OpenCL device without leaving the actor paradigm. The new type of actor is integrated into the runtime environment of CAF and gives rise to transparent message passing in distributed systems on heterogeneous hardware. New actors are instantiated by the function spawn_cl, while the runtime environment handles the discovery and setup of OpenCL devices in the background. Our evaluations on a commodity GPU, an Nvidia TESLA, and an Intel PHI reveal the expected linear scaling behavior when offloading larger work items. For sub-second duties, the efficiency of offloading was found to largely differ between devices. Moreover, our findings indicate a negligible overhead over programming the native OpenCL API.

Categories and Subject Descriptors C.1.3 [*Other Architecture Styles*]: Heterogeneous (hybrid) systems; C.2.4 [*Distributed Systems*]: Distributed applications; D.1.3 [*Programming Techniques*]: Concurrent programming; D.3.4 [*Processors*]: Run-time environments

Keywords Actor Model, C++, GPGPU Computing, OpenCL, Coprocessor

AGERE! 2015, October, 2015, Pittsburgh, PA, OSA.

ACM 978-1-4503-2189-1/14/10. . . $15.00.
http://dx.doi.org/10.1145/2824815.2824820

1. Introduction

The stagnating clock speed forced CPU manufacturers into steadily increasing the number of cores on commodity hardware to meet the ever-increasing demand for computational power. Still, the number of parallel processing units on a single GPU is higher by orders of magnitudes. This rich source of computing power became available to general purpose applications as GPUs moved away from single purpose pipelines for graphics processing towards compact clusters of data-parallel programmable units [26].

Algorithms that can be mapped to the data-parallel architecture of GPUs can expect a massive boost in performance. Combined with the widespread availability of general-purpose GPU (GPGPU) devices on desktops, laptops and even mobiles, GPGPU computing has been widely recognized as an important optimization strategy. In addition, accelerating coprocessors that better support code branching established on the market.

Since not all tasks can benefit from such specialized devices, developers need to distribute work on the various architectural elements. Managing such a heterogeneous runtime environment inherently increases the complexity. While some loop-based computations can be offloaded to GPUs using OpenACC [7] or recent versions of OpenMP [9] with relatively little programming effort, it has been shown that a consistent task-oriented design exploits the available parallelism more efficiently. Corresponding results achieve better performance [18] while they are also applicable to more complex work loads. However, manually orchestrating tasks between multiple devices is an error-prone and complex task.

The actor model of computation describes applications in terms of isolated software entities—actors—that communicate by asynchronous message passing. Actors can be distributed across any number of processors or machines by the runtime system as they are not allowed to share state and thus can always be executed in parallel. The message-based decoupling of software entities further enables actors to run on different devices in a heterogeneous environment. Hence, the actor model can simplify software development by hid-

ing the complexity of heterogeneous and distributed deployments.

In this work, we introduce actors programmed with OpenCL—the Open Computing Language standardized by the Khronos Group [29]. We integrate heterogeneous programming into the C++ Actor Framework [8], and thoroughly examine the runtime overhead introduced by our abstraction layer. We aim at integrating heterogeneous hardware to the existing benefits of CAF such as network-transparency, memory-efficiency and high performance.

The remainder of this paper is organized as follows. Section 2 introduces the actor model as well as heterogeneous computing in general and OpenCL in particular. Our design goals and their realization are discussed in Section 3 as well the limitations of our approach. In Section 4, we evaluate the performances of our work with a focus on overhead and scalability. Finally, Section 5 concludes and gives an outlook to future work.

2. Background and Related Work

Before showing design details, we first discuss the actor model of computation, heterogeneous computing in general, and OpenCL.

2.1 The Actor Model

Actors are concurrent, isolated entities that interact via message passing. They use unique identifiers to address each other transparently in a distributed system. In reaction to a received message, an actor can, (1) send messages to other actors, (2) spawn new actors and (3) change its own behavior to process future messages differently.

These characteristics lead to several advantages. Since actors can only interact via message passing, they never corrupt each others state and thus avoid race conditions by design. Work can be distributed by spawning more actors in a divide and conquer approach. Further, the actor model addresses fault-tolerance in distributed systems by allowing actors to monitor each other. If an actors dies unexpectedly, the runtime system sends a message to each actor monitoring it. This relation can be strengthened through bidirectional monitors called links. By providing network-transparent messaging and fault propagation, the actor model offers a high level of abstraction for application design and development targeted at concurrent and distributed systems.

Hewitt et al. [13] proposed the actor model in 1973 as part of their work on artificial intelligence. Later, Agha formalized the model in his dissertation [1] and introduced mailboxing for processing actor messages. He created the foundation of an open, external communication [2]. At the same time, Armstrong took a more practical approach by developing Erlang [5].

Erlang is a concurrent, dynamically typed programming language developed for programming large-scale, fault-tolerant systems [4]. Although Erlang was not build with the actor model in mind, it satisfies its characteristics. New actors, called processes in Erlang, are created by a function called `spawn`. Their communication is based on asynchronous message passing. Processes use pattern matching to identify incoming messages.

To combine the benefits of a high level of abstraction and native program execution, we have developed the C++ Actor Framework (CAF) [8]. Actors are implemented as sub-thread entities and run in a cooperative scheduler using work-stealing. As a result, the creation and destruction of actors is a lightweight operation. Uncooperative actors that require access to blocking function calls can be bound to separate threads by the programmer to avoid starvation. Furthermore, CAF includes a runtime inspection tool to help debugging distributed actor systems.

In CAF, actors are created using the function `spawn`. It creates actors from either functions or classes and returns a network-transparent actor handle. Communication is based on message passing, using `send` or `sync_send`. Note that the latter function only suspends an actor until the response arrives but does not block any system resources.

CAF offers dynamically as well as statically typed actors. While the dynamic approach is closer to the original actor model, the static approach allows programmers to define a message passing interface which is checked by the compiler for both incoming and outgoing messages.

Messages are buffered at the receiver in order of arrival before they are processed. The behavior of an actor specifies its response to messages it receives. CAF uses partial functions as message handlers, which are implemented using an internal domain-specific language (DSL) for 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.

In previous work [8], we compared CAF to other actor implementations. Namely Erlang, the Java frameworks SALSA Lite [10] and ActorFoundry (based on Kilim [28]), the Scala toolkit and runtime Akka [30] and Charm++ [15]. We measured (1) actor creation overhead, (2) sending and processing time of message passing implementations, (3) memory consumption for several use cases and (4) picked up a benchmark from the Computer Language Benchmarks Game. The results showed that CAF displays consistent scaling behavior, minimal memory overhead and very high performance.

2.2 Heterogeneous Computing

Graphic processing units (GPUs) were originally developed to calculate high resolution graphic effects in real-time [23]. High frame rates are achieved by executing a single routine concurrently on many pixels at once. While this is still the dominant use-case, frameworks like OpenCL [27] or CUDA (Compute Unified Device Architecture) [16] offer

an API to use the available hardware for non-graphical applications. This approach is called general purpose GPU (GPGPU) computing.

The first graphics cards were build around a pipeline, where each stage offered a different fixed operation with configurable parameters [20]. 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 [6]. 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. Further, the capacities of a stage were fixed and could not be shifted depending on the algorithm. Eventually, the pipelines were replaced by data-parallel programable units to achieve an overall better workload and more flexibility [26]. 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.

By now, this architecture can be found in non-GPU hardware as well. Accelerators with the sole purpose of data-parallel computing are available on the market. While some have a more similar architecture to GPUs, for example the Nvidia Tesla devices [24], others are build closer to x86 machines, most prominently the Intel Xeon Phi coprocessors [14]. Both have many more cores than available CPUs and require special programming models to make optimal use of their processing power.

Naturally, algorithms that perform similar work on independent data benefit greatly from the parallelism offered by these architectures. Since most problems cannot be mapped solely to this category, calculations on accelerators are often combined with calculations on the CPU. This combination of several hardware architectures in a single application is called heterogenous computing.

2.3 OpenCL

The two major frameworks for GPGPU computing are CUDA (Compute Unified Device Architecture) [16]—a proprietary API by Nvidia—and OpenCL [27]—a standardized API. In our work, we focus on OpenCL, as it is vendor-independent and allows us to integrate a broad range of hardware. The OpenCL standard is developed by the OpenCL Working Group, a subgroup of the non-profit organization Khronos Group [29]. 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" [22].

A study by Fang et al. [11] examines the performance differences between OpenCL and CUDA. Their benchmarks

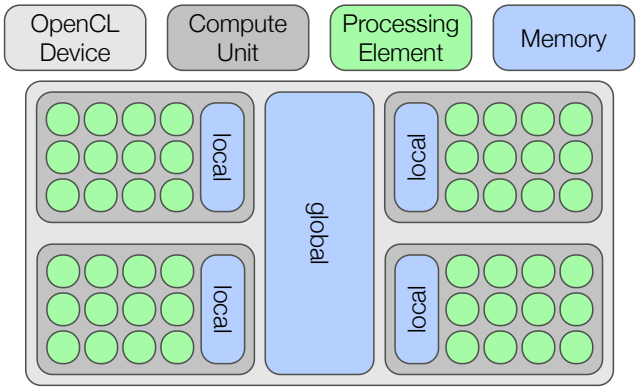


Figure 1. The OpenCL view on a computation device.

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 author explain the gap with differences in the programming model, optimizations, architecture and compiler. They continue to define a fair comparison that includes several steps such as individual optimizations and multiple kernel compilation steps.

Figure 1 depicts a computing device from the perspective of OpenCL. Each device is divided into compute units (CU), which are further divided into processing elements (PE) that perform the actual calculations. OpenCL defines four different memory regions, which may differ from the physical memory layout. The global memory is accessible by all PEs and has a constant memory region with read-only access. Each local memory region is shared by the PEs of a single CU. In addition, each PE has a private memory region which cannot be accessed by others.

Each OpenCL program consists of two parts. One part runs on the host, normally a CPU, and is called host program. The other part consists of any number of kernels that run on an OpenCL device. A kernel is a function written in an OpenCL-specific C dialect. OpenCL does not require a binary interface, as kernels can be compiled at runtime by the host program for a specific GPGPU device.

A 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 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 and allow the identification of work-items during the kernel execution. Further organization is achieved through work-groups. 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, work-items can be arranged in up to three dimensions inside a work-group. All items in a work-group run in parallel on a single CU. De-

pending on the available hardware, work-groups may run sequentially or in parallel.

The host program initializes data on the device, compiles kernels, and manages their execution. This requires a series of steps before running a kernel on an OpenCL device. Available device drivers offer an entry point in form of different platforms. These can be queried through the OpenCL API. Once a platform is chosen, all associated device IDs can be acquired. The next step is to create a context object for managing devices of the platform in use.

Communication with a device requires a command queue. The number of command queues per context or device is not limited, though a queue is associated with a single device. Multiple commands can be organized with events. Each command can generate an event which can than be passed to another command to define a dependency between them. Alternatively, OpenCL allows associating an event with a callback. In this way, an asynchronous workflow can be implemented.

Before a kernel—usually stored as source code in the host application—can run on a device, it needs to be compiled using the API of OpenCL. The compilation is then wrapped in a program object. Each program can compile multiple kernels at once and allows their retrieval by name. Running a kernel requires the transfer of its input argument to the target device, as the memory regions of host and GPGPU device are usually disjoint. OpenCL organizes chunks of memory as memory buffer objects that can be created independently and set as read-write, read-only or write-only. Once each argument is assigned to a buffer and the programmer has specified all dimensions in the index space, the kernel can be scheduled. The last step in this process is copying any produced results from the GPGPU device back to the host.

OpenCL offers reference counted handles for its components using the `cl_` prefix, e.g., a kernel is stored as a `cl_kernel`. The internal reference count needs to be managed manually. In a similar manner, functions are prefixed with `cl`, e.g., `clGetPlaformIDs`. Most API calls can be executed blocking as well as non-blocking.

The Khronos Group is actively working on advancing OpenCL. The next version of the specification is available as a provisional document since January 2015. In addition to OpenCL itself, the group supports projects that build upon or support OpenCL. SYCL (C++ Single-source Heterogeneous Programming for OpenCL) [19] aims to provide the same source code for the CPU and device part, compared to a separate code base for the OpenCL kernels. Since the code for all targets is written with C++ templates, it can be shared across platforms. However, the specification keeps to the familiar execution model from OpenCL and imposes the same restrictions to the SYCL device code as to OpenCL C.

2.4 Approaches to Heterogeneous Computing

As with multi-core machines, accelerators can be programmed through many different frameworks. The above-mentioned frameworks OpenCL and CUDA are the main stream solutions. They offer a lot of control at the price of an extensive API. Many libraries have emerged that use OpenCL or CUDA as a backend to offer a higher level API and implementations of often-used algorithms. Examples are Boost.Compute [1] or VexCL [2].

The projects Aparapi [3] and PyOpenCL [17] provide interfaces to write OpenCL kernels in their respective language, Java and Python. By avoiding the use of OpenCL C they ease the entrance to heterogeneous computing for developers not familiar with OpenCL. Having this level of abstraction further allows the execution of code on CPUs in case no suitable OpenCL devices is available. While Aparapi provides an interface similar to Java Threads, PyOpenCL relies on annotations to define which functions are offloaded. In contrast, OCCA [21] has the goal to provide portability and flexibility to developers. They contribute a uniform interface for programming OpenMP, CUDA and OpenCL. Writing the offloaded code in macros allows translation depending on the target platform at runtime. An extensible approach allows the addition of new languages in the future.

A pragma-based approach uses code annotations to specify which code should be parallelized by the compiler. A major advantage is the portability of existing code by adding the annotations to the offloaded code blocks. At the same time the developer has much less control over the execution and less potential for optimization. OpenACC [25] is a such standard. It supports data parallel computations distributed on many cores as well as vector operations. A comparison between OpenCL and OpenACC can be found in the work of Wienke et al. [31]. Although OpenCL showed much better performance in their test, the authors conclude that OpenACC opens the field to more programmers and will improve in performance over time.

Integrating GPU computing into the actor model is also explored by other scientists. For example, Harvey et al. [12] showed actors running OpenCL code as part of the actor based programming language Ensemble. By adding an additional compiler step, they allow the device code to be written in the same language as the rest of their code. This approach simplifies the development as it allows the use of language features such as multi-dimensional arrays. Further optimizations allow the language to keep messages sent between OpenCL actors on the device instead of copying it back and forth. The code used as the actors behavior still must be written to address the parallel nature of OpenCL devices. Their benchmarks compare OpenACC, Ensemble and native OpenCL. In most cases Ensemble performs close to OpenCL while OpenACC lacks behind in performance.

[1] `https://github.com/boostorg/compute` (August 2015)

[2] `https://github.com/ddemidov/vexcl` (August 2015)

3. The Design of OpenCL Actors

We are now ready to introduce our approach in detail, discuss its rationales and implementation challenges along with its benefits as well as its limitations

3.1 Design Goals and Rationales

OpenCL is a widely deployed standard containing a programming language (OpenCL C) and a management API. Unlike other approaches such as OCCA [21], CAF does neither attempt to build a new language unifying CPU and GPGPU programming nor to abstract over multiple GPGPU frameworks. Instead, our approach allows programmers to implement actors using data-parallel kernels written in OpenCL C without contributing any boilerplate code. Hence, CAF is hiding the management complexity of OpenCL. We want to keep CAF easy to use in practice and confine tools to a standard-compliant C++ compiler with available OpenCL drivers. In particular, we do not require a code generator or compiler extensions.

A possible design option would be to specify a domain-specific language (DSL) for GPGPU programming in C++ based on template expressions. Such a DSL essentially allows a framework to traverse the abstract syntax tree (AST) generated by C++ in order to enable lazy evaluation or to generate output in a different language such as OpenCL C. However, programmers would need to learn this DSL in the same way they need to learn OpenCL C. Further, we assume GPGPU programmers to have some familiarity or experience with OpenCL or CUDA. Introducing a new language would thus increase the entry barrier instead of lowering it. Also, this would force users to re-write existing OpenCL kernels. For this reason, we chose to support OpenCL C directly.

Our central goals for the design of OpenCL actors are (1) hiding complexity of OpenCL management and (2) seamless integration into CAF with respect to access transparency as well as location transparency.

Hiding Complexity The OpenCL API is a low-level interface written in C with a style that does not integrate well with modern C++. Although OpenCL does offer a C++ header that wraps the C API, it shows inconsistencies when handling errors and requires repetitive manual steps. The initialization of OpenCL devices, the compilation and management of kernels as well as the asynchronous events generated by OpenCL can and should be handled by the framework rather than by the programmer. Only relevant decisions shall be left to the user and remain on a much higher level of abstraction than is offered by OpenCL.

Seamless Integration OpenCL actors must use the same handle type as actors running on the CPU and implement the same semantics. This is required to make both kinds of actors interchangeable and hide the physical deployment at runtime. Further, using the same handle type enables the runtime to use existing abstraction mechanism for network-transparency, monitoring, and error propagation. Additionally, the API for creating OpenCL actors should follow a conformal design, i.e., the OpenCL abstraction should provide a function that is similar to `spawn`.

3.2 Core Approach to the Integration of OpenCL

The asynchronous API of OpenCL maps well to the asynchronous message passing found in actor systems. For starting a computation, programmers enqueue a task to the command queue of OpenCL and register a callback function that is invoked once the result has been produced. This naturally fits actor messaging, whereas the queue management is done implicitly and a response message is generated instead of relying on user-provided callbacks.

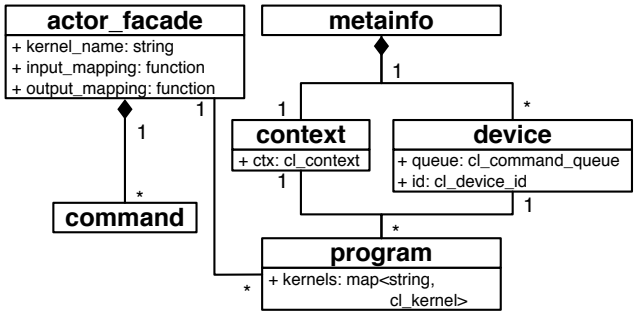


Figure 2. Class diagram for the OpenCL integration.

OpenCL actors introduce easy access to heterogeneous computing within the context of CAF actors. Our main building block is the class `actor_facade` which is shown in Figure 2. The facade wraps the kernel execution on OpenCL devices and provides a message passing interface in form of an actor. For this purpose, the class implements all required interfaces to communicate with other components of CAF (omitted in the diagram for brevity). Whenever a facade receives a message, it creates a `command` which preserves the original context of a message, schedules execution of the kernel and finally produces a result message. The remaining classes implement the bookkeeping required by OpenCL.

- `metainfo` is a singleton that performs device discovery lazily on first access, creates one command queue per device, and provides the global OpenCL context;
- `context` wraps a `cl_context` which stores OpenCL-internal management data;
- `device` describes an OpenCL device and provides access to its command queue;
- `program` stores compiled OpenCL kernels and provides a mapping from kernel names to objects.

CAF handles all steps of the OpenCL workflow automatically, but allows for fine-tuning of key aspects. For example, developers can simply provide source code and names for kernels and have CAF create a `program` automatically by selecting a device and compiling the sources. Particularly on

host systems with multiple co-processors, programmers may wish to query the `metainfo` object for accessible devices manually and explicitly create a `program` object by providing a device ID, source code, kernel names, and compiler options.

3.3 Use Case for OpenCL Actors

We illustrate our concepts and give source code examples referring to the use case of multiplying square matrices. This problem is a very good fit and a common use case for this programming model as each index of the result matrix can be calculated independently.

Listing 1. OpenCL kernel to multiply two square matrices.

```
1  constexpr const char* name = "m_mult";
2  constexpr const char* source = R"__(
3  __kernel void
4  m_mult(__global float* matrix1,
5         __global float* matrix2,
6         __global float* output) {
7    size_t size = get_global_size(0);
8    size_t x = get_global_id(0);
9    size_t y = get_global_id(1);
10   float result = 0;
11   for (size_t idx=0; idx<size; ++idx) {
12     result += matrix1[idx + y * size]
13             * matrix2[x + idx * size];
14   }
15   output[x+y*size] = result;
16 })__";
```

Listing 1 shows an OpenCL kernel for multiplying two square matrices stored as string in the variable `source`. Additionally, the variable `name` stores the in-source name of the function implementing the kernel. OpenCL requires all kernels to return **`void`** and use the prefix **`__kernel`**. The first two arguments to the function `m_mult` are two input matrices and the last argument is the result. All matrices are placed in the global memory region to be accessible by all work-items (GPU cores). Since OpenCL does not support multi-dimensional arrays, the matrices are represented as one-dimensional arrays and the position is calculated from the x and y coordinate. At runtime, each instruction will run in parallel on multiple GPU cores but use different memory segments (single instruction, multiple data) identified by the function `get_global_id`. In this example, we use two dimensions, which can be queried as index 0 for the x axis and 1 for the y axis. Since we multiply square matrices `get_global_size` returns the same value for both axes.

3.4 Programming Interface

While the OpenCL interface can be translated to actor-like communication in a straightforward way, generating the behavior of the actor is more complex. Since OpenCL source code is compiled at runtime from strings, the C++ compiler needs additional information regarding input and output types.

OpenCL actors are created using a variant of `spawn`, specifically the function `spawn_cl`. The execution of a kernel requires configuration parameters like the number of work-items to execute it. Listing 2 illustrates how to create an actor for the kernel shown in Listing 1.

Listing 2. Spawning OpenCL actors.

```
1  using fvec = std::vector<float>;
2  constexpr size_t mx_dim = 1024;
3  auto worker = spawn_cl(
4    source, name,
5    spawn_config{dim_vec{mx_dim, mx_dim}},
6    in<fvec>{}, in<fvec>{}, out<fvec>{});
7  auto m = create_matrix(mx_dim * mx_dim);
8  scoped_actor self;
9  self->sync_send(worker, m, m).await(
10   [](const fvec& result) {
11     print_as_matrix(result);
12   });
```

The first two arguments to `spawn_cl` are strings containing source code and kernel name. CAF will automatically create a `program` object from this source code. For more configuration options, programmers can also create a `program` manually and pass it as the first argument instead. The third argument—the spawn configuration—describes the distribution of work-items in three dimensions. A spawn configuration always contains the global dimensions and optionally offset for the global IDs and local dimensions (to override defaults and fine-tune work-groups in OpenCL). The dimensions are passed as instances of `dim_vec`, which is a tuple consisting of either one, two, or three integers. Our example creates one work-item for each index, i.e., $matrix\_size \cdot matrix\_size$ items, meaning that one GPU core computes one element of the result matrix at a time.

The remaining arguments must represent the kernel signature as list of `in`, `out` and `in_out` declarations. This type information allows CAF to automatically generate a pattern for extracting data from messages and to manage OpenCL buffers. While input arguments are provided by the user, storage for output buffers must be allocated by CAF. By default, CAF assumes output buffers to have a size equal to the number of work-items. This default can be overridden by passing a user-defined function to an `out` declaration which calculates the output size depending on the inputs at runtime.

In our example, the kernel expects two input arguments and one output argument, all represented by one-dimensional dynamic arrays of floating point numbers—in C++ named `std::vector<float>`. In line 11 of Listing 2, we send two input matrices to the OpenCL actor using `sync_send`. The message handler for the result in line 12 awaits the resulting matrix and prints it.

Optionally, programmers can pass two conversion function following the `spawn_config` argument as shown in Listing 3. The first function is then responsible for extracting

data from a message while the second function converts the output generated by the kernel to a response message. This mapping gives users full control over the message passing interface of the resulting actor. Per default, these functions are generated by CAF. A message is then matched against all `in` and `in_out` kernel arguments, while the output message is generated from all `in_out` and `out` arguments.

Listing 3. Pre- and post-processing in OpenCL actors.

```
1  template <size_t Size>
2  class square_matrix { /* ... */ };
3  using fvec = vector<float>;
4  constexpr size_t mx_dim = 1024;
5  using mx = square_matrix<mx_dim>;
6  auto preprocess = [](message& msg)
7                    -> optional<message> {
8    return msg.apply([](mx& x, mx& y) {
9      return make_message(move(x.data()),
10                         move(y.data()));
11 });};
12 auto postprocess = [] (fvec& res)
13                     -> message {
14   return make_message(mx{move(res)});
15 };
16 auto worker = spawn_cl(
17   kernel_source, kernel_name,
18   spawn_config{dim_vec{mx_dim, mx_dim}},
19   preprocess, postprocess,
20   in<fvec>{}, in<fvec>{}, out<fvec>{});
```

The example in Listing 3 introduces the class `square_matrix`, which is used for message passing. Since OpenCL does not allow custom data types, the OpenCL actor needs to convert the matrix to a one-dimensional **`float`** array before copying data to the GPU and do the opposite after receiving the result from OpenCL. This pattern matching step is modeled by the two functions `preprocess`, which converts two input matrices to arrays, and `postprocess`, which maps a computed array to a matrix. It is worth mentioning that the `postprocess` function can also be used to send messages to other actors using the computed result. Further, automatically sending a response message can be suppressed by returning a default-constructed message.

In addition to making use of a `postprocess` function, programmers can also use a client-sided approach for redirecting messages using `send_as`. This function allows programmers to send an asynchronous message and specify which actor should receive the response. Transparent redirection of messages is a feature of CAF and not limited to OpenCL actors.

3.5 Design Discussion

CAF achieves a very high-level of abstraction in comparison to the management API provided by OpenCL. Only key decisions such as the work-item distribution is required by the user. The OpenCL device binding for a kernel defaults to the first discovered device, but can be set by the user optionally.

The OpenCL actors presented in this section introduce data-parallel intra-actor concurrency to CAF. The behavior of an OpenCL actor consists of three parts: (1) a preprocessing function that pattern-matches input messages and forwards extracted data to OpenCL, (2) a data-parallel kernel that runs on an OpenCL device, and (3) a post-processing function that finalizes the message processing step and per default converts data produced by the kernel to a response message. Since the data-parallel kernel is running in a separate address space and can only use the limited instruction set provided by OpenCL C, sending messages or spawning new actors from OpenCL C directly cannot be achieved. However, the pre- and post-processing functions run on the CPU and allow programmers to spawn more actors and send additional messages in the common way. These two functions can be automatically generated for convenience by deriving all message types from the signature of a kernel.

Transparent message passing and error handling are achieved in our design by mapping the mailbox of an actor to a command queue of OpenCL. From the perspective of the runtime system of CAF, an OpenCL actor is not distinguishable from any other actor since it implements the same interfaces as actors running on the CPU. With the `spawn_cl` function, we provide an interface for the creation process of actors that hides most complexity while still granting access to all performance-relevant configuration options via optional parameters.

Once created, the actor handle can be used and addressed independent of its location. The creation process itself has its limitations, though. OpenCL is available for GPUs and dedicated accelerators as well as CPUs. This suggests to run OpenCL actors on the CPU if no other devices are available. While this is conceptually possible, device drivers commonly deployed do not support code compilation for the CPU. Another problem to consider is the workload caused by an OpenCL actor running on the CPU. It is not scheduled with other actors, but competes for the same resources. Alternatively, a single actor could have two implementations, one in OpenCL and one in regular C++. CAF could then choose the implementation that promises the best performance.

Dynamic behavior of OpenCL actors could be emulated by allocating state on the GPU, which is then passed to each kernel invocation. This would require a per-actor scheduling to guarantee sequential kernel invocations in order to avoid race conditions on the state. Per default, OpenCL tries to execute multiple executions of the same kernel in parallel on supported devices for optimizing performance. Also, OpenCL devices have very limited RAM resources compared to the host system. Hence, the number of stateful OpenCL actors must remain small. Exploring this design space is part of ongoing and future work.

An advanced aspect is scheduling kernels across multiple similar devices. To enqueue kernels for concurrent execution, a scheduler needs to keep track of the available resources, such as processing elements and memory, as these informations are not offered by OpenCL at runtime. Another aspect is scheduling kernels across different hardware. Depending on the target device, a kernel must be configured differently to reach optimal performance.

4. Evaluation

We have implemented four benchmark programs to systematically measure runtime characteristics and overhead introduced by our OpenCL wrapper.

The first benchmark compares the creation time of OpenCL actors to the event-based actors of CAF. Our next two benchmarks examine the overhead we induce compared to manually using the OpenCL API. Here, we take a look at single calculation before comparing our implementation against an optimized scenario. Our final benchmark examines the scalability in heterogeneous setups by stepwise transferring workload to a GPU and a coprocessor.

The first benchmarks were performed on a Late 2013 iMac with a 3.5 GHz Intel Core i7 running OS X and OpenCL version 1.2. The GPU is a NVIDIA GeForce GTX 780M GPU with 4096 MB memory. The last benchmarks on scalability use a machine with two twelve-core Intel Xeon CPUs clocked at 2.5 GHz equipped with a Tesla C2075 GPU as well as a Xeon Phi 5110P coprocessor. The server runs Linux and uses the graphics drivers provided by Nvidia and the Intel OpenCL Runtime 14.2.

4.1 Spawn Time

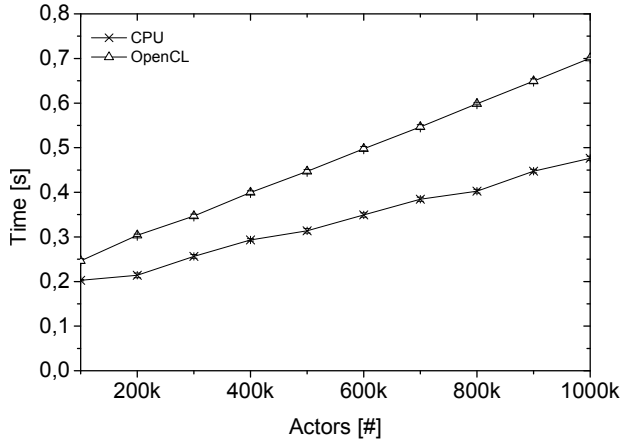


Figure 3. Comparison of the wall-clock time required to spawn OpenCL and event-based actors.

Our first benchmark focuses on the time to instantiate OpenCL actors. The creation of actors is traditionally a lightweight operation. We expect the creation of OpenCL actors to be more heavy weight than the creation of other actors in CAF. Still, we want to examine if the OpenCL actor can be created for short calculations as well as for longer ones.

We compare the creation time of OpenCL actors to that of event-based actors. 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 response.

The time measured is the wall clock time required to spawn an increasing number of actors. This includes the time required to initialize the runtime environment. To provide an equal setup, we spawn the event-based actors with the `lazy_init` flag. It prevents them from being scheduled for small initialization tasks unless they receive a message, as is the case with OpenCL actors.

Figure 3 depicts the wall-clock runtime in seconds as a function of the number of spawned actors. It plots the mean of 50 runs with error bars showing the 95 % confidence interval. In all cases the error bars are barely visible. Both implementations show a linear dependency with minor growth. However, event-based actors take less time than OpenCL actors and exhibit a smaller slope. The difference in slope indicates a longer spawn time for each individual OpenCl actor. Similar slopes with a constant distance would have indicated a similar creation time with longer initialization time of the runtime.

Compared to the time required for a simple calculation, the creation time is reasonably small. It is worth mentioning that OpenCL actors are parallelized internally by OpenCL. They are not created as frequently as event-based actors. Hence, creation time is usually less important.

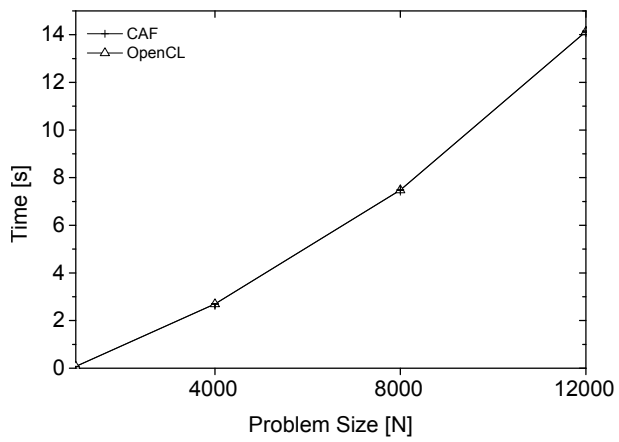
4.2 Runtime Overhead of Actors Over Native OpenCL Programming

Our second benchmark measures the overhead induced by our actor approach compared to the native API of OpenCL. While the OpenCL actor uses the OpenCL API internally, it performs additional steps such as the setup of the OpenCL environment and the actor creation. This benchmark quantifies the overhead added by message passing and wrapping the OpenCL API.

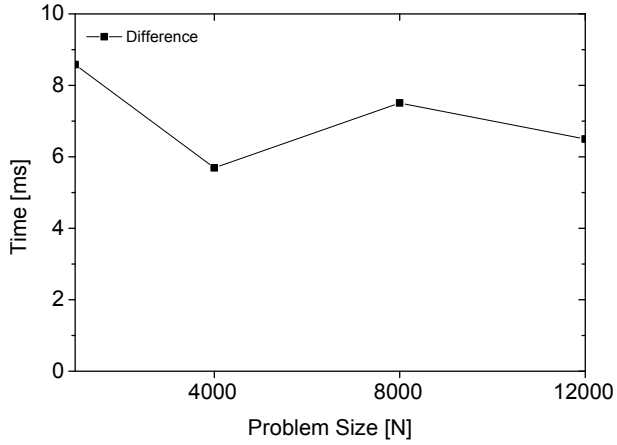
It implements a program that executes a simple task on a GPU using an OpenCL actor. In this case, the benchmark kernel calculates the product of two $N \cdot N$ matrices. We sent the actor matrices with 1000, 4000, 8000 and 12000 as values for N. The increase in problem size should test for a correlation between the message size and the overhead.

Two measurements are of interest in this case. First, the duration required for the whole calculation, from sending the message to receiving the answer. Second, the time between enqueuing the kernel until OpenCL invokes the callback, which includes data transfer as well as the kernel execution. Ideally, both times should be nearly equal.

Figure 4(a) depicts the runtime in seconds as a function of the problem size N. Each value is the mean of 50 runs, plot-



(a) Time between sending and receiving messages in CAF compared to the required by OpenCL.



(b) The difference between the graphs in (a).

Figure 4. Matrix multiplication of $N \cdot N$ matrices.

ted with the 95 % confidence interval. The total calculation time ranges from 0.07 s up to 14.1 s. We have also plotted the time difference separately in Figure 4(b) since the two lines in Figure 4(a) are not distinguishable. The difference between the measured values ranges between 5.7 ms and 8.6 ms. No discernible slope can be observed in the graph and the measurements fluctuate independently of the problem size.

The results of this measurement clearly show a negligible overhead that does not depend on the problem size. Hence, our high level interface can be used at a very low cost.

4.3 Baseline Comparison

The previous benchmark examines the overhead for a single calculation by comparing the runtime distribution between CAF and OpenCL. In this benchmark we want to compare the performance when calculating a sequence of independent tasks. Two $1000 \cdot 1000$ matrices are multiplied with an increasing number of iterations, starting at 1000 and increasing by 1000 in each step up to 10000. The environment is only initialized once and the calculations are preformed sequentially. For CAF, an actor sends a new message when it receives the results of the last calculation. In comparison, the native OpenCL implementation initiates the next calculation as part of the callback. Both programs use the same kernel for the multiplication. We avoid simultaneous kernel executions as we want to examine the overhead in our framework.

Figure 5 displays the wall-clock time as a function of the iterations performed. We plotted the average of 10 measurements as well as a 95% confidence interval. Since we use the OpenCL API within CAF, it is not possible to achieve a better performance than OpenCL itself. The OpenCL graph is the baseline we aim for with our performance. Both implementations exhibit linear growth. However, the native OpenCL implementation has a smaller slope and the runtime difference between the programs increases. This indicates a consistent overhead required for the message passing compared to the direct API usage. The relative performance difference is 8.3 % for 1000 iterations and slightly decreases to 7.4 % at 10000 iterations.

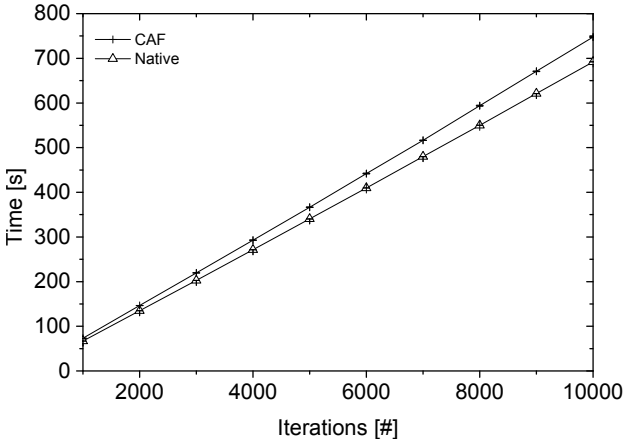
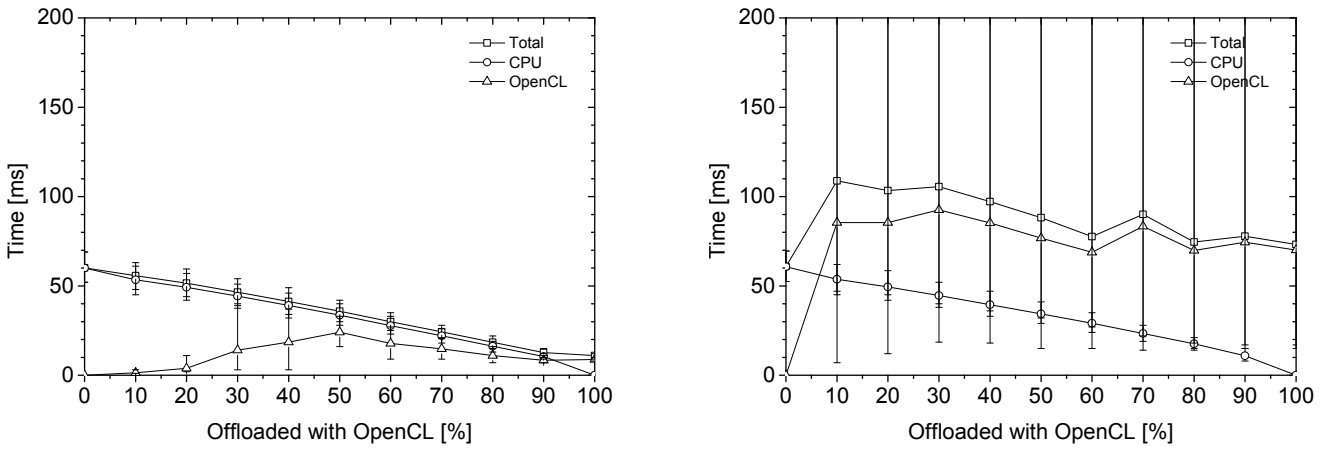


Figure 5. The runtime of multiple independent tasks in CAF compared to native OpenCL.

It is worth mentioning that this micro benchmark is looking at a minimal baseline that is not a realistic application scenario. A program using OpenCL will need to include some synchronization to pass GPU-computed results to the CPU and generate the next task for the GPU. Hence, a native application will not meet the baseline simply because it uses the OpenCL API directly.

4.4 Scaling Behavior in a Heterogeneous Setup

Our last benchmark focuses on the scalability of our heterogeneous computing approach by incrementally shifting work from the CPU to an OpenCL device. OpenCL distinguishes



(a) Mandelbrot on the Tesla.

(b) Mandelbrot on the Xeon Phi.

Figure 6. Moving a small workload to OpenCL devices.

between CPU, GPU and accelerator devices. Our system includes the two mentioned device, an NVIDIA Tesla GPU and an Intel Xeon Phi accelerator. The difference between a GPU and an accelerator is that GPUs are traditionally used for 3D APIs such as OpenGL or DirectX, while accelerators are dedicated for offloading computations from the host. The Xeon Phi features an architecture based on x86 processors, although not a compatible one, and differs greatly from the architecture of the Tesla GPU. It consists of 60 cores with 512 bit vector registers and 4 threads each, totaling to up to 240 threads.

We use the calculation of a Mandelbrot set in the benchmark, as the workload can be easily divided into many independent tasks. The problem is a cut from the inner part of a Mandelbrot set that has a balanced processing complexity for the entire image. The workload is offloaded in 11 steps, starting with 0 % on the coprocessor and increasing by 10 % in each step up to 100 %. Each computed image of the Mandelbrot set represents the area of $[-0.5 - 0.7375i, 0.1 - 0.1375i]$. Our measurements include two different workloads, a resolution of $1920 \cdot 1080$ pixels in Figure 6 and a resolution of $16000 \cdot 16000$ in Figure 7, both measured with 100 iterations. In addition, we increased the number of iterations to 1000 for the larger workload to further examine the scaling behavior.

Figure 6 depicts the runtimes in milliseconds as functions of the problem fraction offloaded. The problem is offloaded to the Tesla in Figure 6(a) and to the Xeon Phi in Figure 6(b). Each graph depicts the runtime for the CPU and OpenCL device calculations separately, i.e., the time between starting all actors and their termination. Since calculations are performed in parallel, the total runtime is not a sum of the separate runtimes, but measured independently.

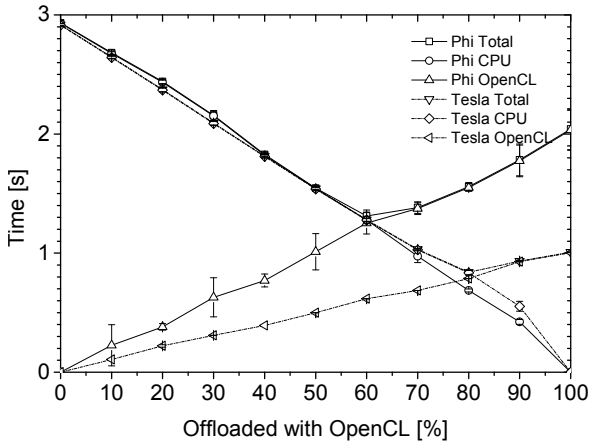
The problem plotted in Figure 6(a) exhibits excellent scalability. The runtime declines until the workload is completely offloaded to the GPU. While the CPU runtime is lower than the total runtime on average, it takes longer to calculate 10 % of the problem on the CPU than is needed to calculate 100 % on the GPU. As a result, the lower bound is the time required to process the complete workload on the GPU.

In contrast, Figure 6(b) reveals a measurable overhead. While the CPU runtime declines steadily, the runtime measured for OpenCL fluctuates heavily and the total execution time doubles when offloading 10 % of work to the Phi. Even when running 100 % of the problem size on the Phi, the computation is still slower than the initially measured 60 milliseconds for the CPU-only setup. The initial cost of offloading computations to the Phi are not amortized by faster, parallel computations on the accelerator device. It is worth mentioning that we did not optimize the OpenCL kernel for the Phi, which might result in suboptimal performance on this device.

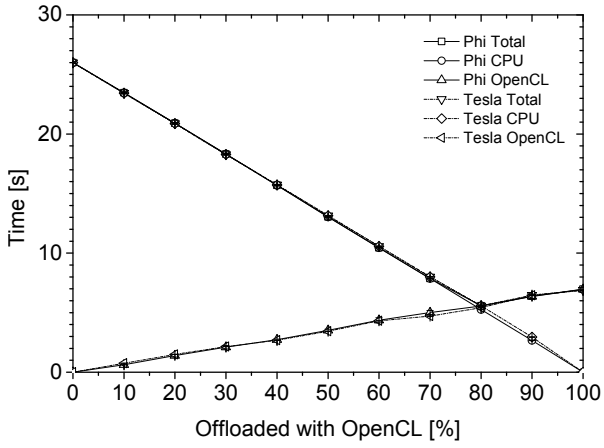
In summary, these experiments reveal excellent scalability of programming GPUs with CAF actors. However, offloading work to the Xeon Phi is not worth it with regard to this problem size. Since the performance of OpenCL applications largely depends on the driver implementation and configuration, it is left to future work to examine the Phi results in more detail.

Figure 7 shows the runtime in milliseconds as a function of the offloaded problem in % for a larger Mandelbrot image. We have increased the number of pixels from $1920 \cdot 1080$ to $16000 \cdot 16000$. The larger image drastically increases the computation time on the device to offset the initial cost of offloading computations. We have run the benchmark using 100 iterations and 1000 iterations per pixel.

Figure 7(a) depicts the smaller measurements with 100 iterations for both the Tesla and the Xeon Phi. In difference to the previous benchmark in Figure 6(a), the best performance



(a) Calculation with a 100 iterations.



(b) Calculation with a 1000 iterations.

Figure 7. Moving a large workload to OpenCL devices.

is achieved at around 80 % on the GPU and around 60 % on the accelerator. Since the initial cost of offloading the computation is smaller in comparison to the overall runtime, the Xeon Phi achieves drastically better performance as shown in Figure 6(b), but does not reach the performance of the Tesla.

Finally, the measurements with 1000 iterations are depicted in Figure 7(b). Here, the Phi and Tesla perform equally well. Since this setup has the same data rate as before but an increased runtime on the device, it becomes evident that the data transport to the Phi did hinder better results in the previous benchmarks. Hence, this accelerator (with current drivers) is best suited for problems of small data size but large computation demands.

In a naive approach, we simply transferred a problem from the Tesla to the Phi. This proved to be inefficient for small problems, but improved with an increase in problem size. As should be noted again, optimizing kernels and configurations for the Phi may improve its performance for smaller problems.

5. Conclusions and Outlook

Integrating GPGPU computing into an application can increase its performance by orders of magnitudes. This is true on all scales from mobiles to server systems. The challenge of integrating GPGPU devices into applications, though, is left to a programmer, who is faced with an ever-growing complexity of hardware architectures and APIs.

The actor model is an important concept for taming the complexity of parallel and concurrent systems and the task-oriented work flow of actors fits the work flow of GPGPU computing very well. The present work on OpenCL actors shows that an intelligent actor runtime can manage GPGPU devices autonomously while inducing minimal performance overhead. Supporting OpenCL as first-class implementation option in CAF further broadens the scope of our native actor system by introducing data-parallel intra-actor concurrency.

Our presented implementation of OpenCL actors is based on OpenCL 1.1. This version is available across Intel, NVIDIA, and AMD drivers. Our directions for future development fall into three categories: (1) improve performance of device-local actor-to-actor communication by keeping data on the GPU in pipelining scenarios, (2) explore how OpenCL actors can hold state without enabling race conditions on the state, e.g., by developing a per-actor scheduling ensuring sequential kernel execution, and (3) improve scheduling by load balance across multiple OpenCL devices both locally and in a network.

Acknowledgments

The authors would like to thank Marian Triebe for implementing benchmarks, testing, and bugfixing, as well as the anonymous reviewers for helping to sharpen the discussion on OpenCL actors. We further want to thank Matthias Vallentin and the iNET working group for vivid discussions and inspiring suggestions. Funding by the German Federal Ministry of Education and Research within the projects ScaleCast and PEEROSKOP is gratefully acknowledged.

References

[1] G. Agha. Actors: A Model of Concurrent Computation In Distributed Systems. Technical Report 844, MIT, Cambridge, MA, USA, 1986.

[2] G. Agha, I. A. Mason, S. Smith, and C. Talcott. Towards a Theory of Actor Computation. In *Proceedings of CONCUR*, volume 630 of *LNCS*, pages 565–579, Heidelberg, 1992. Springer-Verlag.

[3] AMD. Aparapi. http://aparapi.github.io, 2015.

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

[5] J. Armstrong. 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, 2007. ACM.

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

[7] CAPS. Cray Inc., NVIDIA and the Portland Group. The OpenACC Application Programming Interface, v1.0, November 2011.

[8] D. Charousset, R. Hiesgen, and T. C. Schmidt. CAF - The C++ Actor Framework for Scalable and Resource-efficient Applications. In *Proc. of the 5th ACM SIGPLAN Conf. on Systems, Programming, and Applications (SPLASH '14), Workshop AGERE!*, New York, NY, USA, Oct. 2014. ACM.

[9] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science and Engineering, IEEE*, 5(1):46–55, 1998.

[10] T. Desell and C. A. Varela. SALSA Lite: A Hash-Based Actor Runtime for Efficient Local Concurrency. In G. Agha, A. Igarashi, N. Kobayashi, H. Masuhara, S. Matsuoka, E. Shibayama, and K. Taura, editors, *Concurrent Objects and Beyond*, volume 8665 of *Lecture Notes in Computer Science*, pages 144–166. Springer Berlin Heidelberg, 2014. ISBN 978-3-662-44470-2.

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

[12] P. Harvey, K. Hentschel, and J. Sventek. Parallel Programming in Actor-Based Applications via OpenCL. In *The 16th International Conference on Middleware*, New York, NY, USA, Dec. 2015. ACM.

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

[14] Intel. Intel Xeon PhiTM Coprocessor x100 Product Family Datasheet. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-datasheet.html, April 2015.

[15] L. V. Kale and S. Krishnan. Charm++: Parallel programming with message-driven objects. *Parallel Programming using C++*, pages 175–213, 1996.

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

[17] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012.

[18] S. J. Krieder, J. M. Wozniak, T. Armstrong, M. Wilde, D. S. Katz, B. Grimmer, I. T. Foster, and I. Raicu. Design and Evaluation of the Gemtc Framework for GPU-enabled Many-task Computing. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 153–164, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2749-7.

[19] Lee Howes and Maria Rovatsou. *SYCL integrates OpenCL devices with modern C++*. Khronos Group, 2015.

[20] E. Lindholm, M. J. Kilgard, and H. Moreton. 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, 2001. ACM.

[21] D. S. Medina, A. St-Cyr, and T. Warburton. OCCA: A unified approach to multi-threading languages. *ArXiv e-prints*, March 2014.

[22] A. Munshi. *The OpenCL Specification*. Khronos OpenCL Working Group, Khronos Group, 2012. URL `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf,(13.09.2013)`. Version 1.2, Revision 19.

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

[24] NVIDIA. *Tesla C2075 Computing Processor Board (Board Specification)*, September 2011.

[25] OpenACC-standard.org. *The OpenACC Application Programming Interface*, 2013.

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

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

[28] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *Proceedings of the 22nd ECOOP*, volume 5142 of *LNCS*, pages 104–128, Berlin, Heidelberg, 2008. Springer-Verlag.

[29] The Khronos Group. The Khronos Group. http://www.khronos.org/, August 2015.

[30] Typesafe Inc. Akka. `http://akka.io`, March 2012.

[31] Wienke, Sandra and Springer, Paul and Terboven, Christian and an Mey, Dieter. OpenACC: First Experiences with Real-world Applications. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 859–870, Berlin, Heidelberg, 2012. Springer-Verlag.