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

Raphael Hiesgen, Dominik Charousset, Thomas C. Schmidt
{raphael.hiesgen,dominik.charousset,t.schmidt}@haw-hamburg.de

iNET RG, Department of Computer Science
Hamburg University of Applied Sciences

October 2015, AGERE!@SPLASH
Motivation

- Many-core devices available for general-purpose computation
  - Specialized components can vastly outperform CPUs
  - Available on servers, desktops and mobile devices
  - GPUs, dedicated accelerators, ...
- Require specialized frameworks and programming models
  - Often low-level APIs
  - Management overhead
  - Specialized syntax or language

- Can we use the actor model to abstract over heterogeneous computing?
Motivation

- Many-core devices available for general-purpose computation
  - Specialized components can vastly outperform CPUs
  - Available on servers, desktops and mobile devices
  - GPUs, dedicated accelerators, ...
- Require specialized frameworks and programming models
  - Often low-level APIs
  - Management overhead
  - Specialized syntax or language

- Can we use the actor model to abstract over heterogeneous computing?
Outline

1. Heterogeneous Computing
2. The C++ Actor Framework (CAF)
3. The OpenCL Actor
4. Performance Evaluation
5. Conclusion & Outlook
OpenCL
Our Starting Point

- Vendor-independent standard
- Maintained by the Khronos Group
- Supports a wide range of hardware
  - GPUs, CPUs, accelerators
  - Simplified device model
- Concurrent execution of “kernels”
  - Executed in 3D index space
  - Command queues for device interaction
  - Offers asynchronous API

The OpenCL device model.
CAF
The C++ Actor Framework

- Actor library written in C++11
  - Low memory footprint
  - Fast, lock-free mailbox implementation
  - Work-stealing scheduler
  - Type-safe message passing
- Focus on scalability
  - Up to multi-core machines
  - Down to embedded devices with RIOT
- Working on Runtime Inspection and Configuration tools
Map the OpenCL workflow to actor message passing

- Intra-actor concurrency
- Many kernel instances executed in parallel
- Kernels may run in parallel on the device (if supported)

Hide complexity

- OpenCL provides a low-level interface
- Similar steps in each program
- Management of OpenCL setup and device initialization

Integrate seamlessly

- Use the same handle types as other actors
- Hide physical deployment
- Network transparency, monitoring and error propagation
Use Case

```cpp
constexpr const char* source = R"__(
__kernel void m_mult(__global float* matrix1,
                   __global float* matrix2,
                   __global float* output) {
    size_t size = get_global_size(0);
    size_t x = get_global_id(0);
    size_t y = get_global_id(1);
    float result = 0;
    for (size_t idx = 0; idx < size; ++idx) {
        result += matrix1[idx + y * size] * matrix2[x + idx * size];
    }
    output[x + y * size] = result;
})__";
constexpr const char* name = "m_mult";
```
constexpr const char* source = R"__(
  __kernel void m_mult(__global float* matrix1,
                       __global float* matrix2,
                       __global float* output) {
    size_t size = get_global_size(0);
    size_t x = get_global_id(0);
    size_t y = get_global_id(1);

    float result = 0;
    for (size_t idx = 0; idx < size; ++idx) {
      result += matrix1[idx + y * size] * matrix2[x + idx * size];
    }
    output[x + y * size] = result;
  })__";
constexpr const char* name = "m_mult";
Use Case

```
constexpr const char* source = R"__(
  __kernel void m_mult(__global float* matrix1,
                       __global float* matrix2,
                       __global float* output) {
    size_t size = get_global_size(0);
    size_t x = get_global_id(0);
    size_t y = get_global_id(1);
    float result = 0;
    for (size_t idx = 0; idx < size; ++idx) {
        result += matrix1[idx + y * size] * matrix2[x + idx * size];
    }
    output[x + y * size] = result;
})__";

constexpr const char* name = "m_mult";
```

Specifies memory regions of arguments.
```c
constexpr const char* source = R"__(
  __kernel void m_mult(__global float* matrix1,
                       __global float* matrix2,
                       __global float* output) {
    size_t size = get_global_size(0);
    size_t x = get_global_id(0);
    size_t y = get_global_id(1);
    float result = 0;
    for (size_t idx = 0; idx < size; ++idx) {
      result += matrix1[idx + y * size] * matrix2[x + idx * size];
    }
    output[x + y * size] = result;
})__";
constexpr const char* name = "m_mult";
```

Identifies kernel instance in the index space.
Use Case

```c
constexpr const char* source = R"__(
  __kernel void m_mult(__global float* matrix1,
                       __global float* matrix2,
                       __global float* output) {
    size_t size = get_global_size(0);
    size_t x = get_global_id(0);
    size_t y = get_global_id(1);
    float result = 0;
    for (size_t idx = 0; idx < size; ++idx) {
      result += matrix1[idx + y * size] * matrix2[x + idx * size];
    }
    output[x + y * size] = result;
  })__";
constexpr const char* name = "m_mult";
```

Data-parallel assignment of results.
int main() {
    using fvec = std::vector<float>;
    constexpr size_t mx_dim = 1024;
    auto worker = spawn_cl(
        source, name,
        spawn_config{dim_vec{mx_dim, mx_dim}},
        in<fvec>{}, in<fvec>{}, out<fvec>{}
    );
    auto m = create_matrix(mx_dim * mx_dim);
    scoped_actor self;
    self->sync_send(worker, m, m).await(
        [](const fvec& result) {
            print_as_matrix(result);
        }
    );
}
int main() {
    using fvec = std::vector<float>;
    constexpr size_t mx_dim = 1024;
    auto worker = spawn_cl(
        source, name,
        spawn_config{dim_vec{mx_dim, mx_dim}},
        in<fvec>{}, in<fvec>{}, out<fvec>{}
    );
    auto m = create_matrix(mx_dim * mx_dim);
    scoped_actor self;
    self->sync_send(worker, m, m).await(
        [](const fvec& result) {
            print_as_matrix(result);
        }
    );
}

Function to create an OpenCL actor.
Requires OpenCL specific parameters
int main() {
    using fvec = std::vector<float>;
    constexpr size_t mx_dim = 1024;
    auto worker = spawn_cl(
        source, name,
        spawn_config{dim_vec{mx_dim, mx_dim}},
        in<fvec>{}, in<fvec>{}, out<fvec>{}
    );
    auto m = create_matrix(mx_dim * mx_dim);
    scoped_actor self;
    self->sync_send(worker, m, m).await(
        [](const fvec& result) {
            print_as_matrix(result);
        }
    );
}
int main() {
  using fvec = std::vector<float>;
  constexpr size_t mx_dim = 1024;
  auto worker = spawn_cl(
    source, name,
    spawn_config<dim_vec{mx_dim, mx_dim}>,
    in<fvec>{}, in<fvec>{}, out<fvec>{}
  );
  auto m = create_matrix(mx_dim * mx_dim);
  scoped_actor self;
  self->sync_send(worker, m, m).await(
    [](const fvec& result) {
      print_as_matrix(result);
    }
  );
}

Specifies index space for the execution.
```cpp
int main() {
    using fvec = std::vector<float>;
    constexpr size_t mx_dim = 1024;
    auto worker = spawn_cl(
        source, name,
        spawn_config{dim_vec{mx_dim, mx_dim}},
        in<fvec>{}, in<fvec>{}, out<fvec>{}
    );
    auto m = create_matrix(mx_dim * mx_dim);
    scoped_actor self;
    self->sync_send(worker, m, m).await(
        [](const fvec& result) {
            print_as_matrix(result);
        }
    );
}
```

Specifics the signature of the kernel, i.e., argument type, position and if it is input, output or both.
int main() {
    using fvec = std::vector<float>;
    constexpr size_t mx_dim = 1024;
    auto worker = spawn_cl(
        source, name,
        spawn_config{dim_vec{mx_dim, mx_dim}},
        in<fvec>{}, in<fvec>{}, out<fvec>{}
    );
    auto m = create_matrix(mx_dim * mx_dim);
    scoped_actor self;
    self->sync_send(worker, m, m).await(
        [](const fvec& result) {
            print_as_matrix(result);
        }
    );
}
Limitations

- Current limitations of OpenCL actors
  - No message passing from OpenCL context
  - Actor creation only from the CPU context
  - Behavior cannot be changed at runtime
  - OpenCL actors have no state

- Depends on available hardware & drivers
Baseline Benchmark

- Performance comparison to native OpenCL
  - Initialization only occurs once
  - Management performed in OpenCL callbacks
- Multiply two $1000 \times 1000$ matrices
  - Chain of independent calculations
  - Same code for kernel
  - No simultaneous executions
- Environment
  - Desktop with OS X (10.11)
  - NVIDIA GeForce GTX 780M GPU, OpenCL Version 1.2
  - Intel Core i7 clocked at 3.5 GHz
Results
Baseline Benchmark

- OpenCL runtime exhibits a smaller slope
- Indication of a consistent overhead for message passing
- Not reachable with a realistic application
Scaling Benchmark

- Scalability in a heterogeneous scenario
  - Examine relation to problem size
- Calculation of the Mandelbrot set
  - Easily dividable into independent tasks
  - Inner cut with a balanced processing complexity
  - Offload in steps of 10%, from 0% to 100%
- Environment
  - Server with Linux (kernel 3.19)
  - Nvidia Tesla C2075 GPU, OpenCL Version 1.1
  - Intel Xeon Phi 5110P, OpenCL Version 1.2
  - Two twelve-core Intel Xeon CPUs clocked at 2.5 GHz
Results: Small Problem

Scaling Benchmark

Small workload (1920 × 1080 pixels, 100 iterations) offloaded to a Tesla GPU (left) and a Xeon Phi accelerator (right).

- The Tesla exhibits excellent scaling behavior
- Calculating 10% on the CPU takes longer than 100% on the Tesla
- Xeon Phi shows large overhead
Results: Large Problems

Scaling Benchmark

Large workload (16,000 × 16,000 pixels) on the Phi for 100 (left) and 1000 iterations (right).

- Best performance no longer reached by offloading everything
Results: Large Problems

Scaling Benchmark

Large workload (16,000 × 16,000 pixels) on the Tesla and Phi for 100 (left) and 1000 iterations (right).

- Best performance no longer reached by offloading everything
- Performance of Phi and Tesla converge
Conclusion & Outlook

- Introducing OpenCL actors
  - Handles OpenCL management tasks
  - Small overhead compared to native OpenCL
  - Good scalability when offloading work
  - Further benchmarks in the paper

- A native implementation of the actor model: CAF
  - High level of abstraction without sacrificing performance
  - Small memory footprint & efficient program execution
  - Freely available & open source (BSD or Boost)

- Future directions
  - Improve communication between OpenCL actors
  - Explore how OpenCL actors can hold state
  - Load-balancing across local and remote devices
Thank you for your attention. Questions?

Developer blog:  http://actor-framework.org
Sources:        https://github.com/actor-framework/
iNET:          https://inet.cpt.haw-hamburg.de