Actors: the Ideal Abstraction for Programming Kernel-Based Concurrency

Paul Harvey[1], Kristian Hentschel[1], and Joseph Sventek[2]

[1] School of Computing Science, University of Glasgow, Glasgow, U.K.
[2] Dept of Computer and Information Science, University of Oregon, OR, USA

Abstract. GPU and multicore hardware architectures are commonly used in many different application areas to accelerate problem solutions relative to single CPU architectures. The typical approach to accessing these hardware architectures requires embedding logic into the programming language used to construct the application; the two primary forms of embedding are: calls to API routines to access the concurrent functionality, or pragmas providing concurrency hints to a language compiler such that particular blocks of code are targeted to the concurrent functionality. The former approach is verbose and semantically bankrupt, while the success of the latter approach is restricted to simple, static uses of the functionality.
Actor-based applications are constructed from independent, encapsulated actors that interact through strongly-typed channels. This paper presents a first attempt at using actors to program kernels targeted at such concurrent hardware. Besides the glove-like fit of a kernel to the actor abstraction, quantitative code analysis shows that actor-based kernels are always significantly simpler than API-based coding, and generally simpler than pragma-based coding. Additionally, performance measurements show that the overheads of actor-based kernels are commensurate to API-based kernels, and range from equivalent to vastly improved for pragma-based annotations, both for sample and real-world applications.

1 Introduction

Due to power consumption, heat dissipation, and clock propagation limits, modern hardware architectures are now designed with many, concurrent processing elements, as opposed to single processing elements with increasing clock rates; examples of such architectures include GPUs and multicore CPUs. These hardware platforms are designed to provide the user with multiple physical threads of execution, thus enabling many computations to occur simultaneously.

Software threads have traditionally been used to enable parallel execution, on CPU architectures. However, due to the different nature of GPU hardware architectures, a number of different programming techniques are used. OpenCL is a standardised programming framework available for the main GPU vendors (NVIDIA and AMD), as well as other parallel hardware architectures.

While the OpenCL API enables access to these architectures, there are three main limitations. Firstly, the user is required to write large amounts of boilerplate

code to create the OpenCL environment for a particular calculation. Secondly, the programming style requires explicit data movement between the host CPU and the OpenCL device; this requires manually flattening multi-dimensional arrays and structures of non-primitive types. Thirdly, the language and style used to program the device is often different from the programming language being used on the host. A similar argument can be made against the CUDA framework; since CUDA is only available on NVIDIA hardware, this work is focused on the more broadly applicable OpenCL.

OpenACC is a pragma-based approach to concurrent programming, where a developer explicitly annotates sections of code to be parallelised, as well as the data which should be moved between host and device. While this approach abstracts much of the boilerplate code of OpenCL, applying the simple annotations to single threaded code does not guarantee good performance and is not effective for all application classes.

We describe the first application of actor-based programming to kernel-based concurrency at the language level by including the OpenCL framework within the Ensemble programming language. We hypothesise that moving from low-level C code to a concurrent, shared-nothing, high-level actor programming model simplifies the use of OpenCL by providing appropriate structuring, thus enabling greater access to high-performance and heterogeneous computing. We demonstrate that applications written using actors exhibit less complexity via quantitative metrics compared to handwritten OpenCL in C (C-OpenCL), and that these applications run efficiently on different hardware platforms, with low overhead when compared to C-OpenCL and equivalent or better performance to OpenACC annotated C (C-OpenACC). This is shown for a number of different types of application, including a real-world document ranking example.

The remainder of the paper is arranged as follows: Section 2 describes the OpenCL framework, Section 3 gives an overview of the state of the art, Section 4 describes the Ensemble language, Section 5 discusses the combination of OpenCL and actors with the language and runtime, Section 6 describes experimental results, and Section 7 summarises the work and discusses future directions.

2 OpenCL

OpenCL is a programming framework for heterogeneous and parallel computing. It is standardised and is managed by the Khronos working group[3]. In OpenCL, users are required to think in terms of host and device code, where a host is a coordinator application on the CPU, and a device is an *accelerator*. An accelerator may be a CPU, GPU, FPGA, or co-processor such as the Xeon Phi [10].

2.1 OpenCL Configuration

In OpenCL, the host is tasked with setting up, dispatching, and collecting results from a device. OpenCL is accessed through an API, which enables relatively low-

[3] https://www.khronos.org/opencl/ - Accessed 5 January 2015

```
1  __kernel void square(__global float* input, __global float* output,
2                                          const unsigned int count){
3      int i = get_global_id(0);
4      if(i < count)
5          output[i] = input[i] * input[i];
6  }
```

Fig. 1: OpenCL Kernel to Compute the Square of An Input Array

level access to data types and functions in order to program and interact with one or more accelerators.

Creating an OpenCL environment consists of first querying the hardware at runtime to determine the available vendor *platforms* and the *devices* available in each platform. Platforms are essentially drivers provided by the hardware vendor, and the devices represent the actual accelerators. Then, a `context` must be created. A context is an umbrella structure that holds the device(s) to be used, as well as other runtime software constructs. A `command_queue` is then associated with each device and placed within the context. A `command_queue` is used to issue commands to a device. Commands include device queries, memory management operations, and kernel (Section 2.2) invocations. After this, a user creates a `program` with the kernel source file, and compiles it at runtime. The specific function to be executed within the compiled source is then used to create the `kernel` object. At this point the OpenCL environment has been constructed.

From here the user allocates memory on the device and then copies host data into this memory. The device memory is then associated with the correct position in the kernel arguments. Then, the number of dimensions upon which the kernel should work is calculated, and the kernel is launched on the device, with this information, via the `command_queue`. Usually, the host then blocks attempting to read data back from the device once it has finished its computation. Once all computation is complete and the device is no longer required, there are appropriate destructor functions. The device itself is treated simply as a functional unit. Data and code are passed to the device, the device executes this code, and the results are read back by the host.

2.2 Kernels

A device runs a special piece of code known as a *kernel*. An OpenCL kernel is written in a C-like syntax and represents the logic of a single thread. The number and groupings of threads are supplied during the configuration stage on the host. These values are known as the `local` and `global` worksizes, and are used to optimise the allocation of threads to the underlying hardware for a given dataset. Within a kernel, the currently executing thread may be identified via the API. This can be used to customise application logic. The kernel is expressed as a function with parameters. Information for the actual computation is passed to this function as arguments by the host.

The OpenCL model uses a memory hierarchy in which memory is split into `global`, `local`, `private`, and `constant` regions. This is a direct mapping to the

hardware configuration of memory found in GPUs, however the same model is applied to all hardware devices. Global memory is shared amongst all threads, local memory is shared between a specified group of threads, and private memory is specific to a thread. Global and local memory are subject to unsynchronised modifications, although there are mechanisms to synchronise access. Constant memory is shared by all threads, but is read only. Listing 1 shows a simple kernel.

3 Parallel Programming Frameworks

There are a number of different techniques which are used to program GPUs. These can be split into three different equivalence classes: APIs, fully-automated parallelisation, and semi-automated parallelisation.

3.1 API Approach

The most low level approach to programming an accelerator is via an API within an existing language. Examples of this include Python [11], Java [4], and the original implementations in C/C++ of OpenCL [1] and CUDA [13].

While an API is a simple approach, requiring no modifications to the host language, the general drawback of using an API is the need to write large amounts of boilerplate code simply to setup the programming environment, as described in Section 2.1. This boilerplate code often follows the steps required to setup and initialise the relevant framework, and can have very little relation to the programming idioms of the host language - e.g., the Java API requires the use of pointer objects. Similarly, the code required to express the calculations on the accelerator is written in a C-like language, which is embedded in the host language as a string. For non-C-like languages, such as Java and Python, this can be challenging for programmers without experience in C/C++. More generally, this leads to two different programming styles for the host and the accelerator, although it has the potential for the best performance.

3.2 Automated Approach

A different approach is to hide from the developer all the low-level details such as initialisation, data allocation and movement, the specification of work sizes, and when to dispatch kernels. A number of domain-specific languages have been developed to provide such higher level abstractions. Examples of these include **Accelerator** [16], **LIME** [6], **SkelCL** [14], and **Chestnut** [15].

With the exception of Accelerator, which pre-dates OpenCL and CUDA and is no longer under development, the primitives of these languages abstract data movement and computation through operations such as map, reduce, stencil, and other vector operations. In general, the effectiveness of the approach relies on the maturity of the source language, and the quality of the code transformation

[4] http://www.jocl.org/ - Accessed 5 January 2015

to the underlying transformation. For example, Chestnut does not support data structures, SkelCL requires stack operations to assign variables to a kernel, and LIME only allows static topologies of tasks and immutable state.

3.3 Semi-Automated Approach

Much work has been done on semi-automated translation of serial programs to parallel OpenCL or CUDA code. The techniques used include recognition of common parallelisable patterns in the source code, such as nested loops that can be unrolled and executed in parallel. However, most rely on the programmer to provide annotations or some form of refactoring applied to the original code.

AMD's open-source **Aparapi**[5] system allows partial offloading of Java code at runtime. Aparapi relies on the programmer to refactor a function or loop into an inner class with a run method containing the computation. The Java-byte code for this method is translated to OpenCL/C code at runtime. The programming model is similar to our work as it requires some refactoring but still allows the kernel code to be written in a subset of the original language, making use of the primitives (such as classes) provided by that language to express the required meta-information. It also abstracts, to a certain extent, the exact memory layout and management of data movement and provides automatic adjustment to the available devices, rather than having the programmer optimize the code for a specific device. Aparapi offers a trade-off between a simplified programming model, and non-trivial performance cost [5].

OpenACC [17] is a set of explicit annotations and API for C or Fortran code, which enables annotated code to execute on GPUs. Separate **OpenMP** [3] annotations are required for CPUs. With OpenACC, the user must explicitly annotate data to be moved between host and device, as well as sections of code which should be parallelised. Using these hints, the compiler attempts to generate optimised code for the supported patterns. The system enables the use of existing code with fewer modifications than is required to use the OpenCL API. The PGI[6] compiler provides the best performance, but is proprietary. There are also open-source compilers, with varying quality and reliability. **hiCuda** [8] is a similar approach for the CUDA framework, applying annotations to sequential C.

By their nature, annotations are extraneous to the logic of the underlying application. This has the advantage that existing code can be used as a starting point, with annotations extending or enhancing the functionality. However, this means that there is no intuition as to what is happening *under the hood*. Annotations must be applied to *each* construct to be parallelised, resulting in applications which are increasingly difficult to follow as their size increases. Also, there is no guarantee that the compiler will be able to generate an effective parallel strategy for the annotated section of code. For example, if there is a non-linear data dependency in a for loop, sequential code may be generated instead of parallel. Thus, the approach is unsuitable for all application classes, requiring code to be refactored or rewritten, negating a main advantage of the approach.

[5] code.google.com/p/aparapi - Accessed 5 January 2015
[6] http://www.pgroup.com/ - Accessed 5 January 2015

```
type Isnd is interface(
    out integer output
)
type Ircv is interface(
    in integer input
)
stage home{
 actor snd presents Isnd {
  value = 1;
  constructor() {}
  behaviour {
    send value on output;
    value := value + 1;
  }}
 actor rcv presents Ircv {
  constructor() {}
  behaviour {
   receive data from input;
   printString("\nreceived: ");
   printInt(data);
  }}
 boot{
  s = new snd();
  r = new rcv();
  connect s.output to r.input;
}}
```

Fig. 2: Simple Ensemble Send and Receive Example

4 Ensemble

Ensemble is an actor-based programming language. It is an evolution of the Insense programming language [4] which was originally created to ease the development of software for wireless sensor networks. An Ensemble actor has its own private state and a single thread of control. The single thread of control is expressed as a `behaviour` clause. The code within this clause is repeated until explicitly told to stop. All actors execute within a `stage`. Stages represent a memory space, thus many stages may exist per physical machine. Ensemble supports reference counted garbage collection.

Actors have *shared-nothing* semantics, meaning that they share no state. Message passing along uni-directional, typed channels is used to share information between actors. Each channel may have an optional buffer to enable asynchrony during communication. When no buffer is specified or a buffer is full, the system reverts to synchronous, blocking communication. Channels are either statically associated with a specific actor via an interface, or dynamically created and composed at runtime into one or more 1-1, 1-n, or n-1 topologies.

In actor-based languages, data sent along channels must be duplicated to ensure that shared-nothing semantics are preserved. One novel feature of Ensemble is the ability to declare data *movable* (`mov`). By marking heap allocated memory as movable, only a reference will be sent along a channel, as opposed to a duplicate. Movability is included within the type system of the language, using inter-procedural analysis at compiletime to ensure that once a reference is sent it is not accessed again until it is assigned to, otherwise a compiletime error is generated. This feature was originally developed to reduce heap memory consumption and fragmentation in small embedded devices. Figure 2 shows a simple Ensemble program which defines, instantiates and connects two actors, one of which sends linearly increasing values to the other.

One of the core principles of Ensemble is adaptability and scalability. To support this, Ensemble programs are compiled to custom Java bytecodes, which then run on a custom built virtual machine (VM) [2]. This in turn runs on a custom operating system (OS) [9]. Currently, Ensemble is capable of running on small resource constrained sensor platforms, as well as embedded hardware, reasonably-provisioned systems like the Raspberry Pi, and desktop machines.

```
1  type data_t is struct (
2      real [][] a;
3      real [][] b;
4      real [][] result
5  )
6  type settings_t is opencl struct (
7      integer [] worksize;
8      integer [] groupsize;
9      in data_t input;
10     out real [][] output
11 )
12 type dispatchI is interface(
13         out settings_t requests;
14         out data_t dout;
15         in real[][] din
16 )
17 type mulI is interface(
18         in settings_t requests
19 )
20 stage home{
21   opencl <device_index=0, device_type=CPU>
22     actor Multiply presents mulI {
23       constructor() {}
24       behaviour {
25         receive req from requests;
26         receive d from req.input;
27         x = get_global_id(0);
28         y = get_global_id(1);
29         dim = get_global_size(0);
30         c = 0.0;
31         for i = 0 .. (dim-1) do {
32           c := c + (d.a[y][i]) * (d.b[i][x]);
33         }
34         d.result[x][y] := c;
35         send d.result on req.output;
36     }}
37     actor Dispatch presents dispatchI{
38       constructor() {}
39       behaviour {
40         s = 1024;
41         ws = new integer[2] of s;
42         gs = new integer[2] of 0;
43         i = new in data_t;
44         o = new out real[][];
45         connect dout to i;
46         connect o to din;
47
48         ocl_struct = new settings_t(ws,gs,i,o);
49         d = generate_data(s);
50
51         send ocl_struct on requests;
52         send d on dout;
53         receive result from din;
54     }}
55     boot{
56       d = new Dispatch();
57       m = new Multiply();
58       connect d.requests to m.requests;
59     }
60 }
```

Fig. 3: Matrix Multiply Example

5 Integration of OpenCL in Ensemble Actors

This section describes the integration of OpenCL into Ensemble with minimal changes to the language. By doing this, Ensemble provides a high-level, single-source solution which leverages OpenCL's existing low-level, yet portable, split-source infrastructure.

5.1 Language Integration

The similarity between the isolated nature of computation in actor-based programming and the OpenCL model is strong. Both require explicit data movement between loci of computation. In Ensemble, the channel mechanism provides a natural way to facilitate this without the need to go *outside the language* or obfuscate existing code; consequently, OpenCL kernels are represented as actors. The requirement for separate programming of host and device also matches the distinction between actors. The full example of matrix multiplication in Ensemble shown in Figure 3 is referred to throughout this section. The green colour represents additions to the language, and light yellow shows compiler-enforced structure. These highlighted sections represent the only changes to the language.

Language Model In Ensemble, an actor is marked as being a kernel by adding the `opencl` keyword to its definition (line 21). This tells the compiler that the

actor's interface should only contain a single channel, and that a slightly different structure is expected within the behaviour clause. In this way, kernels are integrated into the language, not a separate source file in a different language.

The single channel conveys an `opencl struct` type as defined by the developer (lines 6-11). This is a normal `struct` except that the `opencl` keyword tells the compiler that the fields of the `struct` should contain two integer arrays, an `in` channel, and an `out` channel. The arrays convey the local and global worksizes required to dispatch the OpenCL kernel, as discussed in Section 2. The two channels are the input and output channels for the data to and from the kernel, respectively. The channels are created and configured in the host actor (lines 43-46), sent to the kernel actor via the input channel (line 51), and then the data is sent (line 52). The host then waits for data from the actor (line 53). This is an example of the dynamic nature of Ensemble channels, and enables any type of data to be conveyed to and from the kernel actor safely, without requiring any extra compiler analysis, beyond normal language processing.

Within the behaviour loop of the kernel actor it is required that the first two statements are receive statements (lines 25-26). The first statement receives an `opencl struct` instance. This enables the runtime to prepare the kernel for dispatch to the device with the appropriate dimensions by using the `worksize` and `groupsize` values. The second statement is used to receive the data that the kernel will process. It is also required that the last statement is a send statement (line 35). This is used to send the processed data onwards. The send statement is also used as a marker, with all statements between the second receive and send statements representing the OpenCL kernel (lines 27-34). The standard set of OpenCL calls natively available in a kernel are also available within the OpenCL behaviour clause between these markers (lines 27-29), including the math functions. Also, the `global`, `local`, and `private` memory modifiers are available, enabling the developer to take full advantage of the different memory regions which are available in the OpenCL model to improve performance.

One of the key advantages of embedding OpenCL within the language is the ability to preserve multi-dimensional array and structure dereferencing. Currently, OpenCL requires that arrays and structures containing pointers are flattened when being passed to a kernel. In Ensemble, this process is automated, leaving the users with normal dereferencing of such data types within the kernel. This also has the advantage of providing the user with warnings and errors for kernels at compiletime, rather than having to wait until runtime kernel compilation. Furthermore, as all the OpenCL actors are connected by channels, should the user wish to change the device upon which the OpenCL actor should run, the language only requires that the device type be modified in the actor definition. No other change is required. Also, should the developer wish to use a different kernel or a different device at runtime, all that is required is to reconnect the configuration channel to an appropriate kernel actor's configuration channel.

By tagging an actor as an OpenCL kernel, a user must still write parallel code within the behaviour clause; however, by integrating this into the language model, the process is much simplified compared to a C version, see Section 6.

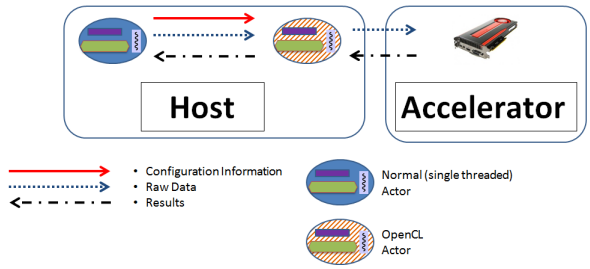


Fig. 4: OpenCL in Ensemble Execution Model

Compiler Modifications The compiler was modified to apply slightly different rules to an OpenCL actor, enforcing the structure described previously. The channel operations mirror the explicit data movement required by OpenCL, and it is the channel operations that dictate the creation of device buffers and the movement of memory between the host and the device. `struct` values are flattened so that each field is sent separately, with the compiler generating the appropriate code within the kernel to manage this. Multi-dimensional arrays are flattened to single dimensional arrays. Again, the compiler generates appropriate kernel code to manage this. Primitive values are sent as 1D-arrays of one element so as to ensure updates within the kernel are applied to the value. Passing a pointer to the host variable is not an option. A potential optimisation here is to wrap all passed primitive variables in a single array.

Execution Model Figure 4 shows the execution model when using OpenCL with Ensemble. Rather than the actor running on the device directly, an OpenCL actor is compiled into a bytecode representation of the actor in the normal way. A C representation of the code identified as the kernel is generated, and stored as a string within the actor's bytecode. Should the kernel actor contain functions defined in other code sections, the compiler will generate C equivalents within this string. This is completely hidden from the developer. The bytecode is interpreted as normal, and acts as the host in the traditional OpenCL sense. This actor handles the incoming and outgoing channel communications, as well as preparing, launching, and collecting data from the kernel. This also enables multiple kernels to execute on a single device.

5.2 Runtime Integration

Runtime In order to add OpenCL support into the Ensemble runtime, a number of modifications were made. Firstly, the use of OpenCL is optional, and conditionally compiled into the runtime. This was necessary to ensure that RAM and ROM on resource-constrained platforms without OpenCL were not consumed. In this way, the same source tree is still available for multiple platforms, reducing maintenance and update effort.

Secondly, during the initialisation of the runtime, a single matrix is created to hold the different platforms and devices available in this system. This is done to

ensure that there is only a single `command_queue` per device. This was necessary as race-conditions were observed with multiple `command_queue`s per device when reading data. The information passed in the declaration (Figure 3, line 21) of an OpenCL actor is used to index into this matrix at runtime to determine the appropriate `context` and `command_queue`. If no information is given in the declaration, default values are used.

Thirdly, OpenCL wrapper functions were created and made available to the interpreter to simplify and abstract the interaction with the OpenCL API.

Interpreter Within the interpreter, all OpenCL operations are implemented in C for performance. Each operation described in Section 2.1 is implemented as a custom native operation in the VM. The VM has a special `invokenative` bytecode that is used to provide access to certain primitive operations in the runtime. This includes actor and channel lifecycle operations, and now includes access to the OpenCL wrappers discussed previously. Also, each OpenCL actor is given an `OpenCLEnvironment` variable. This is a runtime structure only visible within the interpreter that is used to store metadata about the platform, device, and device type, as well as the relevant `command_queue` and `context` for a given OpenCL actor. This structure is populated when the actor is created using the information contained in the previously described runtime matrix.

Lazy Evaluation In OpenCL, a common idiom is to leave data on a device for as long as possible, thus reducing the time spent copying data between the device and host, and ultimately the application's execution time; data movement is often the greatest performance bottleneck. In actor-based languages there is no shared state, hence when messages are sent between actors, a duplicate is created and sent. This ensures no shared state between the actors, and that each actor has a unique copy of the data. While this is correct, it costs time, requires greater memory consumption, and precludes keeping data on the device.

To prevent such duplication, Ensemble supports marking non-primitive types as movable (`mov`), as discussed in Section 4. This approach has been applied to the OpenCL kernels in Ensemble. Marking the `in` channel to the kernel used for data as moveable (`mov`) has two effects. Firstly, once any non-primitive data is copied to the device it is marked as no longer being on the host, and copies of relevant OpenCL data structures are associated with the runtime representation of the data type. Secondly, the compiler will not generate the code to read this data back from the device. Thus, when the data is sent onwards it will hold a reference to the data on the device.

At this point there are two possibilities for the data that is sent onwards. The first option is that the data successfully arrives at another OpenCL actor without being accessed by the host. In this case, the pointer to the device data is set as the appropriate kernel argument, and the kernel is dispatched. Here, the data was kept on the device at all times. The second option is that the data is either accessed directly by host code, or the data is sent to an OpenCL actor associated with a different context. In both cases, the runtime reads the data back from the device and returns the device memory. As Ensemble uses automatic garbage collection, should the host reference count ever reach zero,

both host and device memory will be returned. In this way, the choice to use `mov` gives the user control over memory usage.

One benefit of lazy evaluation is that the runtime can automatically determine, using the `OpenCLEnviroment`, if the incoming data needs to be moved to the current context, and then do so if required. Currently, OpenCL manages data movement between devices in a single context, but not in different contexts.

6 Evaluation

This section explores the linguistic complexity and performance cost of code parallelisation with actors by comparison to API and directive-based programming.

6.1 Applications

A range of applications were chosen to evaluate the linguistic complexity and performance of actor-based OpenCL in Ensemble as compared to equivalent C and OpenACC implementations. The range of applications covered include matrix operations, multiple kernels, parallel reduction and a real world application. All of the source code for the described applications can be found online[7].

- **Matrix Multiplication** multiplies two 1024^2 matrices in a single kernel.
- **Mandelbrot** computes a 1000 iteration Mandelbrot set in a single kernel.
- **LUD (Lower Upper Decomposition)** factorises a square matrix of 2048 elements, and is a common operation in matrix calculations: this example uses three kernels in series.
- **Matrix Reduction** finds the minimal value in an array of 33,554,432 elements using parallel reduction in a single kernel.
- **Document Ranking** takes a set of documents and using a template determines if these documents are wanted, or unwanted: this is a single kernel.

6.2 Experimental Setup

The host machine had 16GB of RAM. CPU refers to an Intel Core i5-3550 CPU @ 3.30GHz, and GPU refers to an AMD Radeon R9 290x GPU. All results are the average of one hundred runs of the respective application, and the error bars indicate the standard deviation of the total execution times. All code has been compiled with the -O2 optimisation. Unless stated otherwise, the same results were observed for all applications and implementations The AMD version of OpenCL 1.2, and version 14.10 of the PGI compiler was used. All configuration information is found within the online code examples.

6.3 Code Complexity

Table 1 shows the (arithmetic) difference between the concurrent and non-concurrent code versions for each approach. As well as the lines of code written,

[7] www.paul-harvey.org/data-and-links.html

Table 1: Difference Between Single Threaded and Concurrent Code per Approach

	Lines of Code			Cyclomatic Complexity			ABC		
Application	C	Ensemble	OpenACC	C	Ensemble	OpenACC	C	Ensemble	OpenACC
Matrix Multiplication	154	-8	5	-1	-2	0	134	2	1
Mandelbrot	96	-4	12	-1	1	0	22	-6	2
Reduction	266	72	3	19	4	1	103	10	0
LUD	144	7	7	5	-8	1	200	-11	0
Document Ranking	45	-16	3	53	-1	0	405	-29	0

the table also shows McCabe's cyclomatic complexity [12] for applications. This metric quantitatively assesses the number of different paths through a program. Also shown is the assignments, branches, and conditions (ABC) metric that assesses the size/complexity of code [7]. In each case, the number shown is for the entire application; negative values indicate a decrease in the specified metric.

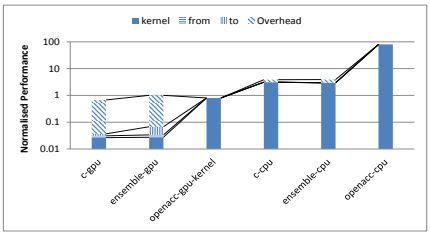
By comparison to C-OpenCL, both Ensemble-OpenCL and C-OpenACC require fewer lines of code, and are simpler by both metrics, with the only exceptions being the matrix multiplication and Mandelbrot cyclomatic complexity examples. The negative values seen are due to the kernel code/actor effectively replacing the outer `for` loop in the single-threaded version, thus reducing code and complexity. In general, annotating code with OpenACC pragmas has little effect on the code size or metrics. The main impact is from having to explicitly specify the sizes of the data to be moved, requiring variables to be accessed.

The implicit kernel `for` loop, plus the actor and channel abstractions accounts for the better results generally shown by Ensemble-OpenCL compared to C-OpenACC. The main discrepancy being the reduction example, which required very different kernel logic to the single-threaded equivalent in both Ensemble and C, however, this mindset is advocated by both approaches, unlike OpenACC.

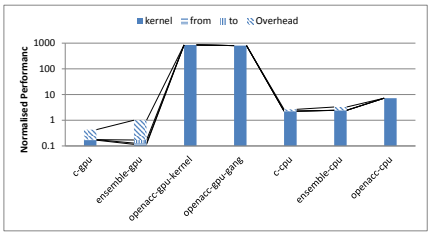
6.4 Performance

Figures 5a-5e show the comparative performance of Ensemble-OpenCL, C-OpenCL and C-OpenACC for the applications described above. C-OpenACC will refer to both GPU and CPU results, as the PGI compiler was used for both OpenACC and OpenMP pragmas. Each column in each figure displays the time taken to complete the given application, normalised to the Ensemble GPU results. This is done to highlight relative application performance, rather than absolute execution performance. The columns are split to show the relative amount of time taken to move data to a device, move data from a device, to execute a kernel, and the overhead which represents the total relative application execution time minus these values. This does not apply to C-OpenACC as it was not possible to correctly identify the distinct operations due to the `pragma`-based abstraction.

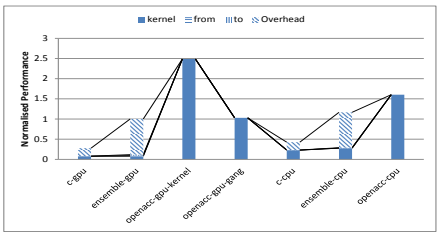
Figures 5a and 5b show the results for the matrix multiplication and Mandelbrot applications, respectively. The OpenCL actions (data movement and kernel calculations) are nearly equivalent between the C and Ensemble versions. The higher overheads in the Ensemble version are due to interpreter overhead for the non-kernel code. C-OpenACC shows similar performance on the GPU for Figure 5a, however much worse performance in Figure 5b, even when using the fine-grained `gangs` and `worker` annotations to explicitly specify the `groupsize`



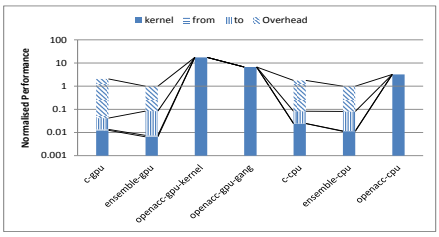
(a) Matrix Multiplication



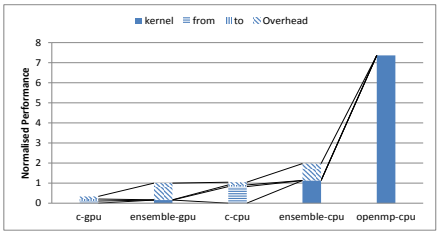
(b) Mandelbrot



(c) LUD



(d) Reduction



(e) Document Ranking

Fig. 5: Performance between C-OpenCL, Ensemble-OpenCL and C-OpenACC Normalised to Ensemble GPU

and `worksize` to be used on the GPU. Whereas an explicit kernel can take advantage of each thread's position within the 2D architecture of the GPU, the C-OpenACC abstraction cannot, hence the poor performance.

Figure 5c is of particular note as it highlights one of the advantages of using channels. In this application there are three kernels, each performing a different operation. In the Ensemble version, a controller actor *plumbs* the channels of the kernel actors together, creating a pipeline of kernels. The controller then sends the data, and waits for the final result. By comparison, the C-OpenCL version sequentially invokes each kernel from the host. As C-OpenACC is annotated single-threaded code, it also invokes the relevant code sequentially.

This application highlights the advantage of movability within Ensemble. Without movability, LUD took approximately 3 minutes (not shown) to complete on the GPU due to all the data movement involved; with movability, it takes approximately five seconds on the GPU. Here we can see that by using movable types, the time taken by Ensemble is comparable to the C version. The higher overheads are caused by the non-OpenCL code being interpreted by the unoptimised Ensemble VM, not data movement.

In order to obtain comparable performance from OpenACC, annotating the outer loop of the relevant code was not sufficient, requiring use of the non-trivial `gangs` and `worker` annotations. This makes the distinction of host and device explicit, with the added disadvantage of being inline in the code, rather than having distinct sections of code. Again, worse performance is seen from the CPU.

Figure 5d shows that Ensemble-OpenCL closely tracks the performance of C-OpenCL. C-OpenACC performs poorly for this application on both the GPU and CPU due to this type of application requiring different logic to take advantage of parallel hardware. Again, annotating the sequential version is not enough for this type of application.

Figure 5e shows results for a real world example and indicates that the kernel execution time in Ensemble-OpenCL is greater than in C-OpenCL. This is due to some fundamental differences between the host languages, rather than the programming models. Firstly, in Ensemble there are no `NULL` values. This means that all data types must be initialised at the point of creation. This has the advantage of making the language safer, but can lead to increased execution time when the variable is written to before reading - i.e. initialisation was not required. In this application, two arrays are initialised in the kernel, within a loop with many iterations. A similar action is taken in the C version, however the two initialisation loops are combined, effectively *halving* the amount of work done by the C version. Loop unrolling in the code generated from the Ensemble compiler could help with this.

Secondly, the ability of C to use an integer value as both a boolean indicator and numerical value leads to faster code when compared to Ensemble, which has no such overloading. Ensemble uses separate types for numeric and boolean values, which in this application requires a number of control structures to be used, and causes greater execution time.

Thirdly, the C-OpenCL version uses short vector types and operations. Ensemble does not yet support OpenCL specific types such as short vectors, and the associated vector operations upon such types. This limitation is due to time constraints, rather than implementation or theoretical barriers. Such types and operations will be added to the Ensemble type system, and will reduce the execution time. Again, this is a limitation of Ensemble, not actors.

The second observation is the smaller communication time in Ensemble-OpenCL compared to C-OpenCL. This was an unexpected consequence of movability (Section 5). In this application, the kernel execution was run multiple times during each individual run to collect sufficiently large time values. In the C-OpenCL version, the data is copied to and from the device each time. No changes or modifications are made to the data between movements. This is also true in the Ensemble version, however due to the lazy evaluation logic the data on the device is never moved back to the host as it is not required to do so.

The PGI compiler was not able to compile this code, hence no results were obtained for the GPU or CPU from C-OpenACC. The CPU results were generated from the OpenMP `pragma`s and the gcc compiler. Even with the gcc compiler we still see the slower CPU results by comparison with the other methods.

From the results shown, the general trend is that the performance of hand coded C-OpenCL is comparable to Ensemble-OpenCL. As both C and Ensemble are using the OpenCL runtime, the main difference in time between these approaches comes from the overhead of the Ensemble VM. There are optimisations that can be applied to the implementation, as discussed previously, however the fact that Ensemble is an interpreted language accounts for the majority of the overhead when compared to the C version, as opposed to issues with the language's actor-based structure.

Given these results and the previous discussion, we have demonstrated that:

- Ensemble-OpenCL always enables simpler, functionally-equivalent code with many fewer lines of code compared to C-OpenCL, and generally simpler, equivalent code with fewer lines when compared to C-OpenACC;
- Ensemble-OpenCL applications present commensurate performance to C-OpenCL on GPU & CPU;
- Relative performance of Ensemble-OpenCL to C-OpenACC ranges from equivalent to significantly better on GPUs, and from better to vastly better on CPUs.

7 Conclusions and Future Work

Conclusion This work presents the first attempt at using the actor programming model to simplify kernel-based concurrency. The separation of host and device code, as well as the explicit data movement found in kernel programming, naturally maps to the concept of isolated actors using message-passing communication. By comparing C-OpenCL with Ensemble-OpenCL, this paper has shown that Ensemble-OpenCL requires less code, is qualitatively simpler, and provides comparable performance with hand written C-OpenCL. Also, by taking advantage of novel features in the language, it is still possible to use common OpenCL optimisation techniques which are in contrast to the normal shared-nothing semantics of actor-based programming. Furthermore, by comparison with C-OpenACC, Ensemble-OpenCL shows performance ranging from better to vastly better, at the cost of slightly larger code size in a small number of cases.

By offering a simple and performant programming model for a range of applications classes, actors provide an ideal abstraction for kernel-based concurrency.

Future Work The main bottleneck in performance is the Ensemble VM. To address this, a JIT compiler will be built to remove the overhead of bytecode interpretation. Furthermore, the Ensemble type system will be expanded to include OpenCL types, such as short vectors.

Work is currently in progress to add distribution natively to Ensemble. This includes abstraction of network communication between actors via the channel mechanism, remote deployment, and runtime actor migration. Once complete, one possible use-case will be to program and load-balance a computer cluster of heterogeneous platforms entirely with actors from within the language.

References

1. The opencl specification. Khronos OpenCL Working Group 1, l1–15 (2009)
2. Cameron, C., Harvey, P., Sventek, J.: A virtual machine for the insense language. In: MOBILe Wireless MiddleWARE, Operating Systems and Applications (Mobilware), 2013 Intl. Conference on. pp. 1–10 (Nov 2013)
3. Chapman, B., Jost, G., Van Der Pas, R.: Using OpenMP: portable shared memory parallel programming, vol. 10. MIT press (2008)
4. Dearle, A., Balasubramaniam, D., Lewis, J., Morrison, R.: A component-based model and language for wireless sensor network applications. In: Proc of the IEEE Intl. Computer Software and Applications Conference. pp. 1303–1308 (2008)
5. Docampo, J., Ramos, S., Taboada, G., Exposito, R., Tourino, J., Doallo, R.: Evaluation of java for general purpose GPU computing. In: Advanced Information Networking and Applications Workshops. pp. 1398–1404 (March 2013)
6. Dubach, C., Cheng, P., Rabbah, R., Bacon, D.F., Fink, S.J.: Compiling a high-level language for GPUs: (via language support for architectures and compilers). In: Proc of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 1–12. PLDI '12, ACM, New York, NY, USA (2012)
7. Fitzpatrick, J.: More c++ gems. chap. Applying the ABC Metric to C, C++, and Java, pp. 245–264. Cambridge University Press, New York, NY, USA (2000)
8. Han, T.D., Abdelrahman, T.S.: hiCUDA: High-level GPGPU programming. IEEE Trans. Parallel Distrib. Syst. 22(1), 78–90 (2011)
9. Harvey, P., Dearle, A., Lewis, J., Sventek, J.S.: Channel and active component abstractions for WSN programming - A language model with operating system support. In: SENSORNETS 2012 - Proc of the 1st Intl. Conference on Sensor Networks, Rome, Italy, 24-26 February, 2012. pp. 35–44 (2012)
10. Jeffers, J., Reinders, J.: Intel Xeon Phi Coprocessor High Performance Programming. Newnes (2013)
11. Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A.: PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. Parallel Comput. 38(3), 157–174 (Mar 2012)
12. McCabe, T.: A Complexity Measure. Software Engineering, IEEE Transactions on SE-2(4), 308–320 (Dec 1976)
13. Sanders, J., Kandrot, E.: CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional (2010)
14. Steuwer, M., Kegel, P., Gorlatch, S.: SkelCL - a portable skeleton library for high-level GPU programming. In: Proc of the IEEE 27th Intl. Symposium on Parallel and Distributed Processing. pp. 1176–1182 (2011)
15. Stromme, A., Carlson, R., Newhall, T.: Chestnut: a GPU programming language for non-experts. In: Proc of the 2012 Intl. Workshop on Programming Models and Applications for Multicores and Manycores. pp. 156–167. ACM (2012)
16. Tarditi, D., Puri, S., Oglesby, J.: Accelerator: Using data parallelism to program GPUs for general-purpose uses. In: Proc of the 12th Intl. Conference on Architectural Support for Programming Languages and Operating Systems. pp. 325–335 (2006)
17. Wienke, S., Springer, P., Terboven, C., an Mey, D.: OpenACC: First experiences with real-world applications. In: Proc of the 18th Intl. Conference on Parallel Processing. pp. 859–870. Springer-Verlag (2012)