A Scalable Runtime for the ECOSCALE Heterogeneous Exascale Hardware Platform

Paul Harvey, Konstantin Bakanov, Ivor Spence, Dimitrios S. Nikolopoulos
Queen's University Belfast
University Road
Belfast, United Kingdom
paul@paul-harvey.org, {k.bakanov, i.spence, d.Nikolopoulos}@qub.ac.uk

ABSTRACT

Exascale computation is the next target of high performance computing. In the push to create exascale computing platforms, simply increasing the number of hardware devices is not an acceptable option given the limitations of power consumption, heat dissipation, and programming models which are designed for current hardware platforms. Instead, new hardware technologies, coupled with improved programming abstractions and more autonomous runtime systems, are required to achieve this goal.

This position paper presents the design of a new runtime for a new heterogeneous hardware platform being developed to explore energy efficient, high performance computing. By extending and enhancing the OpenCL framework, this work will both simplify the programming of current and future HPC applications, as well as automating the scheduling of data and computation across this new hardware platform. Also, this work explores the use of FPGAs to achieve both the power and performance goals of exascale, as well as utilising the runtime to automatically effect dynamic configuration and reconfiguration of hardware platforms.

CCS Concepts

•**Computer systems organization** → **Parallel architectures;** •**Software and its engineering** → **Scheduling;** *Language features;* •**Hardware** → *Power and energy;*

Keywords

heterogeneous parallel runtime, automated scheduling, parallel programming, FPGA, opencl, data partitioning

1. INTRODUCTION

There are more applications than ever which require high performance computing platforms. As the need for more detailed weather simulations, oil field simulations, or smart city applications increases, so too does the need to move towards hardware platforms which can perform a billion billion calculations per second: exascale computing.

In order to meet this demand, simply increasing the number of hardware devices is not a scalable option given the relative non-linear scaling of power consumption, heat generation, space requirements, and cost. For example, extrapolating from the top HPC systems, such as China's Tianhe-2, it is estimated that sustaining exaflop performance requires a substantial 1GW of power [13]. Consequently, more power efficient hardware architectures are going to be required.

In addition to the changes required to the hardware platforms, there is also a need to support better programming models, and their associated runtimes, in order to enable developers to take advantage of these platforms. Although the default MPI + X programming approach, where X is a programming language of choice, has been and still is an efficient programming option, the low level at which MPI operates is an ongoing barrier to programmers across the spectrum of abilities. Furthermore, as HPC systems are becoming increasingly heterogeneous, using either parallel or reconfigurable hardware architectures (accelerators), there is requirement to move away from semantically broken programming models which are aimed at current computing technologies towards those better able to support developers for these heterogeneous distributed platforms.

This paper outlines the design of a runtime system for a new heterogeneous hardware platform. In order to ease adoption and uptake, this runtime is based on the task and data parallel models found within the OpenCL programming framework, but expands upon this in order to transparently leverage the partitioned global address space (PGAS) model that it provides across a number of devices. In particular, this work seeks to break the one-to-one association between a hardware device and the current granularity of work in OpenCL to provide flexibility for developers. The runtime will extend OpenCL abstractions, enabling automated placement of data and computation across a heterogeneous hardware platform, with particular focus on FPGAs.

The runtime is being developed as part of the ECOSCALE Project[1] and will execute on a new hardware platform being developed in parallel to address the requirements of energy efficient, scalable computing in order to meet the future needs of an exascale compute platform[2]. Additionally, the runtime and the hardware platform are being developed in

[1] www.ecoscale.eu

[2] ECOSCALE is part of a trio of projects, all looking to creating the building blocks of an exascale platform - http://www.hpcwire.com/2016/02/24/eu-projects-unite-exascale-prototype/

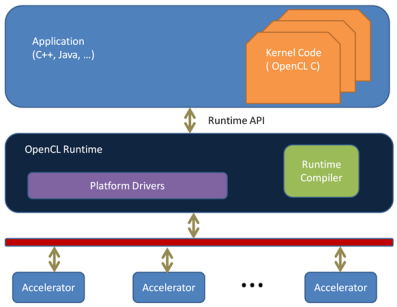

Figure 1: The Runtime Architecture of OpenCL

tandem with a number of industrial applications to provide meaningful use cases to both guide and exercise the work.

The key contributions of this work are:

- Straightforward extensions to the OpenCL programming interface to support automated partitioning of data across multiple heterogeneous hardware platforms.
- A new set of device abstractions for OpenCL to enable location transparent programming.
- The design of a runtime system to enable the concurrent scheduling of data and computation across multiple heterogeneous hardware platforms, including hardware reconfiguration at runtime.

The remainder of this paper is organised as follows: Section 2 provides an overview of the state of the art to place this work in context, Section 3 describes the hardware platform being targeted by this work, the proposed language extensions and design of the runtime are described in Section 4, and a summary of the paper is made in Section 5.

2. BACKGROUND

There have been a number of different approaches to address the creation and execution of HPC applications. In order to establish a context for this work the following is a discussion of OpenCL, as well as a summary of the related literature, separated into four equivalence classes.

2.1 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].

Given the increasing simplicity with which developers are able to program accelerators with OpenCL, it is a prime candidate to explore exascale programming.

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

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

```
1    __kernel void square(__global float* input,
2                         __global float* output,
3                         const unsigned int count){
4        int i = get_global_id(0);
5        if(i < count)
6          output[i] = input[i] * input[i];
7    }
```

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

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 2.1.2) invocations. Next, 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.

At this point 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.

2.1.2 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 spec-

ified 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.

2.2 Location Transparent Accelerators

OpenCL was designed for single machines which contain one or more accelerators. Each of these accelerators are explicitly interacted with to both manage and execute kernel computations. However, as discussed previously, modern applications require access to a much larger pool of resources than a single cluster node can provide. A number of projects have begun to explore the idea of expanding the OpenCL runtime to harness the resources available in a cluster.

SnuCL [12] extends the OpenCL framework by transparently presenting remote devices as though they were local. The OpenCL API (v. 1.1) is largely unchanged, except for a minimal number of optional extensions, with the runtime intercepting all functions and redirecting them as appropriate. MPI [6] is used as the remote communications library with an instance of the OpenCL runtime executing on each node as an MPI rank.

SnuCL presents relative simplicity with respect to the operating model, its ability to run standard OpenCL applications unchanged, and the optimizations implemented to address the problems due to executing in the distributed environment. This said, there are a number of limitations of SnuCL. Firstly, it only supports OpenCL version 1.1[12]. As a consequence it lacks the support of the shared virtual memory (SVM) that the newer versions of the OpenCL standard provide. This is important as the need to explicitly manage data movement is a non-trivial problem for developers in terms of performance and linguistic complexity [8]. Furthermore, each device must be configured and managed individually, which ultimately limits the scalability of such approach. There is also no robust discussion of failure management in SnuCL which is necessary generally, but especially in a distributed environment.

VOCL [21] is a similar system to SnuCL. It also implements OpenCL specification 1.1 and uses MPI for inter-node communication. Given its similarity to SnuCL, VOCL has similar advantages. Notable differences are that VOCL is positioned as a fully comprehensive virtual framework, with support for device checkpoint/restart. However, like Snucl there are a number of limitations. Whilst VOCL is a step closer to our work in that each device is virtualised, the mapping between an OpenCL (software) device and a hardware device is still one-to-one, which once again limits its scalability. Also, VOCL uses a centralised controller, with no discussion on how to mitigate performance bottlenecks or single points of failure.

2.3 Scheduling of Kernels

FluidiCL [16] is a system that dynamically schedules kernels between a GPU and CPU in order to improve performance. When the kernels are compiled in the host, the runtime will compile the specified kernel for both devices. When dispatched, the kernel computation is split into many smaller computations which are dispatched on both the CPU and GPU at the same time. Each *subkernel* will then read data from each end of a *single* data set. The idea is that each device will perform as much work as possible, with the more suitable device (CPU/GPU) processing faster and consuming more work. The work experiments with the optimal size of subkernels and notes that it is dependant on whether the kernel is data or compute-intensive.

In order to ensure consistency between the two devices, FluidiCL requires that a complete duplicate of the data be present on each device. This is highly inefficient, and for large computations with large datasets, such as those being targeted in ECOSCALE, makes this an infeasible approach. Also, the approach requires developers to manually alter their kernel code to include coordination points to synchronise data between the two devices. There is no support for synchronisation primitives. Finally, the authors note that FluidiCL is not designed for distributed computations or devices other than GPU/CPU, and is therefore not appropriate for the hardware described in this paper.

Shepard [15] is a framework with similar goals to this work. Shepard aims to decouple application development from the target platform in systems containing multicore CPU and GPU setups. The assumption is that developers will have access to a standard library of kernels for a range of operations. Developers will group these kernels into tasks. The runtime can then group these sets of kernels into sequential executions, with minimal data movement. Through a combination of data input size and runtime profiling, the runtime will then compute the costs of executing these kernels on different devices. Based on these costs, the scheduler will decide if the kernel should launched on a CPU or a GPU.

Although collections of kernels are grouped into tasks, Shepard fundamentally schedules at the level of an entire kernel. This limits the potential parallelism in the system as hardware platforms with multiple devices, such as a cluster, would not be fully utilised. Hence, having the kernel as the schedulable unit is sub-optimal.

Yan et al. [22] extend the OpenMP [4] and OpenACC [20] pragmas to support scheduling entire kernel executions on different devices. By using annotations, Yan et al. aim to modify existing code, thus reducing the barrier to uptake and integration with existing code bases. Again, this work limits potential parallelism by scheduling entire kernels.

Wen et al. [19] describe a scheduler for OpenCL kernels. The system uses machine learning to decide which device (CPU/GPU) an entire kernel should be executed on. By monitoring a number of different linguistic properties of a kernel, as well as the expected input data to a kernel, the machine learning algorithm will decide upon which device the kernel would be most efficiently executed in terms of performance (and not power).

This scheduler is designed for systems executing multiple applications, whereas ECOSCALE targets HPC systems which execute a single application. The work of Wen et al. is designed for systems with GPUs or CPUs, not FPGAs. Also, their system is designed for one or two networked devices, as opposed to a cluster or supercomputer. However, the runtime described in this paper will attempt to apply a number of these techniques in an HPC setting.

A related and similar system has been developed by Grewe et al. [7]. This approach also uses machine learning, but with a greater emphasis on predicting scheduling of kernels, rather than online learning. This approach again schedules entire kernels on GPU/CPU systems.

2.4 HPC Languages

There are a number of different HPC languages [23, 14, 3, 1]. The goal of these languages is to provide appropriate programming idioms specifically for HPC computing to simplify the expression of HPC applications with regard to parallelism and data placement across a cluster of machines or a supercomputer. Additionally, the runtime representation and manipulation of data is often highly optimised for large contiguous arrays of data which fits the requirements of typical HPC applications.

In general, one of the main challenges of these languages is taking existing applications which have been designed for conventional hardware platforms, such as some simulation software or graph-based database applications, and to recreate them in the new language. This requires applications to be completely recreated in a new language, which can be challenging as some applications have been created years ago and maintained across the decades. Although tools can assist, there is no guarantee that the new representation is *bug-for-bug* compatible with the existing version.

2.5 Data Partitioning

Both compute and data intensive applications are present in HPC, as shown by most standard benchmarks now containing a mix of both workloads. As HPC applications incorporate more data-intensive applications, data placement across distributed hardware domains becomes increasingly challenging for the programmer to orchestrate, as well as a bottleneck for performance. The applications targeted in this work are a mix of compute and data intensive.

Although there is a strong desire to automatically classify data partitions, most languages leave this task to the developer and provide tools to simplify this process [23]. The extensions by Yan et al. [22] build on existing work to support specifying how data should be partitioned between different hardware devices during a computation:

- REPLICATE: A separate copy of the specified data is made and sent to each device
- BLOCK(n): Divides the indices in an array dimension into contiguous, equal-sized blocks of size N/P (P = #devices) and each device takes one block (n is the number of elements in the block; default: n= N/P)
- CYCLIC(n): Maps every i^{th} block to number i device of the target dimension of the device topology. (default: n=1)

In addition to these techniques, Yan et al. explain further language annotations to support the exchange of halo information - a key issue for many HPC applications. This is the approach that will be taken initially by this work for data partitioning as it harmonises well with the existing mechanism in OpenCL for computational partitioning.

2.6 Summary

From the literature review in this section, the following key conclusions can be drawn:

- There are no dynamic scheduling approaches for OpenCL kernels which target FPGA hardware
- There is no straightforward programming approach to enable the use of multiple heterogeneous hardware accelerators, distributed or otherwise.

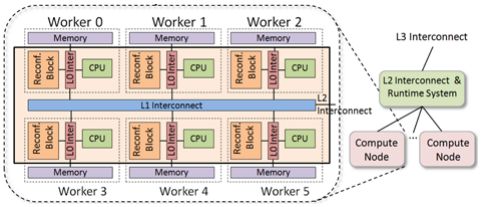

Figure 2: ECOSCALE Hardware Architecture

Given these conclusions, there is a clear gap for fundamental and practical research of runtime systems and scheduling for distributed heterogeneous systems which include FPGA hardware platforms.

3. ECOSCALE HARDWARE

As part of the ECOSCALE project, a new hardware platform is being developed. The following summarises the platform, with a full description found here [13].

This novel system architecture uses CPUs, memory and reconfigurable blocks (FPGAs) in a highly parallel manner. Driven by the characteristics and trends of future HPC applications, the ECOSCALE architecture logically partitions the hardware resources (CPUs, reconfigurable logic, memories, SSDs) into several interconnected **NODE**s which are further partitioned into several **WORKER**s depending on the physical structure of the system. Thus, one or more NODEs offer:

- **UNIMEM**: a shared partitioned global address space that allows WORKERs to communicate via regular loads and stores without global cache coherence.
- **UNILOGIC**: a shared partitioned and reconfigurable set of resources which share the UNIMEM space with software tasks.

Other architectures either require a global cache coherent mechanism, which simply cannot scale, or support only DMA operations, which are not efficient for small data transfers, such as messages to synchronize remote threads, or to configure a remote peripheral [13]. The UNIMEM architecture allows moving tasks and processes close to data instead of moving data around.

Figure 2 shows the ECOSCALE platform. It consists of several WORKERs communicating through a multi-layer interconnection. The exact number of WORKERs per NODE has not yet been fixed, but will be determined by the ability to address the memory on each WORKER in the hardware memory space. Each WORKER is an independent computing unit. It includes a CPU, an FPGA, and an off-chip DRAM memory. The communication and synchronization between WORKERs is performed via a multi-layer interconnection, which enables load and store commands, DMA operations, and interrupts.

Communication overheads between a CPU and an accelerator is one of the greatest challenges, both here and in general. Recently, only explicit memory transfers between host and accelerator memory were supported, as for GPGPUs. Recent advances enable the integration of host and accelerators on the same chip, thus accelerators can directly access host memory.

WORKERs are composed of accelerator blocks which act as a (UNIMEM) *Unit of Compute*. Hence, they can interface directly with any other UNIMEM units of compute, where each unit caches its local data coherently. Each accelerator can also cache its local data and provide coherent access from remote UNIMEM units. If a single accelerator block needs to span across multiple FPGA local memories, then the FPGA units can provide their own coherence schemes independent of UNIMEM.

The reconfigurable resources are typically configured to use physical addresses in order to access shared variables. Since only the OS has access to the physical address space, the intervention of the OS is unavoidable. A dual stage I/O MMU, can resolve this problem by translating virtual addresses to physical addresses in hardware. Using an I/O MMU, the proposed architecture will allow "user-level access" to the reconfigurable accelerators.

Sharing of the limited reconfigurable resources between WORKERs is paramount. Thus, within a NODE, any (local or remote) FPGA may be accessed by any WORKER via the multi-layer interconnect shown in Figure 2.

4. ECOSCALE FRAMEWORK

The ECOSCALE framework is a combination of a runtime and programming interface extensions for the OpenCL framework. The goal for this runtime is to be able to support a range of different application categories including traditional compute-intensive and physics applications, as well as data-intensive applications.

4.1 Language

From a linguistic perspective, this work will be based on the default OpenCL programming model. As noted in Section 2.1, OpenCL is a popular approach to programming and executing computation mainly on CPU/GPU accelerators. Given this popularity, this choice enables this work to be compatible with existing applications, and will require minimal effort from developers in terms of uptake. This work will initially support the C language, but will be as compatible as OpenCL currently is with other languages.

4.1.1 Abstractions

Currently, OpenCL provides a software abstraction for an accelerator known as a device, where each software device has a 1-to-1 relationship with an underlying hardware device. The current range of software devices are `DEFAULT`, `CPU`, `GPU`, `ACCELERATOR`. The work intends to extend this device abstraction to include aggregation of multiple hardware devices. Specifically,

- **WORKER** : Abstraction of the devices found in a worker.
- **NODE** : Abstraction of the devices found in a node.

The motivation for these new devices it to enable a developer to submit work to a software device and allow the runtime to best schedule the kernel computation amongst the hardware devices abstracted by the software device. Scheduling is discussed further in Section 4.3.

Additionally, a developer will be able to express the types of accelerator which will be available within a WORKER or a NODE. For example, a user may wish to have only CPUs, or only FPGAs. Equally, the user may wish to express the locality of kernel execution, such as within a single `WORKER`. This will be done by extending the existing OpenCL API.

The logical conclusion of this approach is to include a `CLUSTER` software device, which would contain `NODE`s. Although linguistically this is a very achievable goal, it is not clear if this would be an effective approach. Given that there are many cluster level scheduling tools, such as slurm [11], it is uncertain if the CLUSTER abstraction would be required. This is an open question and will be explored during the course of the work.

By expanding upon the OpenCL framework, this work will provide the user with extra functionality which builds upon existing idioms, rather than introducing lots of new syntax and concepts. This also has the advantage of increasing compatibility with legacy code.

4.1.2 OpenCL Extensions

As described in Section 4.3, when using a software device which abstracts multiple accelerators it is the responsibility of the scheduler to partition the computation and data placement as best it can. However, as with other efforts noted in Section 2, it should be possible for the developer to provide suggestions to the runtime to assist with this task.

The first extensions are for kernel declarations. The purpose is to indicate to the runtime system which accelerator this kernel is most suited to, as determined by the developer. For example, `#pragma eco <device type>`

The current options include `FGPA` and `CPU`. Conventional OpenCL supported this concept by directly choosing the device to execute the kernel on, however, as a software device now abstracts multiple devices, this feature becomes useful. It should be noted, that one of the main goals of this work to identify the most appropriate accelerator automatically at runtime. This feature is conceptually similar to the `register` keyword in C.

The second set of extensions are to assist in partitioning the data to simplify the process of distributing the data across multiple devices at runtime. OpenCL already enables a developer to specify how a computation should be partitioned into workgroups. This work proposes a similar mechanism to help direct the runtime to partition data. The runtime assumes input and output to kernels to be large contiguous arrays of data. Consequently, the specified number of workgroups will also dictate the number of data blocks. For example,

```
clCreateBuffer(context, REPLICATE,...);
```

In the default situation, the runtime can automatically partition the data. For more complicated situations, such as halo exchanges, the programmer will be presented with a similar set of functions as in Yan et al. [22].

It is important to note that these OpenCL extensions are optional - i.e. the application will complete successfully without them. Thus, the burden on the developer is non-existent if desired. However, application performance in a number of dimensions will improve with their use.

4.1.3 Compilation

As the runtime will use OpenCL, kernels will be compiled at runtime for the CPU. The ECOSCALE project will be creating a custom kernel compiler for an FPGA. As well as generating a bitstream to be encoded onto the FPGA hardware from a kernel, the compiler will output metadata describing the power, performance, and space consumed on

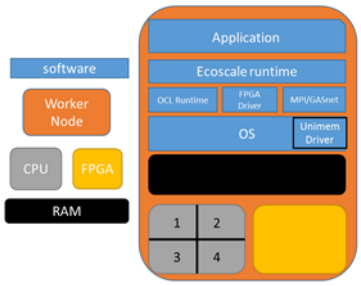

Figure 3: Architecture of the ECOSCALE Runtime

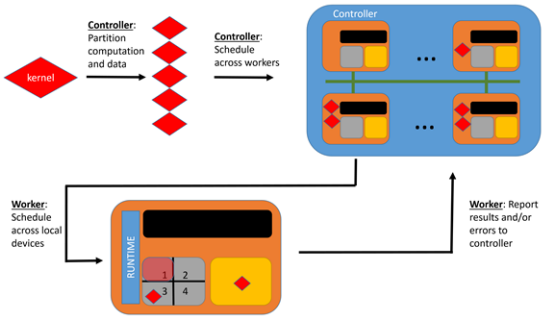

Figure 4: Scheduling of Automatically Partitioned Kernel Invocations

the FPGA hardware. To enable the scheduler to optimise for different metrics, the compiler will generate multiple bitstreams per kernel, each optimised for a different metric. These configurations will be collected together in a library and be made available to the runtime. Due to space limitations, discussion of the FPGA compiler is omitted.

4.2 Software Architecture

As described in Section 3, the hardware platform will consist of several WORKERs located within a NODE. One WORKER per NODE will be used as a controller. Each WORKER will run an operating system and contain an ECOSCALE runtime, which will be executed on a dedicated CPU core. The entire runtime is replicated on each node for robustness, see Section 4.5. The runtime will be based on an open source implementation of OpenCL (POCL [9]).

An ECOSCALE runtime will consist of a communication library, a scheduler, a log of performance metrics of previous executions, access to the FPGA bitstreams and their associated metrics (Section 4.1.3), and an online machine learning component, Figure 3. When initialised, the runtime will be pointed to a library of FPGA bitstreams.

4.3 Scheduling

Within this work, the novel hardware architecture presents the opportunity to explore two aspects of scheduling. The first is hierarchical multi-level scheduling, and the second is individual scheduling within an OpenCL device.

Given the myriad contribution that have been made in the area of software scheduling, this work seeks to leverage as much existing technology as possible. This said, the presence of FPGA accelerators presents a new and novel problem to be solved, particularly in terms of dynamic runtime reconfiguration under the control of the scheduler.

The scheduler will be responsible for the placement of execution and data around the system. Figure 4 shows an example of a scheduling timeline in the proposed system.

4.3.1 Unit of Schedulability

Conventional OpenCL deploys an entire kernel execution on a single device. Given that an average OpenCL application has approximately tens of kernels, this model does not enable full utilisation of a hardware platform with thousands of devices. This would require a programmer to manually break kernels into many smaller kernels, and schedule them.

Alternatively, this work proposes that in such situations, developers submit kernels to the new software devices proposed in Section 4.1.1, and that the runtime partition the kernels into workgroups. It is these workgroups that will be scheduled across the different accelerators. This has three main advantages. Firstly, by using a runtime scheduler, the best hardware platform can be found to execute a workgroup for a given invocation. This placement may be different for different invocations. Secondly, the scheduler will be able to schedule multiple workgroups as units, should this be required, whereas this is a non-trivial task to accomplish manually in an application. Thirdly, as a workgroup is a smaller unit of work compared to an entire kernel, it will be possible to interleave the execution of different workgroups for greater device utilisation. Note that there is no requirement for workgroups to be from the same kernel. Workgroups represent a compromise between scheduling individual work items and entire kernels. The number of workgroups can be programmer defined, or automatically inferred.

The ability to dynamically schedule OpenCL kernel workgroups across heterogeneous hardware platforms is seen as a key research objective of this work.

4.3.2 Preemption

The ECOSCALE runtime will use a *run to completion* model. There are two reasons for this. Firstly, although it is possible to provide preemption on FPGAs by time-multiplexing different workloads, the process of context switching is slow, effecting the performance of the execution, thus nullifying the performance gains of preemption [18]. Secondly, it is the workgroups of the kernels which will be scheduled. These will have a shorter lifetime compared to the kernel as a whole. Consequently, it will be possible to interleave the execution of different works groups as they complete, without requiring preemption.

4.3.3 Device Level Scheduling

In the first instance, each kernel will be scheduled as an entire unit, as is currently done. However, given that average applications consist of tens of kernels, and current HPC platforms consist of thousands or more hardware accelerators, this approach does not scale. Therefore, this work proposes to schedule the workgroups of a kernel across many accelerators in order to concurrently take advantage of as many hardware accelerators as possible.

As noted in Section 4.1.1, this work presents two additional OpenCL device types. As these are software devices, it is the responsibility of the runtime to schedule each workgroup invocation on an accelerator. To make this decision, there are three primary categories that will be considered: performance, power consumption, data locality.

Performance will be established in two ways. Firstly, the runtime will monitor kernel execution on different hardware platforms and use this information for future scheduling decisions. Secondly, the size of the input data will be used as a soft measure of the length of the computation. Unlike other work which uses this information for entire kernels, this will need to be considered for each workgroup.

Power consumption for the CPU will be directly monitored via hardware sensors, however, the FGPA compiler will also provide power consumption data. As the configuration of the FPGA is being precisely generated, this information will be highly accurate.

Finally, the third metric is data locality. As the scheduler is responsible for placing both the computation and the data, it knows exactly where the data will be for the next kernel (set of workgroups) invocation. As it is easier for the scheduler to move a few kilobytes of data representing the computation, rather than the potential gigabytes of raw data for the computation, the scheduler will focus on keeping data in the same physical location for as long as possible. This is similar to the technique used in current GPU programming to reduce the cost of moving data.

Beyond these established metrics, the scheduler will also be optimising for power efficiency and FPGA space occupancy. In order to achieve this, and the previous conditions, machine learning techniques will be used to automatically choose the most appropriate accelerator for an execution. Given that the goal is to classify each schedulable unit to the most appropriate hardware accelerator, support vector machines (SVMs) [2] would be the most sensible approach. SVMs are suited to classifying data into two classes. Many SVMs combined can partition data into multiple classes.

Although machine learning techniques have been used previously in conjunction with OpenCL [7], these have been between a maximum of two devices, where each device was a CPU or a GPU. Given the significant operational differences between a GPU and an FPGA, this work addresses a well-defined advancement in the state-of-the-art.

4.4 Memory Architecture

In conventional OpenCL applications, memory is an important consideration for the developer. When using an accelerator, the developer must manually handle data movements between the host (usually a CPU) and the accelerator (usually a GPU). This is required as the host and accelerator have different physical memory regions. Data movement is often one of the greatest bottlenecks in terms of performance, and requires the careful attention of the developer.

As noted in Section 3, the ECOSCALE hardware will enable memory located in different WORKERs to be accessed via DMA. It is the responsibility of the runtime to control DMA transparently to the application. Thus reducing the latency of memory accesses between different RAM blocks within a NODE, and enable more flexible scheduling options.

As it is impossible to know the number and ordering of memory accesses at compiletime, the ability to transport data around the hardware system for random (and planned) memory access is important.

4.5 Resilience

As the size of the computation and the compute platform increases, it is increasingly necessary to consider the effects of hardware, network, or software failures, and to provide some method of mitigation. The ECOSCALE runtime is designed to address this with respect to continuity of computation and continuity of data.

4.5.1 Computation

The ECOSCALE runtime is a centralised system with one WORKER as a dedicated master, and the others as slaves. However, each worker will posses a full copy of the ECOSCALE runtime meaning that each WORKER may become the controller, even though it is a slave. Also, one WORKER will be nominated, at runtime via a leadership election, as a backup controller to monitor the controller.

It is the responsibility of the master to monitor each slave's health, and to respond appropriately on failure detection. Failure is detected by either explicit failure messages communicated by a slave, such as kernel compilation failure, or implicitly via the lack of a slave's *heartbeat* messages. Here, each slave will periodically send a heartbeat message to the controller indicating that it is still functional and responsive.

If the controller detects that a slave has failed, via one of the above methods, then no further work will be dispatched to that WORKER, and all outstanding work will be reissued to other slaves. The controller maintains accounting information on which computation is located on which WORKER. This information is periodically saved to a file. As workgroups will run to completion, there is no possibility to recover partially completed work, and the workgroup must be reissued. This failure is also reported to a log file for an external manager (human or machine) to monitor.

Should the controller fail there are two possibilities. Firstly, if the backup detects that the controller has failed, it assumes the role of controller, duplicating the accounting information of the controller, performing an election to determine a new backup, and continues operation. The backup also notes these events in a log file. Operation continues as normal. Secondly, if there is some catastrophic failure, the system will crash. Computationally, the log of accounting information which the controller has been logging, enables a checkpoint from which to restart computations.

As the NODEs in a system become networked together, this approach can be replicated up the hierarchy.

4.5.2 Data

The most common way to ensure the continuity of data in the presences of errors is via some distributed checkpoint/restart protocol [5]. This approach sees snapshots of data and computation being taken at different points of an execution. Should an error occur, the data from the last snapshot of the computation is recovered, and the execution restarted from the last known point. A similar approach will be taken in this work, building on existing efforts in this direction [17].

5. CONCLUSION

This paper presents the design for a runtime system to target heterogeneous high performance computer architecture, with an emphasis on power efficiency. By extending and enhancing the existing OpenCL framework, this work will enable the portable performance and ease of programming which is currently offered by OpenCL, combined with the energy efficiency, compute power, and reconfigurable computing offered by the ECOSCALE hardware platform. Also, the ability to transparently schedule OpenCL kernels at the level

of workgroups across heterogeneous hardware platforms is a key innovation that will enable OpenCL applications to take advantage of current and next generation HPC platforms.

In this way, our work will explore and realise a power and computationally efficient way to enable high performance computing, which is intended to lay a foundation for an exascale compute platform.

6. ACKNOWLEDGEMENTS

This research project is supported by the European Commission under the H2020 Programme and the ECOSCALE project (grant agreement 671632).

7. REFERENCES

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[2] B. E. Boser, I. M. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 144–152, New York, NY, USA, 1992. ACM.

[3] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, Aug. 2007.

[4] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[5] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002.

[6] M. P. Forum. MPI: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[7] D. Grewe, Z. Wang, and M. F. P. O'Boyle. OpenCL task partitioning in the presence of GPU contention. In *26th International Workshop, LCPC 2013, San Jose, CA, USA, September 25–27, 2013.*, 2013.

[8] P. Harvey, K. Hentschel, and J. Sventek. Parallel programming in actor-based applications via OpenCL. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 162–172, New York, NY, USA, 2015. ACM.

[9] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg. pocl: A performance-portable opencl implementation. *International Journal of Parallel Programming*, 43(5):752–785, 2015.

[10] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Newnes, 2013.

[11] M. A. Jette, A. B. Yoo, and M. Grondona. Slurm: Simple linux utility for resource management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, pages 44–60. Springer-Verlag, 2002.

[12] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. SnuCL: An OpenCL framework for heterogeneous CPU/GPU clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 341–352, New York, NY, USA, 2012. ACM.

[13] I. Mavroidis, I. Papaefstathiou, L. Lavagno, D. Nikolopoulos, D. Koch, J. Goodacre, V. Papaefstathiou, I. Sourdis, M. Coppola, and M. Palomino. *ECOSCALE: Reconfigurable Computing and Runtime System for Future Exascale Systems*. Institute of Electrical and Electronics Engineers (IEEE), 2016.

[14] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug. 1998.

[15] E. O'Neill, J. McGlone, P. Milligan, and P. Kilpatrick. Shepard: Scheduling on heterogeneous platforms using application resource demands. In *Proceedings of the 2014 22nd Euromicro Intl Conf on Parallel, Dist, and Network-Based Processing*, PDP '14, pages 213–217, Washington, DC, USA, 2014. IEEE Computer Society.

[16] P. Pandit and R. Govindarajan. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 273:273–273:283, New York, NY, USA, 2014. ACM.

[17] A. G. Schmidt, B. Huang, R. Sass, and M. French. Checkpoint/restart and beyond: Resilient high performance computing with FPGAs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 162–169, May 2011.

[18] S. Trimberger. Scheduling designs into a time-multiplexed FPGA. In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, FPGA '98, pages 153–160, New York, NY, USA, 1998. ACM.

[19] Y. Wen, Z. Wang, and M. F. P. O'Boyle. Smart multi-task scheduling for opencl programs on CPU/GPU heterogeneous platforms. In *21st International Conference on High Performance Computing, HiPC 2014, Goa, India, December 17-20, 2014*, pages 1–10, 2014.

[20] S. Wienke, P. Springer, C. Terboven, and D. an Mey. 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.

[21] S. Xiao, P. Balaji, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W. chun Feng. VOCL: An optimized environment for transparent virtualization of graphics processing units. In *In Proc. of the 1st Innovative Parallel Computing (InPar*, 2012.

[22] Y. Yan, P.-H. Lin, C. Liao, B. R. de Supinski, and D. J. Quinlan. Supporting multiple accelerators in high-level programming models. In *Proceedings of the 6th Intl Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, pages 170–180, New York, NY, USA, 2015. ACM.

[23] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. UPC++: A PGAS Extension for C++. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1105–1114, May 2014.