Compilation of OpenCL Programs for Stream Processing Architectures

Pedro Moreira
Instituto Superior Técnico
Email: pedro.m.moreira@tecnico.ulisboa.pt

Abstract—The stream architecture and the stream execution model optimize memory usage while minimizing processor idle time by exploring the multiple types of locality in stream applications: temporal, spatial, and instruction locality. These are the main focus of this work.

Heterogeneous systems can contain various computing devices that implement different architectures, and it can be an arduous task to take the architectural details of each device into account when developing applications for these systems.

This work proposes StreamOCL, an implementation of the OpenCL framework, and its goal is to provide a programming environment for heterogeneous systems, targeting stream devices.

The proposed approach also enables applications to explore the advantages of stream computing, without requiring them to be specifically implemented for stream devices. The polyhedral model is used to analyse memory access patterns of OpenCL kernels and capture the locality of OpenCL applications.

The results validate the approach since the current implementation was able to capture the access patterns of a baseline stream application.

A 2x speedup was also achieved in kernel execution time on the current testing platform by scheduling kernels according to the stream execution model. In spite of the positive results, the overhead of computing memory dependences is currently not compensated by the performance gain in kernel execution.

Index Terms—Stream Architecture, OpenCL, Polyhedral Analysis, Low-Level Dependence Graph

I. INTRODUCTION

To answer the need for parallel processing power, multicore devices have been developed that are capable of executing many arithmetic intensive tasks in parallel. In this particular context, several architectures have been developed in the latest years, each specifically designed for a different purpose.

This work focuses on the stream architecture, and the stream computing model, which, by decoupling computation tasks from communication tasks, allows exposing additional parallelism levels and leads to a faster application execution. In particular, in multicore computing devices, there is the problem of memory bandwidth and latency, since there are many cores competing for the same memory resources. Stream architectures aim at solving this problem by using a different approach to transferring data from device memory to the cores, and scheduling tasks in a way that minimizes memory transfers between the cores and the device’s global memory. The stream architecture maximizes the use of the processing cores in the execution of programs that express the principles of locality in different ways: when different instructions are executed over the same data (instruction locality), when there are data dependences between different tasks (temporal locality), and when data is processed sequentially, as in a stream (spatial locality). Stream architectures assume a specific programming model, that allows programmers to make explicit the various elements that express the locality of stream applications. These elements are, typically, the definition of tasks or kernels that process data, the data streams that those tasks consume and produce, and the relations between those tasks.

The challenge in using various devices, of diverse architectures, in heterogeneous systems [4], [27], [23], [1], is that it is difficult to develop software that is capable of using the different devices efficiently. The result of the compilation process necessarily varies from one device to another, as they can have different instruction set architectures (ISAs), and different memory organizations. Also, the execution models can be different depending on what the architectures were designed for. In the case of stream architectures, it is possible to accomplish a very efficient use of memory, but only for applications that follow the stream execution model.

There are software development tools for multicore devices (such as CUDA or the Stream Virtual Machine), but they can only be used for a restricted subset of device architectures. The use of heterogeneous systems leads to the need for a programming environment that allows the development of applications for different devices and, yet, allows the efficient use of their computational resources, despite the architectural differences that distinguish them. OpenCL is a programming framework that aims to solve precisely this problem. It abstracts the architectures, programming models and execution models of computing devices into a single, unified framework, that allows applications to use the computational power of heterogeneous systems in a transparent way. OpenCL provides an application programming interface (API) that serves as an abstraction for the communication with the computing devices, and a low-level programming language, that are architecture-independent. By providing an abstract front-end implemented by architecture specific back-ends for the different target devices, OpenCL enables the development of portable applications that exploit the computational power of heterogeneous systems.

A. Objectives

The purpose of this work is to provide an implementation of the OpenCL framework for stream devices. Implementing the OpenCL framework also comprises the implementation a
device driver that can make the communication between the host CPU and the target device. The implementation of the OpenCL interface has to take the characteristics of the stream architecture into account, and enable OpenCL applications to use the resources of stream devices efficiently and according to the stream model.

The result is an implementation of the OpenCL framework that adapts the OpenCL model and enables client applications to exploit stream devices, by implementing a set of features that capture the locality in OpenCL applications.

II. RELATED WORK

A. Stream Architecture

With the introduction of programmable graphics processing units (GPUs [20], [6], [22]), stream processing [24], [17], [16], [5], [7] has gained importance. In particular, by decoupling computations with communication tasks, it allows a more efficient exploitation of the application parallelism [3], [26].

The stream execution model simplifies overall control by decoupling data communication operations from the actual computations. Moreover, it allows for a more efficient exploitation of fine-grained data-level parallelism, and simplifies the exploitation of task-level parallelism, by having different cores executing different operations over streaming elements. The stream model allows for kernel execution to occur in parallel with data transfers between the various levels of memory hierarchy, minimizing the time that the cores are idle waiting for data to be fetched. It also allows to exploit data locality in the sense that each kernel consumes data while it is produced by other kernels.

Stream architectures are typically composed of a hierarchy of memory levels and clusters of processing elements.

Stream processors rely on especially devised stream controllers for managing data transfers from global memory to the Local Register Files (LRFs). These transfers are often made through an intermediate memory level called the Stream Register File (SRF), through which all memory transfers between device global memory and the LRFs are made. While the stream controller manages which memory chunks need to be copied from device memory to which cluster’s LRF and from each LRF to which memory addresses in device memory, each cluster simply performs read operations from input streams and write operations to output streams.

B. OpenCL

OpenCL [11], [13], [15] is a standard for heterogeneous parallel processing systems and does not strictly follow a stream processing model. It was specifically designed to support a diverse set of devices, such as multicore CPUs, special purpose accelerators (e.g., GPUs), and even FPGAs. For this, the OpenCL specification abstracts the architectures and the execution models of multicore heterogeneous devices, and provides a programming model that is able to unify the development of software for any computing device that is conformant with the standards of its specification.

The OpenCL execution model is divided in two parts: the host program and the kernel execution that occurs on the guest devices. The host program is executed on the CPU and manages platforms, devices, and issues commands to the devices such as memory transfers and kernel execution. On the guest side, the devices execute the commands issued by the host program.

Each work-item is uniquely identified by a tuple of IDs. When a kernel is scheduled, a coordinate space of work-item IDs is defined, called NDRRange, that represents the IDs of all work-items that will execute that kernel instance. Each work-item can be uniquely identified either by its global position in the NDRRange or by its work-group ID and local position in the work-group.

Studies have been made [25], [19] with the purpose of evaluating whether an OpenCL application attains portable performance. The conclusion is that a single OpenCL code cannot efficiently explore the architecture of different devices. CUDA’s compiler generates more efficient code than the default OpenCL compiler, but with some optimizations that the OpenCL compiler can do automatically, it should be possible to achieve equivalent results to those of CUDA.

C. Kernel Access Pattern Analysis and Optimizations

For the stream controller to manage the multiple data transfers that occur between the device memory and the processing elements, and between the processing elements themselves, it is required that a description of the memory access patterns of the kernels is provided. By knowing which regions of memory are going to be used by which cores the stream controller is able to prefetch data from device memory to the cores and efficiently transfer the intermediate values of a program from one cluster to another in a stream-like fashion.

Static analysis [14] is made by parsing the source code or read a compiled program, searching for the instructions that perform accesses to memory and representing the addresses they use. Access patterns are represented with abstract mathematical models that allow to find parallel regions or dependences between memory accesses. This allows compilers to make optimizations such as vectorization and polyhedral optimizations.

The polyhedral [2], [10], [9] model is used for intermediate compiler representation of memory operations in static control parts (SCoPs) and allows to perform code analysis and code generation for many ends, such as memory access dependence analysis, vectorization, parallelization and scheduling. SCoPs are sets of consecutive statements in a program that can be inside loop control statements. These loops have the restriction of only being bound by affine constraints and constant values, or by the iteration counter variables of the parent loop.

III. ENABLING OPENCL IN STREAM DEVICES

Although OpenCL can be used to express relationships between kernels, the way dependences between kernel execution commands work in OpenCL imply that the these commands cannot be executed in parallel, such that a kernel
command can only begin after the previous command has completely finished. This imposes serious constraints towards the implementation of the stream execution model. Accordingly, it is necessary to conceive a different implementation of the OpenCL execution model that does not suffer from this constraint.

While the stream architecture and the OpenCL architecture are different, it is possible to establish some correspondence between the elements of the two. It is important to identify these relations to select which elements from the OpenCL generic model can be mapped on to the stream architecture, and which need to go through an more careful abstraction process. The elements that need to be adapted are the processing elements (PEs), the memory hierarchy and how PEs are able to access memory.

The global memory as described in the OpenCL specification can be implemented with the global memory in a stream device, although there are important aspects that need to be considered related to how accesses to memory are performed. While in OpenCL any work-item can randomly access a memory address in the global address space, the stream model comprises predictable [8] (preferably sequential) accesses to memory. Also, in OpenCL PEs request data from memory in a similar way to GPGPUs [18], unlike stream devices where memory transfers are performed by a special stream controller. To make global memory accessible in stream devices, the OpenCL implementation will need to describe the memory accesses made by work-items in the form of data streams so that the stream controller can feed the PEs with data from global memory.

Local memory can be made available if the device provides a memory module for each cluster. If not, further adaptation is needed such as requiring the OpenCL implementation to capture the access patterns of each kernel and assigning data streams that transfer the local data of each work-group between the clusters and the SRF.

Private memory in stream devices can be easily achieved using either the registers in each processing element or, if the device implements a local memory module on each cluster, allocating a fraction of local memory for each processing element.

By analysing kernel code, it is possible to capture memory access patterns (see section II-C) and predict which memory regions a given work-item is going to access during it is execution. The NDRange assigns coordinates to each work-item and the execution of that work-item uses those coordinates to compute the memory regions it is going to access. This way, OpenCL has an implicit way of mapping work-items to memory addresses. By merging the access patterns of all work-items in a work-group, we can obtain the mapping between the coordinates of a work-group in the NDRange and the memory addresses that its execution will use.

The execution flow of the proposed solution is illustrated in figure 1. OpenCL applications use the OpenCL API to compile and execute kernels. The first step in the life time of an application is to compile the kernels. The OpenCL compiler receives the kernel code from the application, compiles it and returns a kernel object. In the proposed implementation, the compilation process also includes a code analysis phase, where memory access patterns are captured from kernels using static analysis, which results in a representation of the kernel access patterns. The context (arguments and NDRange) in which the kernel will run is unknown at compile time, so the generated access patterns are generic, in function of a kernel context that will only be known in the future.

After computing all dependences between work-groups of the kernels that have been enqueued by the OpenCL application, a low level dependence graph (LLDG) can be generated, where each node of the graph is a work-group of a kernel instance and each edge is a causal relation (dependence).

A. Memory Access Patterns
At compile time, the polyhedral analysis technique is used to capture memory access patterns from static control parts (SCoPs).

1) Intersect the scattering function of the memory access with the domain of the statement. For example:

\[ P_{output}(n, id, of f set) = \{10 * (of f set + id) + i : 0 <= i <= n} \]

2) Find the origin of the variables that \( P \) depends on. The representation of the access pattern keeps the information of where variables derive from, as will be useful later on.

3) Find which buffer is being accessed. Since a kernel can access multiple memory buffers (in global memory or constant memory), it is important to know the access patterns for each of them individually to be able to compute dependences between kernels per buffer.

4) Distinguish between read and write patterns and store them separately.

5) Make sure all memory access instructions have been analysed.

This process will result in a set of access patterns in the form of mathematical representations in function of the NDRange.
and arguments of the analysed kernel:

\[
P(\text{Arguments}, \text{NDRange}) = \{P_{\text{buffer}1} \cup \ldots \cup P_{\text{buffer}N}\}
\]

At kernel setup time, the OpenCL application sets the context \(C\) in which the kernel will be called. The result will be a function that, given the NDRange coordinates of a work-group, returns the set of addresses that that work-group accesses. Since the memory access patterns that were captured at compile time only describe the addresses accessed by a single work-item, this process also joins the accesses of work-items into one coalesced work-group access pattern.

\[
C(\text{WorkGroupID}_0, \ldots, \text{WorkGroupID}_{D-1}) = \{\text{KernelArguments} \cup \text{CoalescedNDRange}\}
\]

The context for scalar kernel arguments consists in a list of argument names and respective values, each corresponding to a parameter in \(P\) that was tagged at compile time. For work-item functions however, there are various special cases that need to be considered for coalescing the accesses of all work-items in a work-group.

After being generated, \(C\) will be intersected with \(P\), resulting in the actual runtime access pattern \(R\) of the given kernel instance:

\[
R(\text{WorkGroupID}_0, \ldots, \text{WorkGroupID}_{D-1}) = \{P \cap C\}
\]

B. Run Time Dependence Computation

Given a set of kernels, it is necessary to know which work-groups from those kernels overlap their memory accesses, and then establish a dependence between them, in order to create graphs of causal relations between kernels. The dependence function can be obtained using ISL’s flow computation feature [28], which is a feature specifically designed for memory dependence analysis and polyhedral optimizations.

The scheduling of kernels and memory transfer management are important to make an efficient use of device resources (processor usage, memory space and bandwidth). The memory access pattern analysis in combination with the generated LLDO can be used to decide, at runtime, suitable scheduling of threads and memory transfers. To follow the stream execution model, data should be transferred from the DRAM to the SRF only once, be used by all work-items that require it, and never be used again. Kernels should also be scheduled in a way such that the data they produce to global memory can be reused shortly after by other kernels that are running in parallel in different clusters. The memory access pattern analysis and the LLDO described in the previous section can be used in combination to decide, at runtime, which work-groups of which kernels should be started and which memory transfers should occur to feed the clusters with data in an efficient manner. Accordingly, for any given program, the LLDO has information on the scheduling constraints between kernels and memory transfers (between host and device). Also, for any given work-group, the kernel instance access pattern describes which memory addresses are going to be read or written. The knowledge of this information enables the execution of the stream model, and can also allow for different optimizations: prefetching, avoiding unnecessary transfers to global memory, and scheduling.

The LLDO expresses producer-consumer relationships between work-groups, but does not necessarily describe at which ratios data transfers occur between producers and consumers. A possible solution for this problem is the implementation of a dynamic run time load balancing mechanism that can profile the clusters, associate their performance with the LLDO nodes and find the balance between producer and consumer ratios by measuring how much time work-groups are taking to produce and consume data.

IV. IMPLEMENTATION OF STREAMOCL

StreamOCL is an implementation of the OpenCL standard specialized for stream processing architectures. It can be built either as an Installable Client Driver or as a standalone OpenCL implementation.

This work was made with the intent of developing an OpenCL implementation and device driver for a stream-based accelerator being developed at the SiPS group and implemented in FPGA.

StreamOCL implements all the functionality of the OpenCL runtime layer and some of the compiler functionality that is not dependent on the underlying hardware instruction set and architecture (kernel code parsing and analysis). This includes the management of OpenCL contexts, devices, command queues, events, memory buffers, kernels and commands. StreamOCL makes calls to the device driver to communicate with the device (executing kernels and memory transfers). StreamOCL itself is not specialized in any particular device architecture or processor instruction set architecture (ISA). It is a generic implementation for stream devices, and the functionality that is specific of the target device is implemented by the device driver at a lower level.

The device driver is responsible for generating code for the target ISA, managing device memory, transferring data from host to device and from device to host, loading kernels and their context (kernel arguments and NDRange) to the device and runtime thread scheduling. It implements an API that
abstracts the operations that can be requested from the device so that different implementations or different devices can be made available through a single API.

As illustrated in figure 2, the architecture of the StreamOCL library comprises the OpenCL runtime implementation, the device driver library, the Integer Set Library (ISL) and the LLVM [21] compiler infrastructure.

The StreamOCL software stack uses the LLVM infrastructure to implement all the functionalities related to kernel compilation and analysis. OpenCL kernels (written in OpenCL C) are parsed using Clang, which is an LLVM front-end capable of parsing different languages including OpenCL C. Clang generates LLVM IR that is used by the rest of the libraries in our implementation. LLVM is used by the device driver to generate code for the target device using a back-end plugin that is able to generate code for the target ISA.

The compilation process of OpenCL kernels in StreamOCL includes parsing the code provided by the application, getting information about which kernels are present in the code, capturing memory access patterns and generating code.

The runtime layer of StreamOCL was conceived with the stream execution model in mind. Thus, it implements an execution model that allows for the parallel execution of commands, yet respecting the synchronization constraints of the OpenCL model.

The implementation of the OpenCL command queue keeps the commands that are enqueued and submitted to device according to the OpenCL specification. But in StreamOCL, command queues also keep the low level dependence graph (LLDG) with the commands that have been enqueued. When a new command is enqueued, the low level dependences of its subcommands are computed against the commands that have been submitted to the same command queue and the LLDG is extended with the new subcommands.

StreamOCL implements the OpenCL kernel memory access pattern analysis technique previously described. It uses Polly to perform the polyhedral analysis and ISL to represent the access patterns. An LLVM Pass was implemented that traverses through all functions of an LLVM Module and uses Polly to gather all the access patterns from all SCOs.

For each SCoP found, StreamOCL first checks the context parameters of that SCoP, which are the variables that are used in the SCoP to index the memory positions accessed. For each variable, StreamOCL gets the LLVM Value that variable represents from Polly. StreamOCL then finds the origin of the variable, that can be of two types: a value received as an argument of the kernel, or the result of a call to an OpenCL work-item function. If it is an OpenCL function call, StreamOCL finds which function is being called and, if that function receives an argument, the value of that argument. The current version of StreamOCL only supports OpenCL function calls made with constant values as argument. These variables are tagged to be easy to identify as NDRRange-dependent variables.

When a kernel execution command is enqueued, the general access pattern, captured at compile time, is specialized for the context in which the enqueued kernel was called. The values of the arguments and the parameters of the NDRRange are used to create a context in the form of an ISL map. The context is applied to the general access patterns of that kernel for each buffer it receives. This step results in a mapping between each buffer argument of the kernel, the corresponding OpenCL memory object, and the access pattern of that kernel instance for that buffer.

In StreamOCL, there are two levels of synchronization between OpenCL commands: global and local. Global synchronization is the same as in the OpenCL execution model, where commands are not allowed to run in parallel if there is a causal dependence between them. These dependences are set by the OpenCL application using event synchronization. Local synchronization is made at the level of the subcommand, where each subcommand is allowed to run in parallel with subcommands of the causally dependent command as long as its local dependences are satisfied. The subcommands and the local dependences between them are the nodes and edges of the LLDG respectively. If StreamOCL is able to generate an LLDG between two commands, then the causal dependence between those commands becomes a local dependence, rather than global.

The algorithm that builds the LLDG finds the local dependences between commands recursively using a breadth-first search. The goal of this algorithm is to compute the LLDG without redundant dependences between subcommands of different commands. It is assumed that the queue execution mode is always out-of-order (general case) so the global dependences between commands are defined by the list of OpenCL events (event_wait_list) received from the OpenCL application. For each enqueued kernel, a counter is used to count the number of work-items for which a dependence has been found. When this counter reaches the amount of work-items in the NDRRange of the kernel instance, then all necessary dependences have been found and the algorithm ends. Otherwise, the algorithm keeps searching for dependences with commands that were enqueued previously until there are no more commands left. A marker is also kept for each subcommand, referencing the command with which a local dependence was found. This excludes redundant dependence computations.

StreamOCL aims to be in-between target independence and architecture specific. While the implementation of OpenCL is not specific to any target device in particular, it implements stream optimization features specific to stream devices. To allow the StreamOCL layer to use different devices, an abstract device driver interface was created. Any implementation that offers the same interface should be usable by StreamOCL and each implementation is responsible for the details of the underlying hardware.

The device driver interface offers the essential functionality that would be required of any generic target device. The current version offers the following functionality: hardware discovery, memory management, kernel compilation, kernel setup and scheduling.
The architecture of processors and memory hierarchy is similar to the one in OpenCL. Processing elements are grouped in clusters. There are three levels of memory: private, local and global. Private memory is private to each core, local memory is shared only between the cores of each cluster and global memory is shared between all cores.

In order to run kernels, the client application needs to request the creation of a CommandExec object, which represents a kernel and the arguments with which the kernel is called. Thread execution requests are submitted as Jobs. A job is a group of threads that run the same CommandExec on the same cluster. In the current version, it is assumed that all the jobs that have been submitted for execution can be run in parallel, so the driver does not guarantee any order of execution.

The current device does not have local memory, so global memory is used instead. However since each cluster needs its own local memory, a different chunk of global memory is allocated for each cluster.

A. Kernel Compilation

Compiling kernels comprises analysing and transforming the provided LLVM IR and generating the target binary.

The analysis passes retrieve information about which kernels are present in the provided LLVM Module and which arguments kernels receive. For each kernel, a structure is built that stores this information. Each kernel is assigned with a unique ID number.

Another analysis pass searches for uses of local memory in each kernel. Because there is no actual local memory in this particular device, it is necessary to know how much memory needs to be allocated for a cluster to run a given kernel if that kernel uses local memory. This pass gathers that information.

The code transformation passes prepare the kernel to become conformant with the memory organization of the device: local memory mapping on global memory, linking intrinsics and generation of the main function. After these steps, the LLVM Module is ready for binary code generation.

V. EVALUATION

Since the target FPGA device is unable to execute programs according to the stream model in its current version, no tests were run in the context of this work. However the current implementation of the device driver has been used in another work [12] to evaluate an implementation of OpenMP (an API for multiprocessing) on the same target device.

Three case studies were evaluated: computation of prime numbers, computation of factorials and a program with a global synchronization point between threads. In the computation of prime numbers, speedups were achieved in proportion to the number of threads that the application was using. In all the performed tests, the device driver was able to manage device memory on demand, and perform data transfers from/to the device correctly. Kernel setup was reported to be successful as well, which means that the driver was able to set the context of each kernel in the target clusters and correctly assign a thread ID to each core. Finally, the driver also scheduled the submitted jobs maximizing processor usage when using the non-synchronized scheduling mode.

The current version of StreamOCL enables the execution of OpenCL applications on devices that implement the Device Driver interface. This makes it a generic and portable implementation that can use the computational resources of any device, provided an implementation of the driver interface for that device. Furthermore, StreamOCL implements a set of features that enable the stream execution model: capture of OpenCL kernel memory access patterns, computation of kernel instance access patterns at runtime, and computation of the low-level dependence graph at runtime.

A. Dependence Analysis

The first case study is a baseline application that simply performs memory transfers sequentially. The produced LLDG perfectly expresses the local dependences between the work-groups of different kernel instances, as the work-groups of different kernels that access the same chunk of memory are connected with low-level dependences. The second case study is the implementation of the Needleman-Wunsch algorithm in OpenCL. It uses a matrix and computes the best alignment between two given sequences by comparing each pair of elements in the sequences. For this, given two sequences A and B of sizes N and M, the algorithm constructs a table of size N x M. Unlike the baseline memory transfer example studied before, the access patterns in the Needleman-Wunsch kernel are not trivial. The data is organized diagonally and the memory addresses that are accessed are calculated using a formula that depends on both the NDRRange and the arguments the kernel receives. Moreover, each kernel instance executes on a different context with a different NDRRange and argument values. Nonetheless, StreamOCL is capable of capturing the memory access patterns of the kernel at compile time, generating the runtime access patterns when kernels are enqueued, and uses that information combined with OpenCL events synchronization to compute the LLDG represented in figure 3.

![Fig. 3. LLDG for the Needleman-Wunsch algorithm. Each color represents an OpenCL kernel instance.](image)

There is an evident correspondence between the generated LLDG and the matrix used by the Needleman-Wunsch algorithm. It is also important to note that the redundant dependences between each diagonal are not represented in the LLDG. Since each diagonal N depends on the diagonal N – 1, then it also depends on diagonal N – 2 by transitivity. As
shown in figure 3, StreamOCL did not generate any redundant dependences.

B. Setup Time

The results show that kernels perform faster when executing with stream mode enabled, which proves that StreamOCL is capable of exploiting the locality in the tested application. However, the overheads associated with the computation of the LLDG have a greater impact on the overall execution time than the achieved speedup. Notwithstanding, it should be noticed that the kernel used for this demonstration only makes two accesses to memory, and does not perform any computations at all, taking very little time to execute. Therefore, it is expected that for more complex kernels that perform more computations and accesses to memory, the overhead added by StreamOCL becomes less significant compared to kernel execution time.

Finally, it is important to note that while kernels are executed on the device, the computation of the LLDG is performed on the host CPU, so the overheads described in this section do not affect the execution times of programs on the device side. It is possible to parallelize generation of the LLDG with kernel execution, such that while the device is executing a set of kernels, the host CPU is computing the dependences for the kernels that will be executed next.

C. Baseline Memory Transfer

The baseline memory transfer application was also profiled in more detail regarding kernel execution times to evaluate the benefit of the stream execution model in the current testing platform.

The graph in figure 5 shows the kernel execution time measured for the baseline memory transfer application per iteration. The first kernel instance is the first to access the input buffer and needs to load every region of memory from outside the CPU cache (i.e. global memory). In this case there is no significant difference in execution times between the normal mode and the stream mode. For the following kernels, however, memory access times are much smaller in the stream mode since they take full advantage of the cache system. In the current experimental platform the execution times of kernels running in stream mode were approximately two times faster than in normal mode.

This test shows how StreamOCL can increase the efficiency of memory usage by simply scheduling kernel work-groups according to the stream execution model, and gaining speedup without increasing the number of cores executing the kernels.

VI. CONCLUSION

The objective of this work is to enable the use of the OpenCL framework in stream architectures, adapting the OpenCL execution model and extending some of its features to match the requirements of the stream model. The proposed solution captures memory access patterns from OpenCL kernels and establishes low-level dependences between different kernels, capturing the task parallelism inherent to stream applications. The polyhedral model was used to capture memory access patterns in OpenCL kernels, and was further adapted to take into account the way OpenCL maps data with work-items. The information of kernel access patterns is used, at runtime, to compute a low-level dependence graph (LLDG) and establish producer/consumer relationships between work-groups of different kernels. The LLDG can then be used by scheduling algorithms to perform the stream execution model.

In order to enable OpenCL applications to use stream devices, StreamOCL implements a set of unique features that allow the OpenCL runtime to perform the stream execution model. In many applications, it allows a more efficient use of hardware resources and a faster application execution.

The device driver that was implemented for the target FPGA device has been used not only for StreamOCL, but also for an implementation of the OpenMP framework for the same target device.

Capturing memory access patterns was successful as StreamOCL was able to generate the optimal LLDG for the
tested applications, even for the case where kernels had non-trivial access patterns, the input data was organized diagonally and kernels were being called with different arguments and NDRanges. Regarding kernel execution time, we verified that by using the LLDG to schedule threads according to the consumer/producer relations between work-groups, we were able to use the memory system efficiently by maximizing cache hit-ratio, resulting in a $2 \times$ speedup.

### A. Future Work

Further investigation needs to be done regarding the generation of the LLDG, which, for lightweight kernels, can lead to important overheads in the current implementation of StreamOCL. Since memory access patterns of kernel instances are represented mathematically by sets of integers with affine constraints, an algebraic method should be investigated from which StreamOCL can extract the dependences between work-groups without using the current brute force method.

Finally, being able to describe memory access patterns in a way that can be used efficiently by the device scheduler to make prefetching of data and to avoid redundant accesses to global memory would also contribute to further maximize the efficiency of memory usage.

### REFERENCES


